Project: Perception Pick & Place

Brett Gleason

August 27, 2017

The rubric for this project can be found at the following URL:
https://review.udacity.com/#!/rubrics/1067/view
I will consider the rubric points individually and describe how I addressed each point in my implementation.

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! You're done!

—

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

You're reading it!

Exercise 1, 2 and 3 pipeline implemented

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

The objective of exercise 1 was to take a simulated scene with several objects on a table and create two separate point cloud files: the first containing the table surface and the second containing the objects on the table. To accomplish this, several filters were used as well as the random sample consensus (or RANSAC) algorithm.

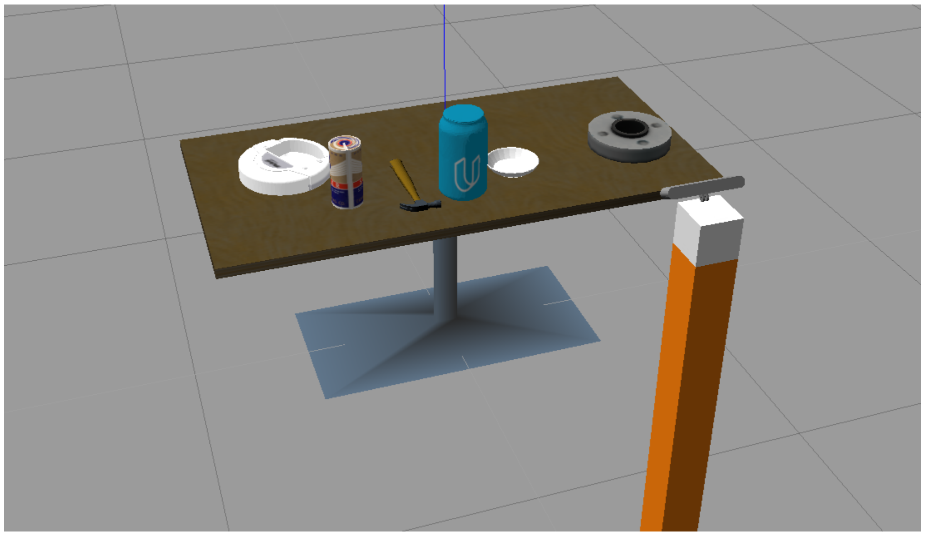


Figure 1: Simulated environment in Gazebo with camera on sensor stick, objects on table

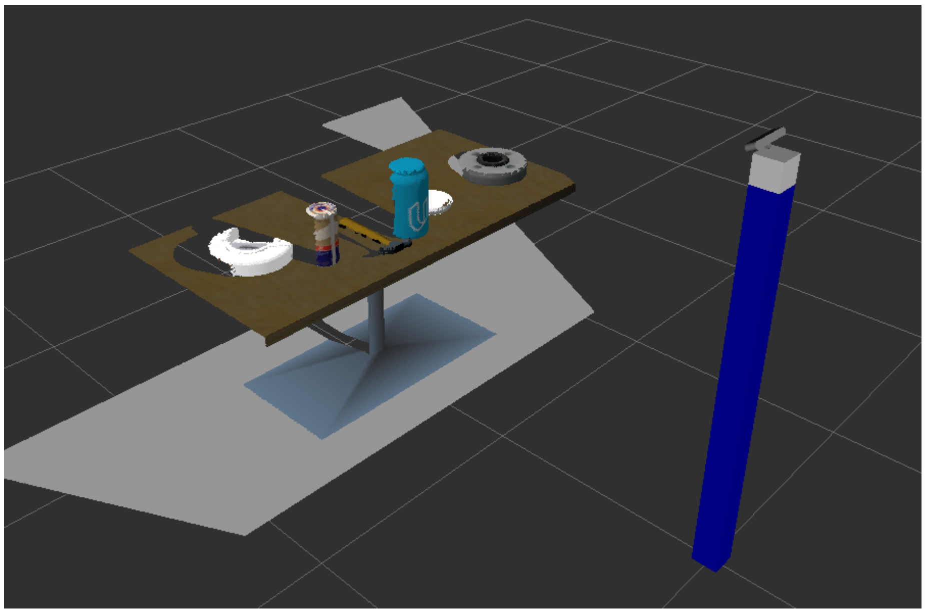


Figure 2: Simulated environment in RViz with camera on sensor stick, objects on table

Voxel Grid Downsampling

The starting point for this exercise is a point cloud file from a simulated RGB-D camera. The point cloud from the RGB-D camera is very dense, meaning any operations on it will be very computationally intense.

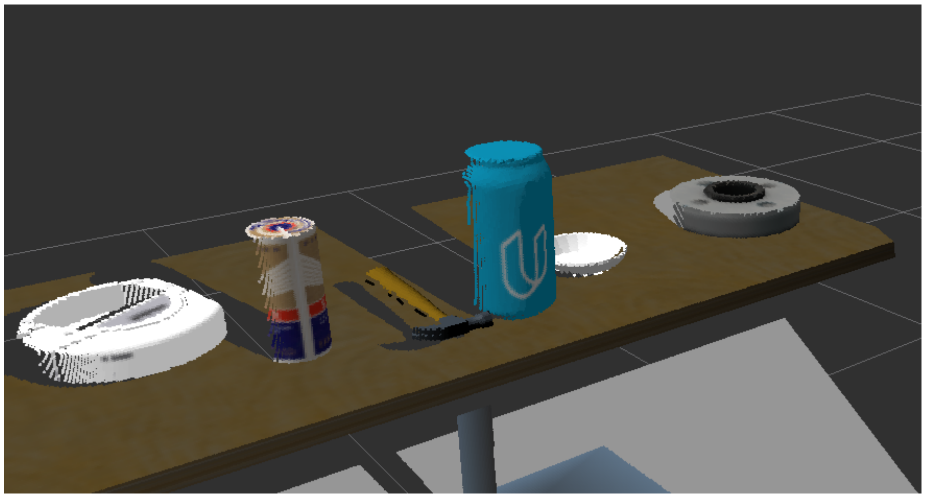


Figure 3: Point cloud file from RGB-D camera as shown in Rviz

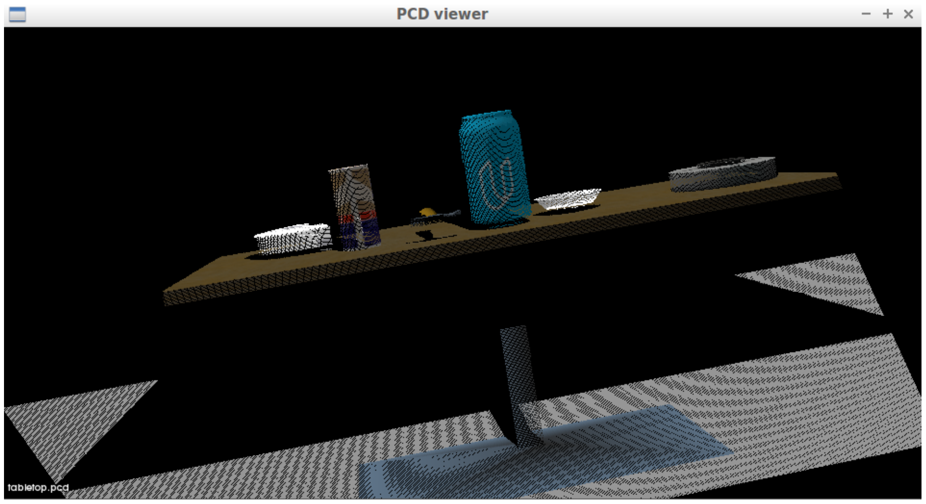


Figure 4: Point cloud file from RGB-D camera as shown in PCL viewer

Voxel grid downsampling reduces the density of the point cloud, allowing for faster computations without any loss in the ability for the point cloud data to be used for object recognition later in the perception pipeline. A voxel can be thought of as a three dimensional analog to a pixel. The input point cloud can be divided into a voxel grid by dividing it into cubic regions. To perform the voxel grid downsampling, the points within each cubic region (voxel) are replaced with a single point that represents the average of the points contained within the voxel. By choosing the size of the voxel appropriately, the down-sampled point cloud will provide a good approximation of the original point cloud while allowing faster computation in later steps. For this project, voxel grid downsampling is accomplished using the python PCL library (Point Cloud Library). A VoxelGrid filter object is created from the input point cloud and a filter() function can be run on this object after the size of the voxel is set.

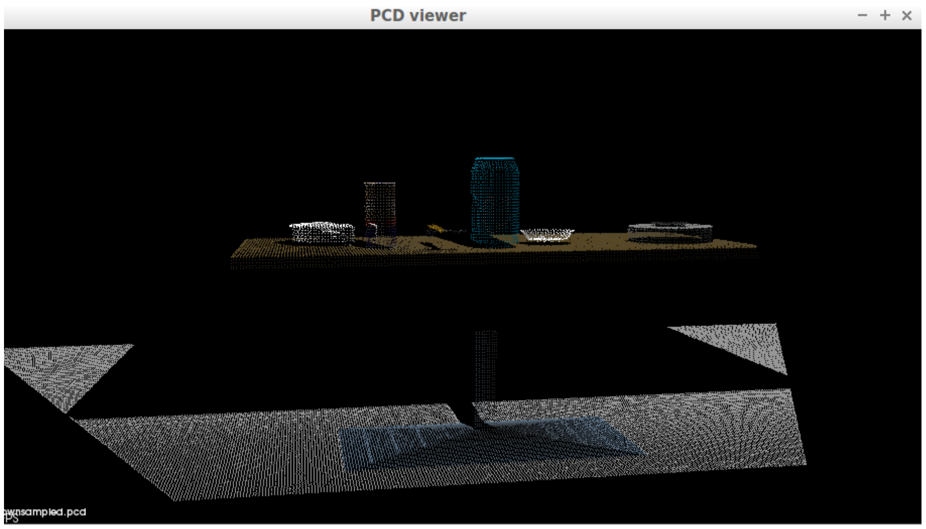


Figure 5: Point cloud file after voxel grid downsampling as shown in PCL viewer

Pass Through Filtering

Because the robot and the table are both stationary for this project it is easy to filter out parts of the point cloud that are not relevant. For this exercise a region on the Z-axis was selected such that only the table and the objects on it would be able to ”pass through” the filter as shown in figure 6.

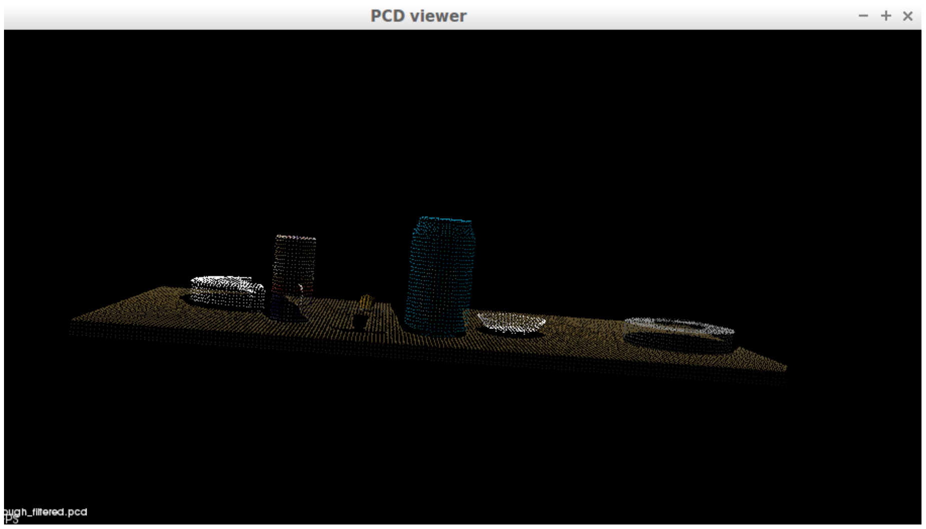


Figure 6: Point cloud file after pass through filtering as shown in PCL viewer

RANSAC

Finally, random sample consensus (or RANSAC) is an algorithm that can be used to remove the table from the point cloud, leaving only the objects of interest behind. RANSAC determines whether each point in the cloud belongs to a particular model. If the point fits the model it is classified as an inlier, if it doesn't fit it is an outlier. For this exercise a plane was used as the model, resulting in the points belonging to the table being classified as inliers to the plane, while the points belonging to the objects on the table were classified as outliers. At this point the objective of the exercise is accomplished, the input point cloud from the RGB-D camera has been separated into a point cloud containing the table surface and a point cloud containing the objects of interest.

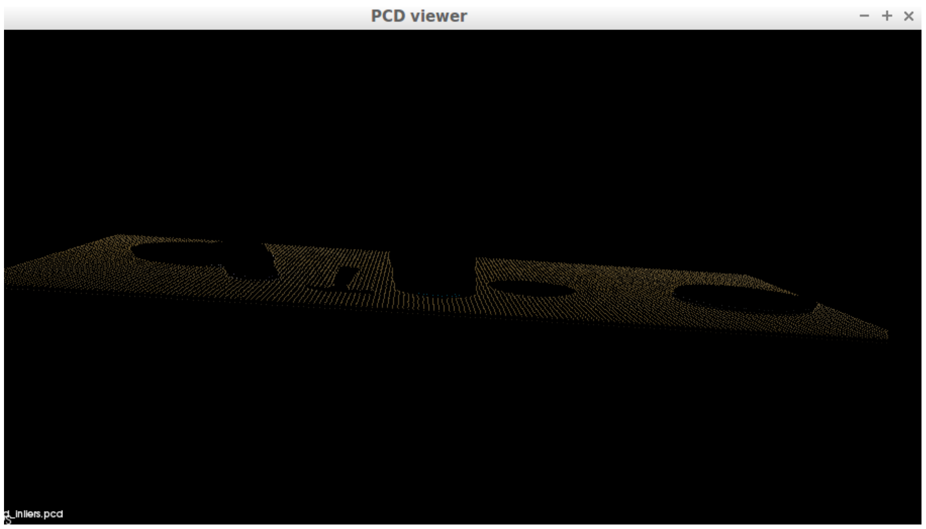


Figure 7: RANSAC inliers as shown in PCL viewer

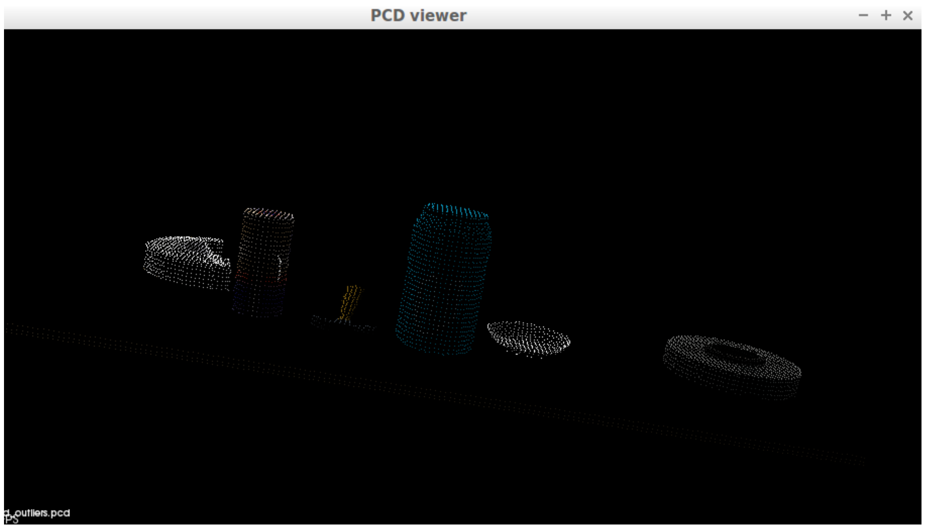


Figure 8: RANSAC outliers as shown in PCL viewer

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

For the second exercise the objective was to take the filtered point cloud containing the objects as shown in figure 8 and use Euclidean clustering to segment the points belonging to each individual object. In this exercise an additional pass through filter along the Y-axis was added so that the edge of the table would be removed from the scene.

Euclidean Clustering

The Euclidean clustering algorithm requires that a k-dimensional tree (or k-d tree) be constructed from the input point cloud before the algorithm is run. A k-d tree is a type of binary search tree that allows a nearest neighbor search to be done efficiently. K-d trees are not the only data structure that can be used to efficiently perform a nearest neighbor seach and accomplish point cloud segmentation, but PCL's implementation of Euclidean Clustering requires a k-d tree.

Euclidean clustering segments the points in the point cloud into clusters based on several parameters:

- Cluster tolerance
- Minimum cluster size
- Maximum cluster size

The cluster tolerance is the maximum distance that a point can have to its nearest neighbor and remain part of the cluster. Cluster tolerance needs to be set large enough that the points within each object are considered part of the cluster but small enough that the distance between two objects is not within the cluster tolerance, causing the two objects to become part of the same cluster. Minimum cluster size is the minimum number of points needed to constitute a cluster. Minimum cluster size needs to be set large enough that noise in the data is not considered a cluster, but small enough that an actual cluster is not disqualified. Maximum cluster size is the maximum number of points that can constitute a cluster, it must be set large enough that none of the objects in the point cloud are disqualified. Once the clustering parameters have been set appropriately, the Euclidean clustering algorithm can be applied to the point cloud. Figure 9 shows the point cloud pre Euclidean clustering, while figure 10 shows the point cloud post Euclidean clustering. The clusters are colorized during the clustering step for visual clarity.

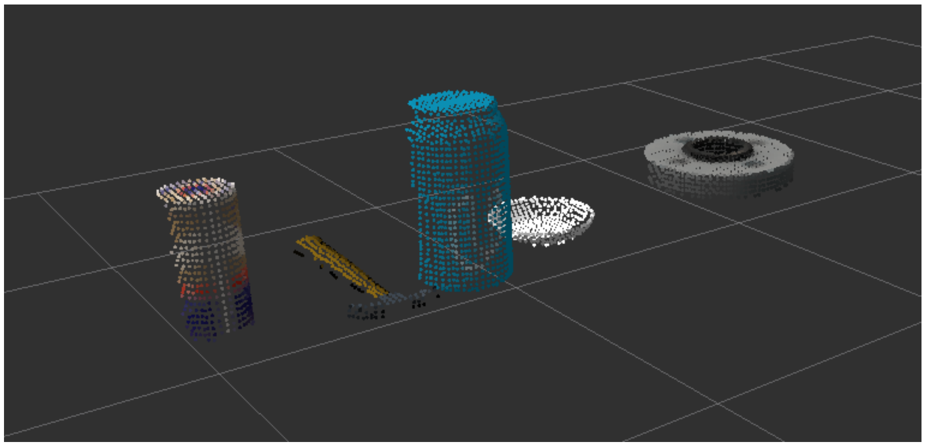


Figure 9: RANSAC outliers pre Euclidean Clustering, as shown in RViz

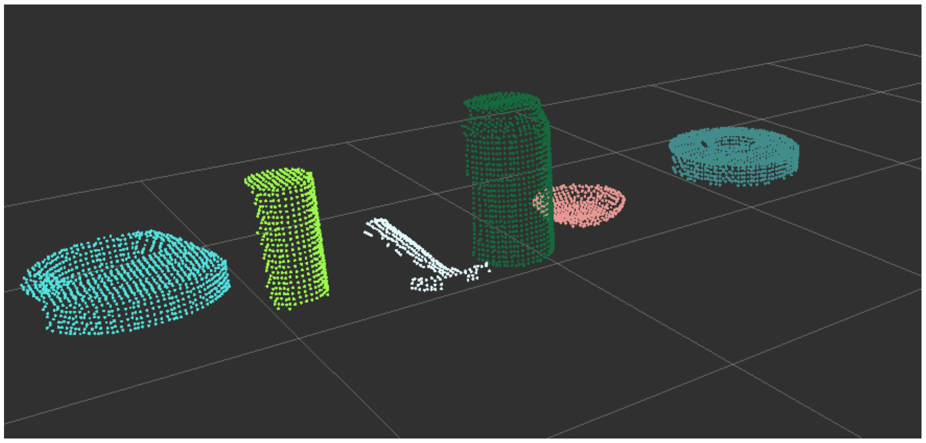


Figure 10: Point cloud after Euclidean clustering, as shown in RViz

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

In exercise-1 and 2 the input point cloud from the RGB-D camera was filtered and segmented, leaving only clusters of points belonging to the objects of interest. The goal of exercise 3 was to train a support vector machine to recognize the objects in the filtered and segmented point cloud.

SVM

A support vector machine (SVM) is a machine learning algorithm used to classify objects. The SVM is trained on a data set containing multiple objects, where each object consists of a feature vector and a label. The SVM maps the feature vectors and the labels and uses the training data to draw boundaries between the different objects in the dataset. Once these boundaries are drawn, a feature vector can be input to the model and it will be characterized and given a label based on which boundaries it falls between.

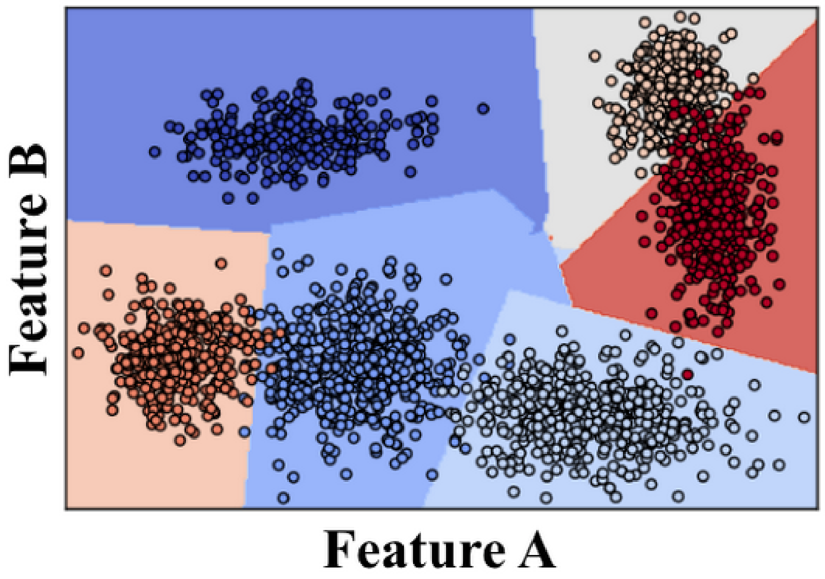


Figure 11: Example SVM data set

Figure **??** shows an example SVM data set based on two feature vectors, feature A and feature B. The feature vectors are grouped into clusters and then boundaries are drawn between the clusters to separate them into classes. An additional object given to the SVM could now be classified based on where it falls within this dataset.

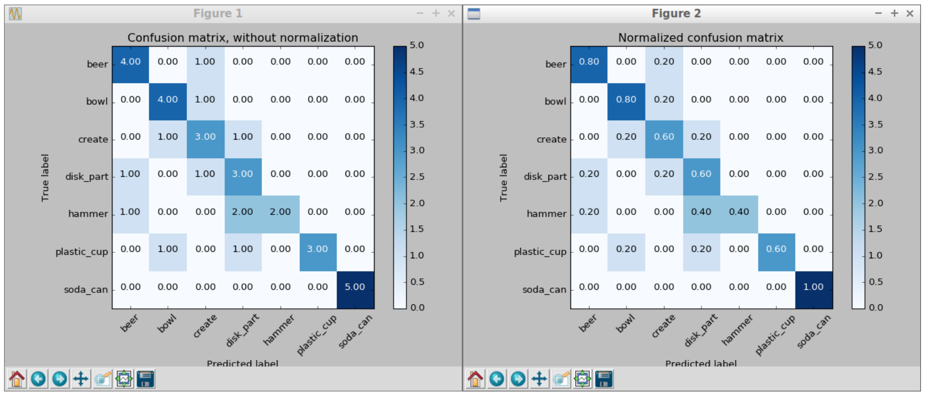
Surface Normals and Color

The feature vectors used for this exercise are the color and the surface normal vectors. For the color vector, 3 color channels are turned into histograms and concatenated, then normalized. Initially the 3 channels were RGB, but they were changed to HSV later to improve the model. The surface normal vector was calculated similarly to the color vector, except that instead of 3 color channels to make the histograms the X, Y, and Z components of the surface normal vectors were used. After the surface normal histograms are concatenated and normalized, the color and surface normal histograms can be concatenated to create the complete feature vector.

SVM Training

The SVM was trained by placing each object to be classified in a random pose for the camera. The features were extracted and used to assemble the training data set. Initially each object was placed in 5 random poses. Once the features had been generated, the python Scikit-learn package was used to train the SVM.

To represent the accuracy of the SVM, a confusion matrix was created. The Y axis represents the actual instances of the object while the X axis represents the SVM's classified instances of the object. On the left is a confusion matrix with the raw counts, on the right is a confusion matrix with the counts normalized from 0-1. Figure **??** shows the confusion matrix for the first SVM.



Confusion matrix, without normalization

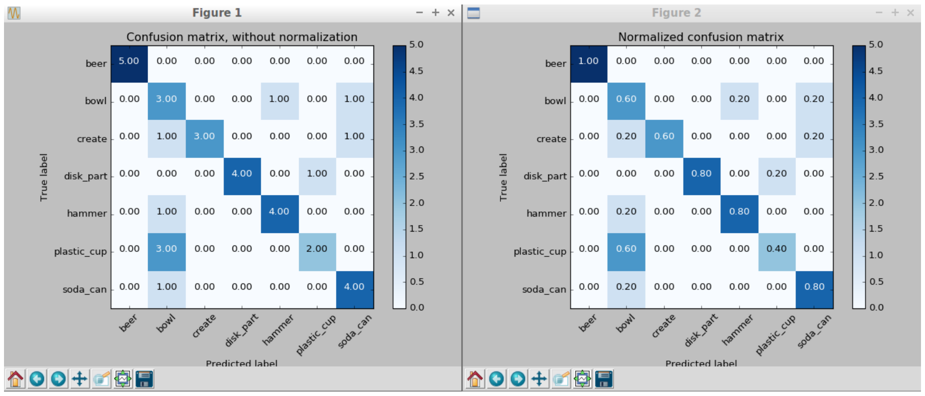
	beer	bowl	create	disk_part	hammer	plastic_cup	soda_can
beer	4.00	0.00	1.00	0.00	0.00	0.00	0.00
bowl	0.00	4.00	1.00	0.00	0.00	0.00	0.00
create	0.00	1.00	3.00	1.00	0.00	0.00	0.00
disk_part	1.00	0.00	1.00	3.00	0.00	0.00	0.00
hammer	1.00	0.00	0.00	2.00	2.00	0.00	0.00
plastic_cup	0.00	1.00	0.00	1.00	0.00	3.00	0.00
soda_can	0.00	0.00	0.00	0.00	0.00	0.00	5.00

Normalized confusion matrix

	beer	bowl	create	disk_part	hammer	plastic_cup	soda_can
beer	0.80	0.00	0.20	0.00	0.00	0.00	0.00
bowl	0.00	0.80	0.20	0.00	0.00	0.00	0.00
create	0.00	0.20	0.60	0.20	0.00	0.00	0.00
disk_part	0.20	0.00	0.20	0.60	0.00	0.00	0.00
hammer	0.20	0.00	0.00	0.40	0.40	0.00	0.00
plastic_cup	0.00	0.20	0.00	0.20	0.00	0.60	0.00
soda_can	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Figure 12: Confusion matrix for the first SVM

The initial result after training the SVM was not good enough to correctly identify the objects. The feature vector was changed to use HSV color space instead of RGB in an attempt to increase the accuracy of the SVM. Figure **??** shows the results after generating features and re-training the SVM.



Confusion matrix, without normalization

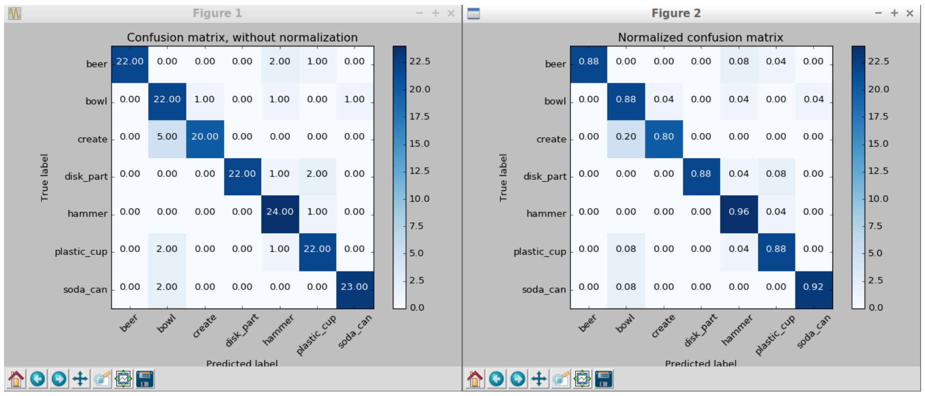
	beer	bowl	create	disk_part	hammer	plastic_cup	soda_can
beer	5.00	0.00	0.00	0.00	0.00	0.00	0.00
bowl	0.00	3.00	0.00	0.00	1.00	0.00	1.00
create	0.00	1.00	3.00	0.00	0.00	0.00	1.00
disk_part	0.00	0.00	0.00	4.00	0.00	1.00	0.00
hammer	0.00	1.00	0.00	0.00	4.00	0.00	0.00
plastic_cup	0.00	3.00	0.00	0.00	0.00	2.00	0.00
soda_can	0.00	1.00	0.00	0.00	0.00	0.00	4.00

Normalized confusion matrix

	beer	bowl	create	disk_part	hammer	plastic_cup	soda_can
beer	1.00	0.00	0.00	0.00	0.00	0.00	0.00
bowl	0.00	0.60	0.00	0.00	0.20	0.00	0.20
create	0.00	0.20	0.60	0.00	0.00	0.00	0.20
disk_part	0.00	0.00	0.00	0.80	0.00	0.20	0.00
hammer	0.00	0.20	0.00	0.00	0.80	0.00	0.00
plastic_cup	0.00	0.60	0.00	0.00	0.00	0.40	0.00
soda_can	0.00	0.20	0.00	0.00	0.00	0.00	0.80

Figure 13: Confusion matrix using HSV color space instead of RGB

After converting from RGB to HSV color vectors the SVM was still not accurate enough for object identification. The next step was to increase the number of poses for each object from 5 to 25 for the feature generation. Figure **??** shows the results of training the SVM with the additional feature data.



Confusion matrix, without normalization

	beer	bowl	create	disk_part	hammer	plastic_cup	soda_can
beer	22.00	0.00	0.00	0.00	2.00	1.00	0.00
bowl	0.00	22.00	1.00	0.00	1.00	0.00	1.00
create	0.00	5.00	20.00	0.00	0.00	0.00	0.00
disk_part	0.00	0.00	0.00	22.00	1.00	2.00	0.00
hammer	0.00	0.00	0.00	0.00	24.00	1.00	0.00
plastic_cup	0.00	2.00	0.00	0.00	1.00	22.00	0.00
soda_can	0.00	2.00	0.00	0.00	0.00	0.00	23.00

Normalized confusion matrix

	beer	bowl	create	disk_part	hammer	plastic_cup	soda_can
beer	0.88	0.00	0.00	0.00	0.08	0.04	0.00
bowl	0.00	0.88	0.04	0.00	0.04	0.00	0.04
create	0.00	0.20	0.80	0.00	0.00	0.00	0.00
disk_part	0.00	0.00	0.00	0.88	0.04	0.08	0.00
hammer	0.00	0.00	0.00	0.00	0.96	0.04	0.00
plastic_cup	0.00	0.08	0.00	0.00	0.04	0.88	0.00
soda_can	0.00	0.08	0.00	0.00	0.00	0.00	0.92

Figure 14: Confusion matrix after using 25 poses for each object for feature generation

After this adjustment the SVM was accurate enough to correctly identify the objects on the table and exercise 3 was complete.

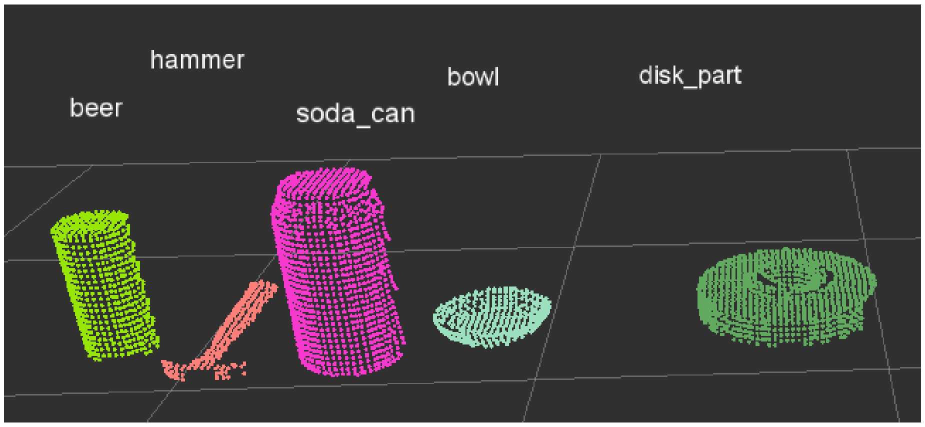


Figure 15: Point cloud after clustering with object recognition applied. Shown in RViz.

Pick and Place Setup

1. 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.