

Project 3: 3D Perception

3D Perception with machine learning

Christopher J. Rukas | Robotic Software Engineer | 10/20/2018

# 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)**

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

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.

**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.

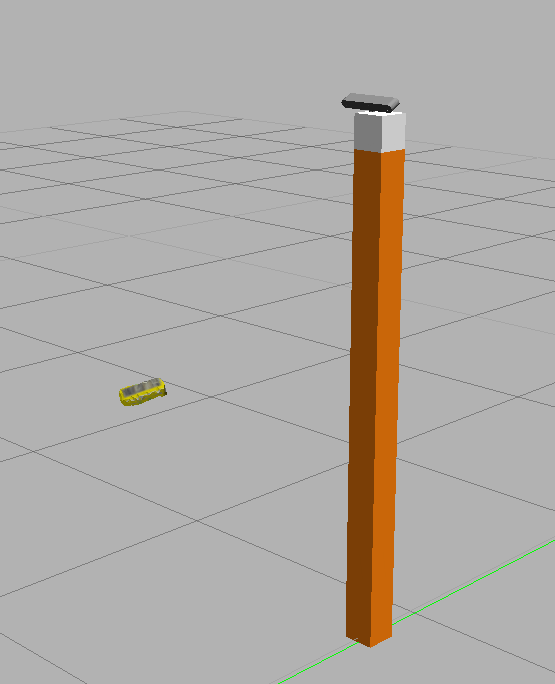


Figure 1: Random orientation of objects for data generation

### Accuracy

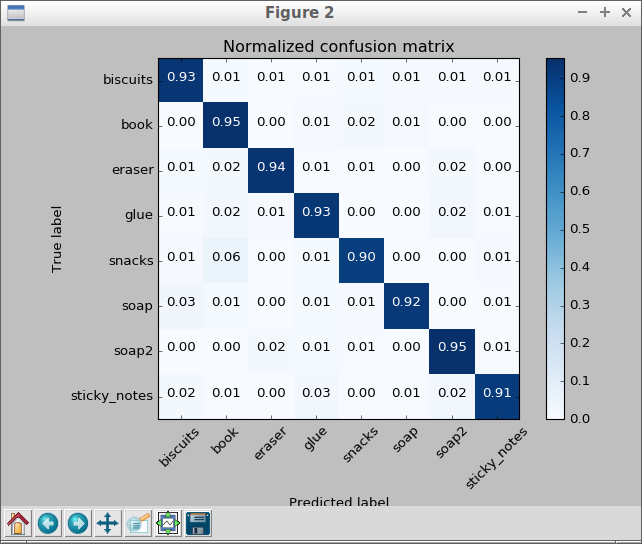
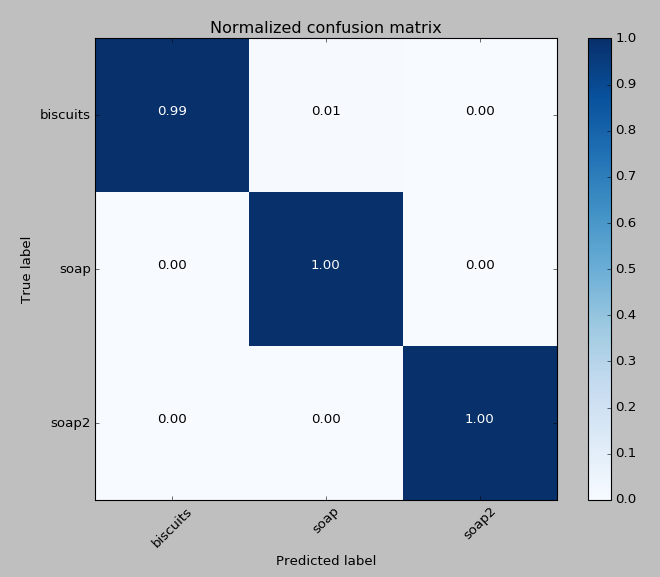
Increasing the number of samples for training improves the capability of classifier across the features of HSV and normal. Increasing the sample count from 200 to 400 doesn’t seem to make a big difference. 

Figure 2: Confusion Matrix with HSV and Normals as features

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



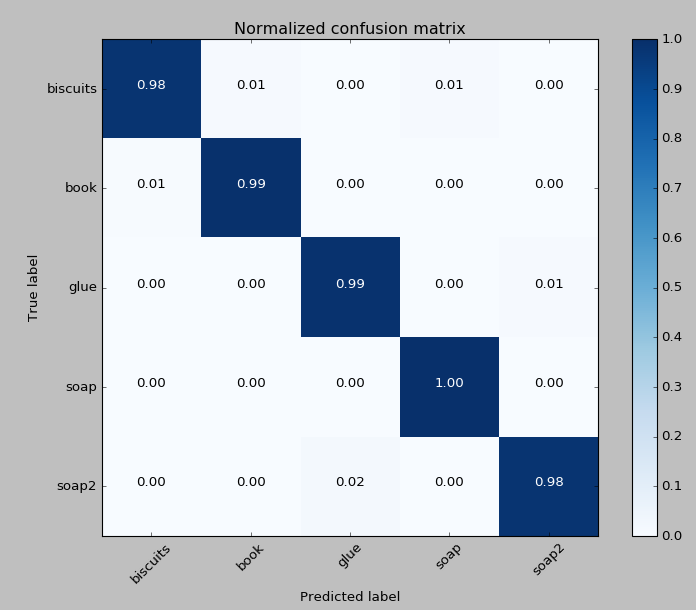


Figure : Training on Set 2 : 500 cycles

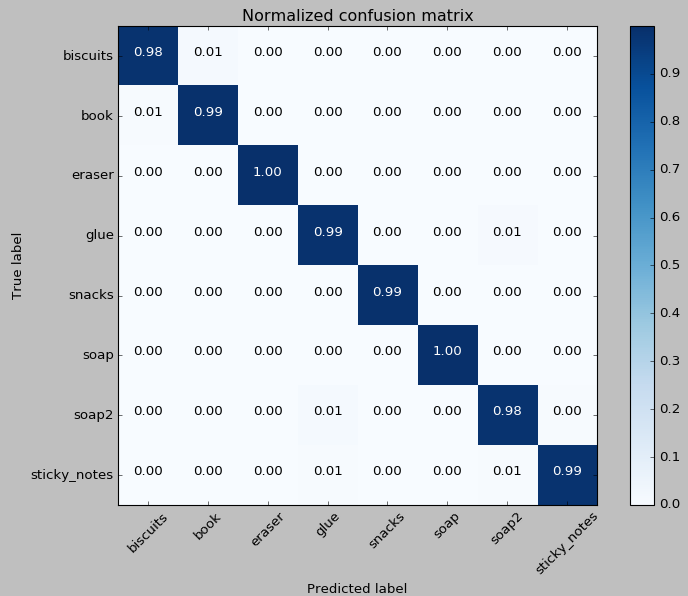


Figure : World 3 Trained on 1000 Cycles

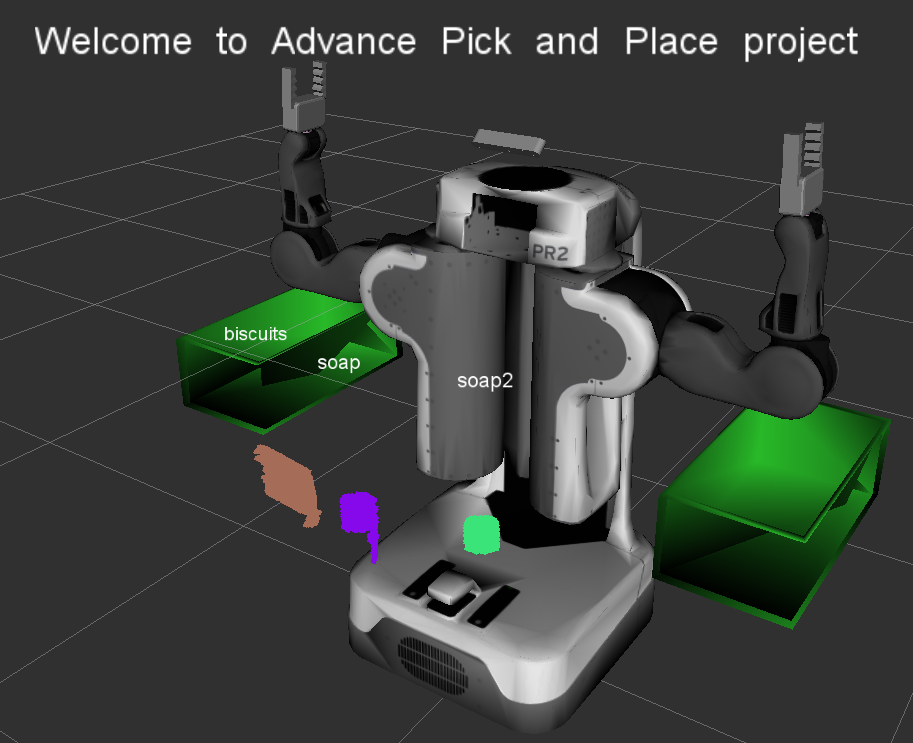
# Project Performance

## Launch

roslaunch pr2\_robot pick\_place\_project.launch

rosrun pr2\_robot project\_template.py

## Pick List 1



| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| 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. The project repository contains a template writeup for this project that you can use as a guide and a starting point. | The writeup / README should include a statement and supporting figures / images that explain how each rubric item was addressed, and specifically where in the code each step was handled. The writeup should include a discussion of what worked, what didn't and how the project implementation could be improved going forward. |

Exercise 1, 2 and 3 Pipeline Implemented

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| Complete Exercise 1 steps. Pipeline for filtering and RANSAC plane fitting implemented. | The pcl\_callback() function within the template Python script has been filled out to include filtering and RANSAC plane fitting. Not required, but to help your reviewer consider adding screenshots of output at different steps in your writeup with brief explanations. |
| Complete Exercise 2 steps: Pipeline including clustering for segmentation implemented. | Steps for cluster segmentation have been added to the pcl\_callback() function in the template Python script. Not required, but to help your reviewer consider adding screenshots of output at different steps in your writeup with brief explanations. |
| Complete Exercise 3 Steps. Features extracted and SVM trained. Object recognition implemented. | Both compute\_color\_histograms() and compute\_normal\_histograms() functions have been filled out and SVM has been trained using train\_svm.py. Please provide a snapshot of your normalized confusion matrix (output from train\_svm.py in your writeup / README. Object recognition steps have been implemented in the pcl\_callback() function within template Python script. Not required, but to help your reviewer consider adding screenshots of output at different steps in your writeup with brief explanations. |

Pick and Place Setup

| CRITERIA | MEETS SPECIFICATIONS |
| --- | --- |
| 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. | You can add this functionality to your already existing ros node or create a new node that communicates with your perception pipeline to perform sequential object recognition. Save your PickPlace requests into output\_1.yaml, output\_2.yaml, and output\_3.yaml for each scene respectively. Add screenshots in your writeup of output showing label markers in RViz to demonstrate your object recognition success rate in each of the three scenarios. **Note: for a passing submission, your pipeline must correctly identify 100% of objects in test1.world, 80% (4/5) in test2.world and 75% (6/8) in test3.world.** |