Carlos Eduardo Sanoja

Project: Perception Pick & Place

Required Steps for a Passing Submission:

1. Extract features and train an SVM model on new objects (see **pick_list_*.yaml** in **/pr2_robot/config/** for the list of models you'll be trying to identify).
2. Write a ROS node and subscribe to **/pr2/world/points** topic. This topic contains noisy point cloud data that you must work with.
3. Use filtering and RANSAC plane fitting to isolate the objects of interest from the rest of the scene.
4. Apply Euclidean clustering to create separate clusters for individual items.
5. Perform object recognition on these objects and assign them labels (markers in RViz).
6. Calculate the centroid (average in x, y and z) of the set of points belonging to that each object.
7. Create ROS messages containing the details of each object (name, pick_pose, etc.) and write these messages out to **.yaml** files, one for each of the 3 scenarios (**test1-3.world** in **/pr2_robot/worlds/**). See the example output.yaml for details on what the output should look like.
8. Submit a link to your GitHub repo for the project or the Python code for your perception pipeline and your output **.yaml** files (3 .yaml files, one for each test world). You must have correctly identified 100% of objects from **pick_list_1.yaml** for **test1.world**, 80% of items from **pick_list_2.yaml** for **test2.world** and 75% of items from **pick_list_3.yaml** in **test3.world**.
9. Congratulations! Your Done!

Rubric Points

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

The main goal of this part is to know about filtering and segmentation and implement this tools to manipulate objects separately. As a core of this project, we are going to work with point clouds obtained from an RGB-D camera. We capture the point clouds through sensor_stick in ROS.

First, we have to downsample the point cloud by applying a Voxel Grid Filter. This filter reduces the points in the point cloud without loosing the essential information required to obtain each of them separately. In addition to, this filter help us to reduces the computational time to preccessing the image.

```
def voxel_grid_downsampling(cloud_data,LEAF_SIZE = 0.01):
    vox = cloud_data.make_voxel_grid_filter()
    # 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()
    return cloud_filtered
```

If we know where the objects must be, we can reduce the area of searching applying a Passthrough filter. This filter help us to crop or limit the area of interest of the external information. To perform this filter we are going to limit in the Z-AXIS (from 0.6 to 1.1) and the Y-AXIS (from -1 to 0.5).

```
def Passthrough_filter(cloud_data,axis_min = 0.6,axis_max = 1.1,filter_axis = 'z'):
    # Create a PassThrough filter object.
    passthrough = cloud_data.make_passthrough_filter()
    # Assign axis and range to the passthrough filter object.
    passthrough.set_filter_field_name(filter_axis)
    passthrough.set_filter_limits(axis_min, axis_max)
    # Finally use the filter function to obtain the resultant point cloud.
    cloud_filtered = passthrough.filter()
    return cloud_filtered
```

Finally, we implement RANSAC plane fitting to separate the table and the objects. Here we are going to obtain two points cloud data separately (cloud_table and cloud_objects). I found good results with max_distance=0.01.

```
def RANSAC(cloud_data,max_distance = 0.01):
    seg = cloud_data.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
    seg.set_distance_threshold(max_distance)
    # Call the segment function to obtain set of inlier indices and model coefficients
    inliers, coefficients = seg.segment()
    cloud_table = cloud_data.extract(inliers, negative=False)
    cloud_objects = cloud_data.extract(inliers, negative=True)
    return cloud_table,cloud_objects
```

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

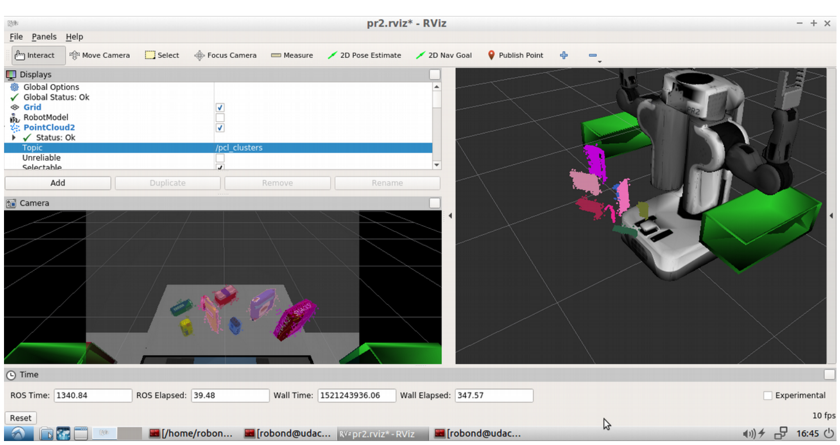
The main goal of this part is to get in touch with nodes and topics in ROS. We have to create publishers to publish the segmented table and the objects as point clouds. After that, apply Euclidean clustering on the objects point cloud. Finally, we have to assign for each object an unique color. Finally publish the colored cluster cloud on a separate topic.

>>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_clusters", PointCloud2, queue_size=1)

```
Def Euclidean_clustering(cloud_objects,cluster_tolerance=0.02, min_cluster_size=30, max_cluster_size=40000):
    # Apply function to convert XYZRGB to XYZ
    white_cloud = XYZRGB_to_XYZ(cloud_objects)
    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)
    # NOTE: These are poor choices of clustering parameters
    # Your task is to experiment and find values that work for segmenting objects.
    ec.set_ClusterTolerance(cluster_tolerance)
    ec.set_MinClusterSize(min_cluster_size)
    ec.set_MaxClusterSize(max_cluster_size)
    # Search the k-d tree for clusters
    ec.set_SearchMethod(tree)
    # Extract indices for each of the discovered clusters
    cluster_indices = ec.Extract()
    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)
    return white_cloud, cluster_indices, cluster_cloud
```

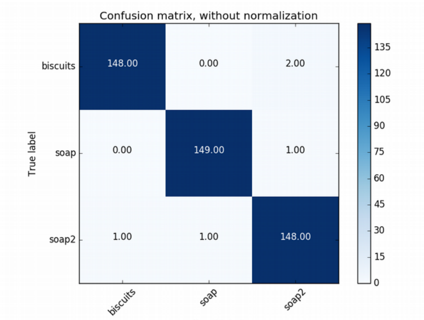
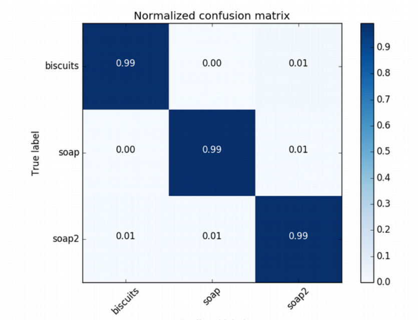
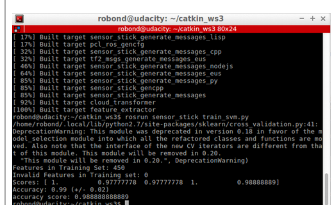
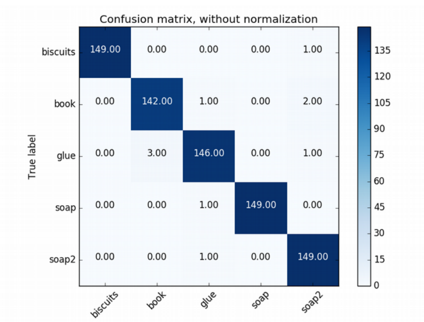
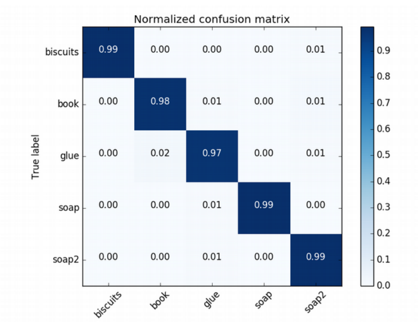
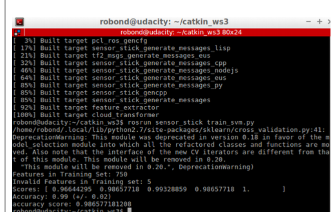
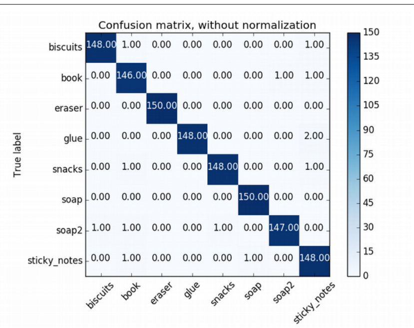
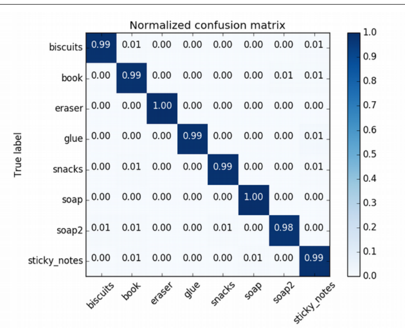
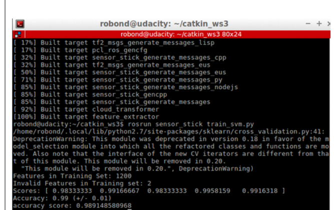


Complete Exercise 3 Steps. Features extracted and SVM trained. Object recognition implemented.

The main goal of this part is to identify each object from the colored objects point cloud. First we have to extract the features. In this project we are considering as features the color and shape characteristic. To understand this information we use a histogram to make a translate and give sense for the model (something to make each object unique).

```
>>color_hist = compute_color_histograms(pcl_cluster_msg, using_hsv=True)
>>normals = get_normals(pcl_cluster_msg)
>>normal_hist = compute_normal_histograms(normals)
>>feature_vector = np.concatenate((color_hist, normal_hist))
```

After that we can train the model to make the prediction. To do the trainning it was clearly that the computational power and time of running is hug for large objects poses. For computer issues, i took just 150 poses to train the three models and i got the following results:

Model	Confussion Matrix	Confussion Matrix Normalized	Accuracy Score
1			
2			
3			

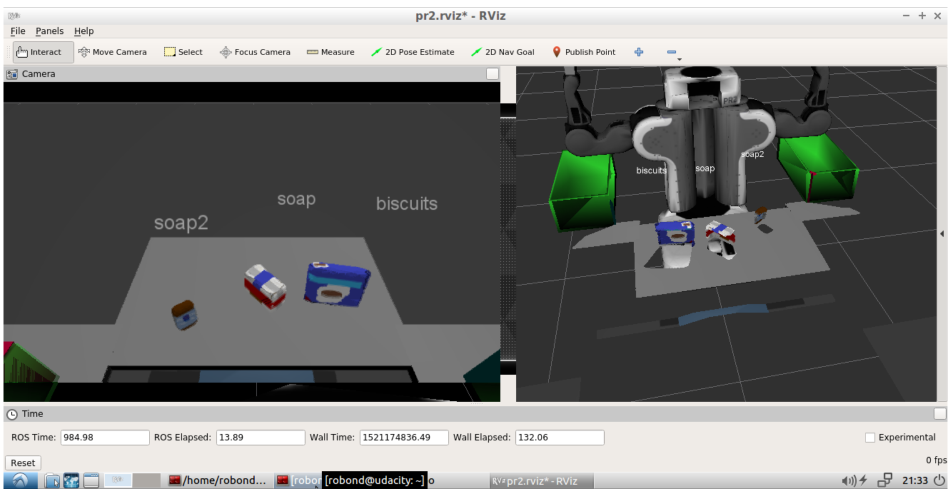
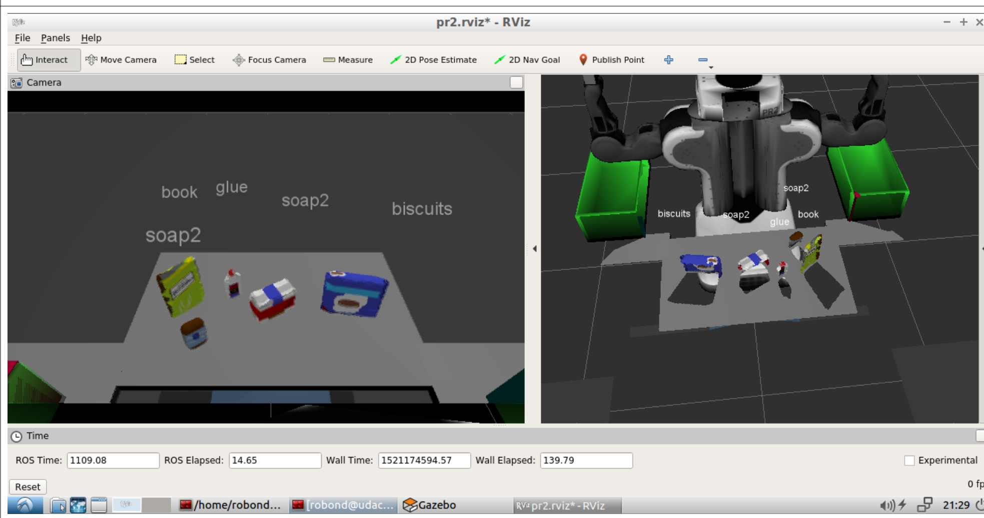
Now, we can implement the prediction and add a label for each one.

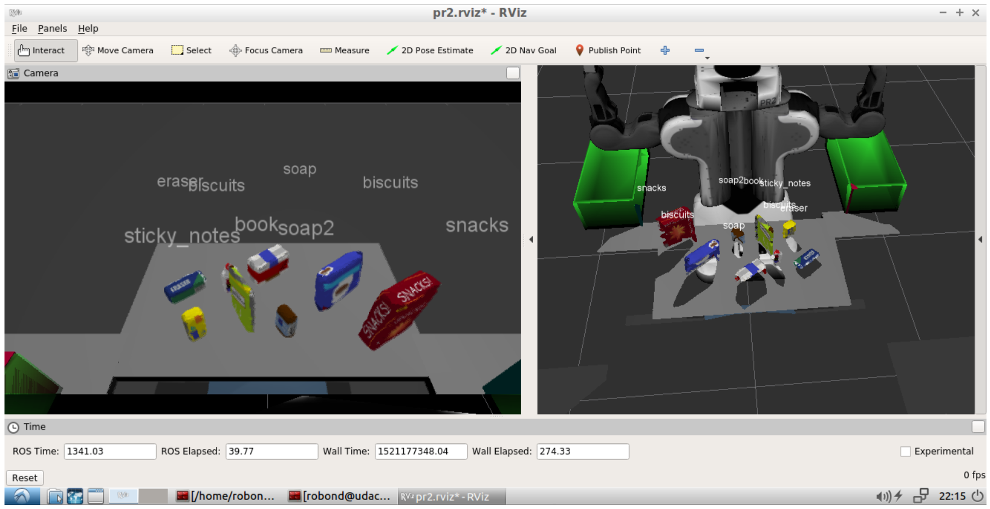
```
>>prediction = clf.predict(scaler.transform(feature_vector.reshape(1,-1)))
>>label = encoder.inverse_transform(prediction)[0]
>>detected_objects_labels.append(label)
```

To send the label information for each object to RVIZ, we have to publish the markers:

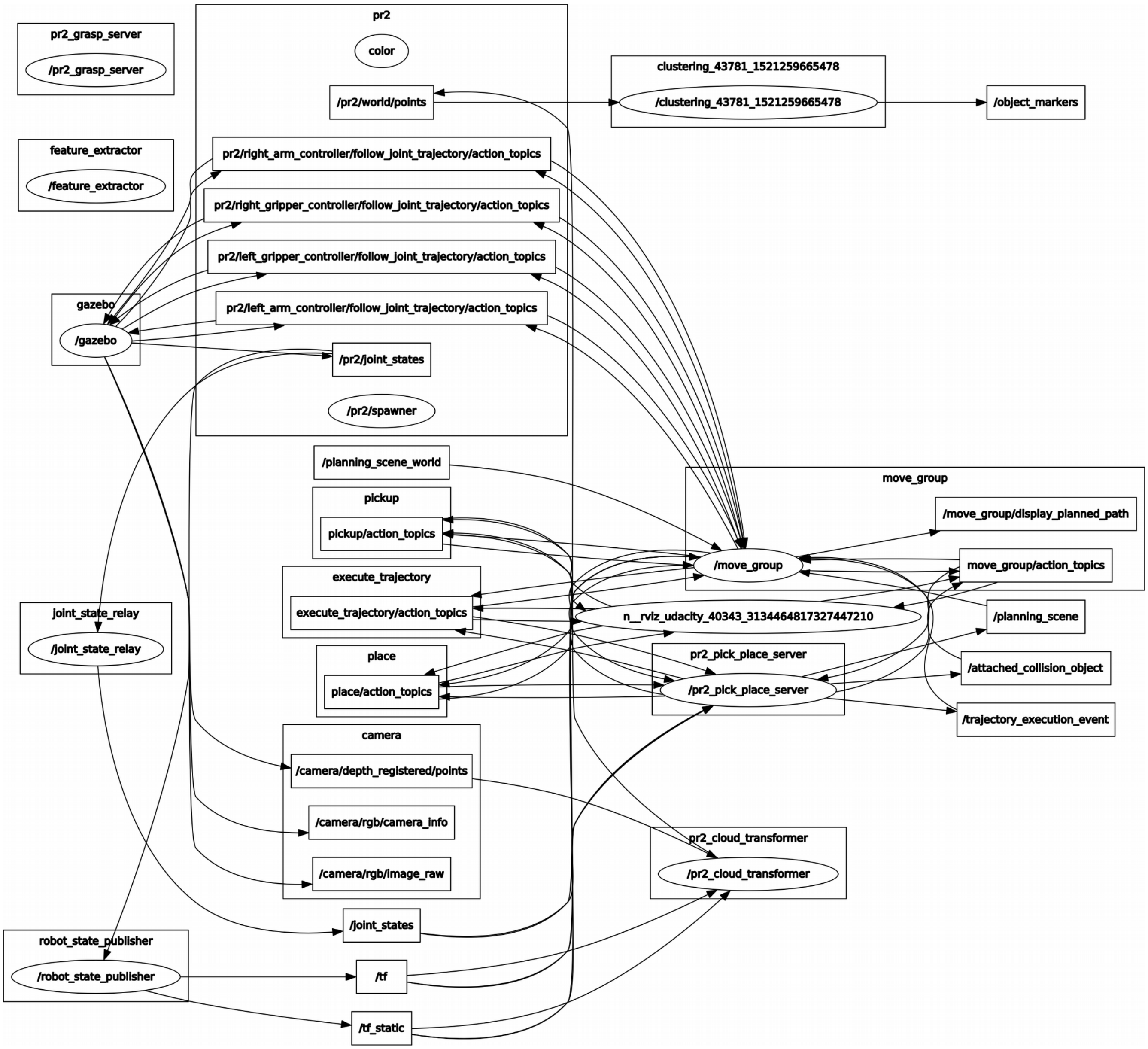
```
>>label_pos = list(white_cloud[pts_list[0]])
>>label_pos[2] += .4
>>object_markers_pub.publish(make_label(label,label_pos, index))
```

Finally, we can see the results of the prediction for every object:

Model	Result of prediction	Accuracy
1		100%
2		80% (It must be soap and it predict soap2)

3		87.5% (It must be glue and it predict biscuits)

In the following image we can see the ROS nodes and topics of the current project:



Pick and Place Setup

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.

The addition for this part is the de .yaml file. We must save information about:

- **The Object:** the object we suppose to detect and grasp
- **Object Name (Obtained from the pick list):** The label name predicted for the object
- **Arm Name (Based on the group of an object):** the arm who is suppose to pick the object
- **Group:** where the object is supposed to be delivered
- **Pick Pose (Centroid of the recognized object):** where the centroid of the object is
- **Place pose:** where the destinity box is

```
class PickObject:
    def __init__(self, object):
        self.name = String()
        self.arm = String()
        self.pick_pose = Pose()
        self.place_pose =  Pose()
        self.name.data = str(object.label)
        self.group = None
        self.yaml_dictonary = None
        points = ros_to_pcl(object.cloud).to_array()
        x, y, z = np.mean(points, axis = 0)[:3]
        self.pick_pose.position.x = np.asscalar(x)
        self.pick_pose.position.y = np.asscalar(y)
        self.pick_pose.position.z = np.asscalar(z)

    def place(self, pick_list, dropbox_list):
        for obj in pick_list:
            if obj['name'] == self.name.data:
                self.group = obj['group']
                break
        for box in dropbox_list:
            if box['group'] == self.group:
                self.arm.data = box['name']
                x, y, z = box['position']
                self.place_pose.position.x = np.float(x)
                self.place_pose.position.y = np.float(y)
                self.place_pose.position.z = np.float(z)
                break

    def Make_yaml_dict(self, scene):
        self.yaml_dictonary = make_yaml_dict(scene, self.arm, self.name, self.pick_pose, self.place_pose)
```

An array of point cloud data corresponding to the object can be obtained and computing a mean of those points will give us the centroid of the object. The code first sets the object name based on the data and converts it into the datatype that ROS understand. The code also check wheter this group was red or geen, and set the correct name (left or right).

Now, we just need to iterate over each object in the list to extract the information and publish in the .yaml file using the helper function provided.

```python
def pr2_mover(object_list):
    test_scene = Int32()
    test_scene.data = 2
    file = []
    # Get information from the YAML files
    pick_list = rospy.get_param('/object_list')
    dropbox_list = rospy.get_param('/dropbox')
    for Object in object_list:
        pickObject = PickObject(Object)
        pickObject.place(pick_list, dropbox_list)
        pickObject.Make_yaml_dict(test_scene)
        file.append(pickObject.yaml_dictonary)
        #print(pickObject.yaml_dictonary)
        rospy.wait_for_service('pick_place_routine')
    rospy.wait_for_service('pick_place_routine')
    send_to_yaml("output_model_" + str(2) + '.yaml', file)
```

Results:

Output1 .yaml	Output 2.yaml	Output 3.yaml
object_list: - arm_name: right object_name: biscuits pick_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.5428752899169922 y: -0.2430519312620163 z: 0.7067028880119324 place_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.0 y: -0.71 z: 0.605 test_scene_num: 1 - arm_name: '' object_name: soap pick_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.5439981818199158 y: -0.018749024718999863 z: 0.675325334072113 place_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0	object_list: - arm_name: right object_name: biscuits pick_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.5718656778335571 y: -0.24968938529491425 z: 0.7052477598190308 place_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.0 y: -0.71 z: 0.605 test_scene_num: 2 - arm_name: '' object_name: book pick_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.5804760456085205 y: 0.27917298674583435 z: 0.7211214900016785 place_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.0 y: 0.0 z: 0.0 test_scene_num: 2	object_list: - arm_name: '' object_name: biscuits pick_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.5903322100639343 y: -0.22067716717720032 z: 0.7038360834121704 place_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.0 y: 0.0 z: 0.0 test_scene_num: 3 - arm_name: '' object_name: snacks pick_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.44379922747612 y: -0.3513682782649994 z: 0.759294331073761 place_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.0 y: 0.0 z: 0.0 test_scene_num: 3 - arm_name: '' object_name: book pick_pose: orientation: w: 0.0 x: 0.0 y: 0.0

```
    position:
      x: 0.0
      y: 0.0
      z: 0.0
  test_scene_num: 1
- arm_name: ''
  object_name: soap2
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.4468686878681183
      y: 0.22259899973869324
      z: 0.6789992451667786
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.0
      y: 0.0
      z: 0.0
  test_scene_num: 1
- arm_name: ''
  object_name: soap2
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.5597199201583862
      y: 0.00433421041816473
      z: 0.6735892295837402
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.0
      y: 0.0
      z: 0.0
  test_scene_num: 2
- arm_name: ''
  object_name: soap2
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.4474266469478607
      y: 0.225227952003479
      z: 0.6758773922920227
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.0
      y: 0.0
      z: 0.0
  test_scene_num: 2
- arm_name: ''
  object_name: glue
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.632434606552124
      y: 0.13113407790660858
      z: 0.6805852651596069
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.0
      y: 0.0
      z: 0.0
  test_scene_num: 2
      z: 0.0
    position:
      x: 0.4932829439640045
      y: 0.08435969799757004
      z: 0.7269715666770935
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.0
      y: 0.0
      z: 0.0
  test_scene_num: 3
- arm_name: ''
  object_name: soap
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.6793539524078369
      y: 0.005475242622196674
      z: 0.674744188785553
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.0
      y: 0.0
      z: 0.0
  test_scene_num: 3
- arm_name: ''
  object_name: eraser
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.6079463958740234
      y: 0.2829769551753998
      z: 0.6464066505432129
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.0
      y: 0.0
      z: 0.0
  test_scene_num: 3
- arm_name: ''
  object_name: soap2
  pick_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.45678168535232544
      y: -0.04383600875735283
      z: 0.6754958033561707
  place_pose:
    orientation:
      w: 0.0
      x: 0.0
      y: 0.0
      z: 0.0
    position:
      x: 0.0
      y: 0.0
```

		z: 0.0 test_scene_num: 3 - arm_name: left object_name: sticky_notes pick_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.4403557777404785 y: 0.2166387140750885 z: 0.6849715113639832 place_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.0 y: 0.71 z: 0.605 test_scene_num: 3 - arm_name: '' object_name: biscuits pick_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.6120038032531738 y: 0.14238470792770386 z: 0.684952437877655 place_pose: orientation: w: 0.0 x: 0.0 y: 0.0 z: 0.0 position: x: 0.0 y: 0.0 z: 0.0 test_scene_num: 3

Conclusion

The classifier perform well besides i took just 150 poses to train de classifier. Following the confusion matrix results, for each environment i got really good results. For that, i think the errors of missclassification must be some marginal error due to the noise. I completed the project covering all the minimum requirements satisfactoy.

Future improvements include a better trainning of the classifier, make a deep analysis of the initial cloud point data to apply the pre processing more efficiently.

I really enjoy this project about perception, even more, take a deep breath inside of ROS. I really like to see how the nodes are connected to get a better understanding of the project workflow. That was possible using the command "rosrun rqt_graph rqt_grasp" who show a "map" of the entire connections.

For the submission, thecode succeed in recognizing:

- 100% (3/3) objects in test1.world
- 80% (4/5) objects in test2.world
- 87.5% (7/8) objects in test3.world

And finally the .yaml files were created satisfactory.