Udacity: 3D Perception Report

Shane Reynolds

May 3, 2018

Contents

1 Introduction & Background

In order for a robot to perform meaningful actions it requires some capacity to perceive its environment - the devices used to capture data about an environment are called sensors. There are two main categories into which sensors fall into: active, and passive. The principal distinction between these two types of sensors are that passive sensors measure energy which is already present in the environment, whilst active sensors emit some form of energy and measure the reactions of this energy with the environment. Table 1 (over the page) shows some examples of the more common sensors which are used for robotic perception. Generally, robotic perception systems are comprised of both active and passive sensors. In fact, both active and passive sensors are often combined into a single sensor to create a hybrid sensor. An example of a widely used hybrid sensor which features in many households is the Microsoft Kinect, shown in Figure 1.

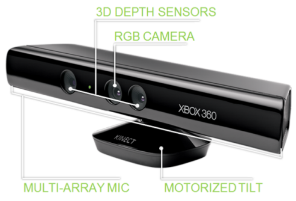


Figure 1: The Microsoft Kinect is an example of a hybrid sensor called an RGBD camera. It captures 2D pixel arrays in 3 colour channels, in addition to capturing depth information using structured infra-red light pattern.

This project explores the perception of an environment using a hybrid sensor called an RGBD camera. This type of sensor captures a 2D pixel array on red, green, and blue channels using a monocular camera. Further, the sensor captures depth information by measuring the deformation of reflections from structured infra-red (IR) light emitted into the environment. The sensor will be employed using a WillowGarage PR2 robot simulated using ROS, Gazebo, and Rviz. A real world image of the PR2 can be seen in Figure 2, and a close up of the robot's sensing hardware can be seen in Figure 3.

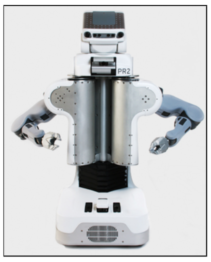


Figure 2: A picture of the WillowGarage PR2 robot.

Figure 3: A close up of the PR2's RGBD camera which captures image and depth data, and is used to create point clouds.

The captured sensor information is processed into a point cloud using the `pcl` library which is implemented in ROS. Perception of an environment is not simply the capture of point cloud data, rather, it is the implementation of software in order to make sense of the point cloud data. The ultimate goal of this project is for the PR2 to identify objects on a table, pick these objects in a specified order, and stow them into the desired container either on the left or the right of the robot. In order to complete this task, there are two main sub-tasks that the robot's perception architecture must achieve: Segmentation, and Object Recognition. Segmentation is a complex process made up of many subtasks. Briefly, the segmentation implementation for this project is made up of the following sequence of activities:

Table 1: Examples of passive and active sensors which can be used for robotic perception systems (note that this list is not exhaustive)

Passive Sensors		**Active Sensors**	
Name	**Description**	**Name**	**Description**
Monocular Camera	A single RGB camera providing information on texture and shape	LIDAR	A 3D laser scanner which determines information about an environment through reading pulsed laser emission reflections
Stereo Camera	Consists of 2 monocular cameras providing the same information as the single monocular camera, but with the addition of depth information too	Time of Flight Camera	A 3D ToF camera performs depth measurement by illuminating environment with infra-red and observing the time taken to reflect from surfaces to the camera
		Ultrasonic	Provides depth information by sending out high-frequency sound pulses and measuring time taken for sound to reverberate

1. Statistical filtering;
2. Voxel downsampling;
3. RANSAC plane filtering;
4. Passthrough filtering; and
5. Euclidean clustering (DBSCAN)

These activities are applied to the captured point cloud data, and will be explored in more detail in Section 2.1. Object recognition is a similarly complex activity, which employs the uses of machine learning. There are numerous machine learning schemes that may be implemented in this instance, to varying degrees of effectiveness and computational cost. Irrespective of which machine learning algorithm is used, features which will be important to the developed model will need to be selected, and data captured which is used to train the model. This project employs the use of a supervised learning algorithm called a support vector machine and will be explored in more detail in Section 2.2.

2 Segmentation

2.1 Statistical Filtering to Remove Image Noise

As previously mentioned, the point cloud is obtained using an RGBD camera to capture a 2D image which consists of three feature maps, and a depth representation. The three feature maps represent the individual Red, Green, and Blue (RBG) channels for colour image. Each discrete pixel in the 2D array is also assigned an image depth. There are a total of 4 dimensions for each individual point in the point cloud which is assembled in the ROS environment by the `pcl` library. The captured point cloud is not a perfect representation of the environment, rather, there are elements of noise introduced through dust, humidity, and light sources in the environment. Further, often instrumentation and transmission channels are imperfect which produce noise in the signal. The noisy signal can be seen in Figure 4 - the noise are the stochastic particles throughout the image. To remove the noise, a statistical outlier filter is used. This employs a Gaussian method to remove statistical outlier from the pointcloud. The implementation in the `pcl_callback` function can be seen in Listing 1. A signal which has had noise removed can be seen in Figure 5. The successful implementation of the object recognition relies on clean segmentation of objects in a frame. The presence of noise results in the confusion of the Euclidean clustering method which provides segmentation, so it is important to apply this statistical filter prior to any further processing.

Listing 1: Obtain the point cloud and statistically filter to remove noise

```
##############################################################################
    # Convert ROS msg to PCL data
##############################################################################
    cloud = ros_to_pcl(pcl_msg)

##############################################################################
    # Statistical Outlier Filtering
##############################################################################
    outlier_filter = cloud.make_statistical_outlier_filter()
    outlier_filter.set_mean_k(2)
    outlier_filter.set_std_dev_mul_thresh(0.5)
    cloud_filtered = outlier_filter.filter()
```

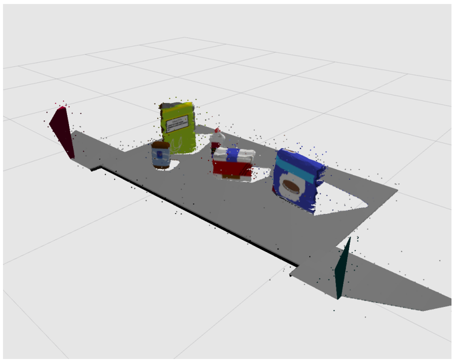


Figure 4: Noisey point cloud of table and objects

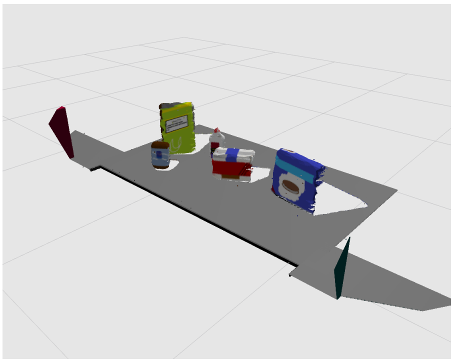


Figure 5: The application of the Gaussian filter removes noise from the signal

2.2 Voxel Downsampling

Point clouds often provide more data than necessary to achieve an accurate representation of the physical environment. The processing of this data, left unchecked, is computationally expensive. Downsampling is the process of removing data points in a systematic fashion, and is a technique that is often employed in the field of image processing. Voxel downsampling is analogous to the 2D process, that is, points in the three dimensional point cloud model are removed in a systematic fashion. Care needs to be taken when adjusting the parameters for the Voxel downsampling - if the downsampling is too aggressive, too much information may be removed compromising the ability to provide effective segmentation. The implementation of the Voxel downsampling can be seen in Listing 2. Figure 6 shows the pointcloud before Voxel downsampling, and Figure 7 shows the pointcloud after downsampling.

Listing 2: Obtain the point cloud

```
##############################################################################
    # 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()
```

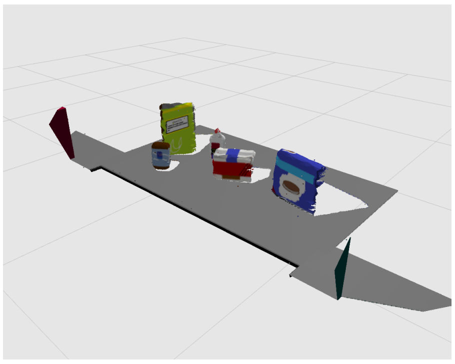


Figure 6: include a figure which shows no Voxel Downsampling

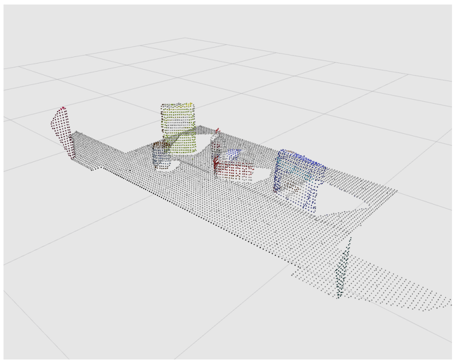


Figure 7: include a figure which shows the basics of Voxel Downsampling

2.3 Passthrough Filtering

A pass through filter is a simply designed to remove points from the point cloud which fall outside a spatially bounded region. This filter is very basic in its implementation, and does not rely on any statistical filters to achieve the results. The implementation can be seen in Listing 4. In this instance, we remove points which don't fall within an area bounded by a rectangular prism. Whilst this filtering is not essential, it helps stops the robot from segmenting points which belong to the boxes which lie to the left and right of the robot's work space. It also removes the robot's arms from the point cloud. Figure 8 shows the unfiltered pointcloud, and Figure 9 shows the pointcloud after the passthrough filter has been applied.

Listing 3: Obtain the point cloud

```
###############################################################################
    # PassThrough Filter
###############################################################################
    # Create a PassThrough filter object.
    passthrough_z = cloud_filtered.make_passthrough_filter()

    # ZAXIS Assign axis and range to the passthrough filter object.
    filter_axis = 'z'
    passthrough_z.set_filter_field_name(filter_axis)
    axis_min = 0.60
    axis_max = 1.8
    passthrough_z.set_filter_limits(axis_min, axis_max)

    # Use the filter function to obtain the filtered z-axis.
    cloud_filtered = passthrough_z.filter()

    # Create PassThrough filter object
    passthrough_y = cloud_filtered.make_passthrough_filter()

    # YAXIS Assign axis and range to the passthrough filter object
    filter_axis = 'y'
    passthrough_y.set_filter_field_name(filter_axis)
    axis_min = -0.5
    axis_max = 0.5
    passthrough_y.set_filter_limits(axis_min, axis_max)

    # Use the filter function to obtain the filtered y-axis
    cloud_filtered = passthrough_y.filter()
```

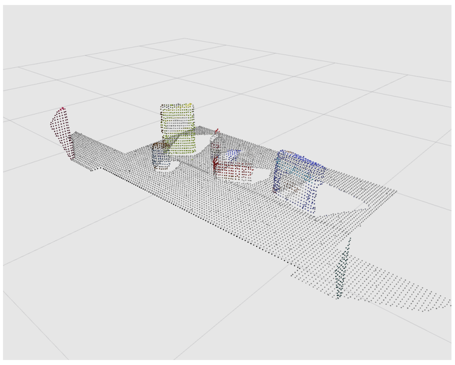


Figure 8: Include and image of the unfiltered point cloud

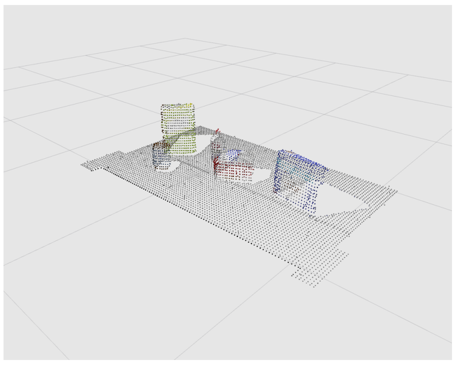


Figure 9: Include an image of the filtered point cloud

2.4 RANSAC Plane Segmentation

Random sample consensus (RANSAC) is an segmentation algorithm which detects statistical outliers which do not fit a geometrical mathematical model. The mathematical model is specified depending on the filtering requirement. In this instance the mathematical model being used is a plane, which closely resembles the table which the objects sit on in the point cloud image. The model outliers, that is those points which don't statistically fit the mathematical representation of a plane, are filtered from the point cloud. Filtered points which fit the model are stored in the variable `could_table`, and those that don't fit the model are stored in the variable `cloud_objects`. This processing helps to isolate the objects in the image, ready for segmentation and object recognition. The implementation can be seen in Listing 4. The filtered could points which fit the model can be seen in Figure 10 - the table has been extracted. The remaining points, the outliers, can be seen in Figure 11. Including the table in the pointcloud image creates a potential opportunity to confuse the Euclidean clustering algorithm since the segmentation is based on spatial proximity. Further benefits include the decreased computation cost for processing the remainder of the point cloud.

Listing 4: Obtain the point cloud

```
##############################################################################
    # 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 = 0.01
    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 and outliers
##############################################################################
    # Extract inliers
    cloud_table = cloud_filtered.extract(inliers, negative=False)

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

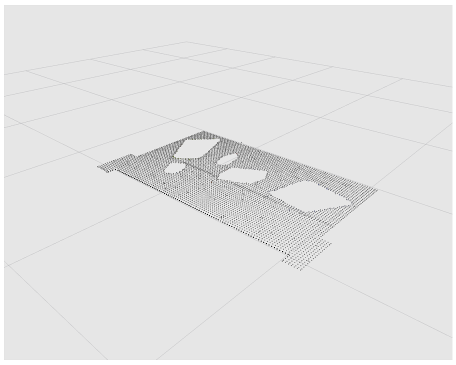


Figure 10: Include an image of filtered point cloud table

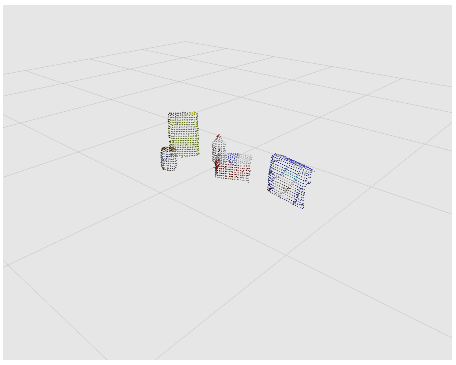


Figure 11: Include an image of the filtered point cloud which has all of the objects only

2.5 Euclidean Clustering (DBSCAN)

Segmentation is undertaken using a Euclidean clustering algorithm called DBSCAN - the implementation can be seen in Listing 5. The algorithm is a density based spatial clustering algorithm. Typically this algorithm is employed when k-means clustering is not appropriate, that is, points in the point cloud together based on their proximity to each other.
Potential problems with this method. This method encounters problems if the objects are too close to each other. If the objects are too close to each other, then the Euclidean clustering method fails - the algorithm will think that the objects

Listing 5: Apply Euclidean clustering to the pointcloud

```
###############################################################################
    # Extract inliers and outliers
###############################################################################
    # Extract inliers
    cloud_table = cloud_filtered.extract(inliers, negative=False)

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


###############################################################################
    # Euclidean Clustering
###############################################################################
    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)
    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()
```

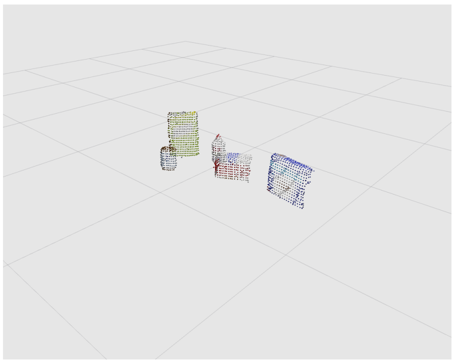


Figure 12: Include and image of the unfiltered point cloud

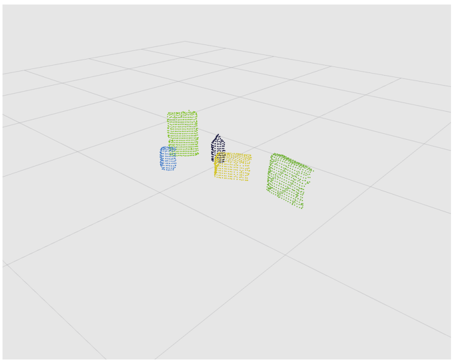


Figure 13: Include an image of the filtered point cloud

Listing 6: Assign a mask colour to each segment of the segmented point cloud

```
###############################################################################
    # Create Cluster-Mask Point Cloud to visualize each cluster separately
###############################################################################
    #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)
```

3 Object Recognition

Object recognition allows a robot to correctly locate an object of interest in the environment. The first step to achieving this is for the robot to distinguish between point cloud data which belong to objects, as opposed to pcl data belonging to the environment. This has largely been achieved using Passthrough filtering, and RANSAC plane segmentation. The next step is to identify the set of pcl data points actually belonging to an individual object - this has been achieved using Euclidean Clustering (DBSCAN). With these two things achieved, the task of identifying an object, can be simplified to identifying a set of features in the object point cloud data, which match the object of interest.

This task requires the robot to be able to distinguish between **XXXX** different objects that may be present in a scene. A typical set of objects placed in a scene is shown in Figure XXXX.

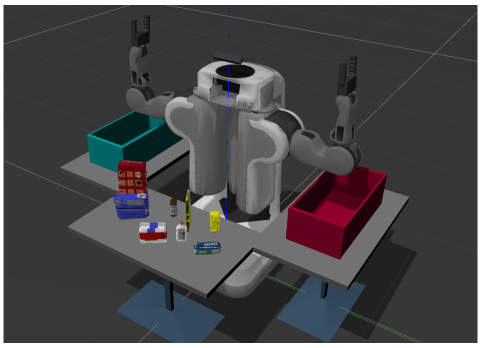


Figure 14: text

Need to talk about a support vector machine

3.1 Object Features

The object shown in Figure XXXX has two useful data sets that we can expoit for a machine learning application: colour, and object shape. Capturing colour data requires an understanding of how colour is represented digitally. There are many choices for representing colour spaces, however, normally either Red, Green, and Blue (RGB), or Hue, Saturation, and Value (HSV) are chosen for computer vision applications. The main reason for these choices are due to a well established code base for RGB and HSV. Also there are many packages providing conversion between the two, making for easy implementation. RGB and HSV colour spaces are shown in Figures XXXX and XXXX, respectively. Which colour space is best for this application? Both would suffice, however, there is an advantage of choosing HSV. The HSV colour space separates the *luma* (colour intensity), from the *chroma* (colour information). This is desirable because it makes colour data less susceptible to changes in lighting, or shadows, resulting in more robust performance from machine learning models.

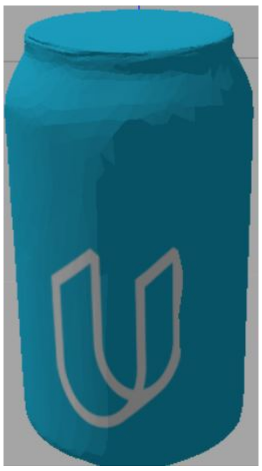


Figure 15: text

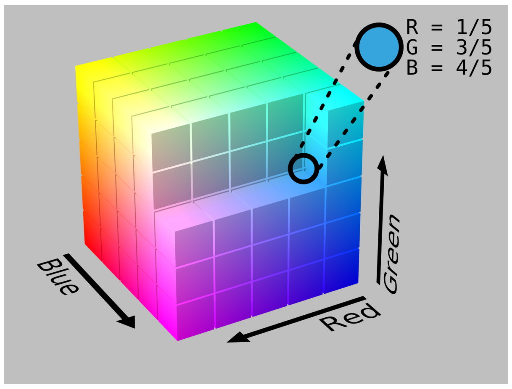


Figure 16: text

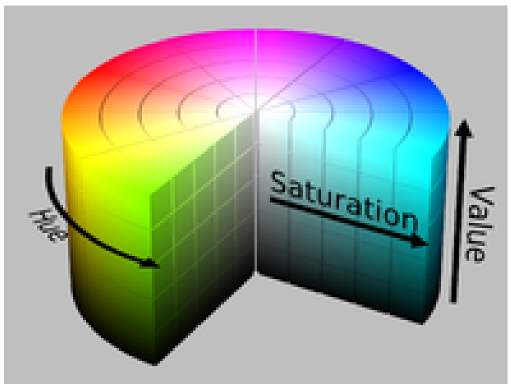


Figure 17: text

Shape based data allows for the description of the object surface. Whilst there are many mathematical descriptions for manifold surfaces which we may employ, a simple idea to capture this information uses surface normals discretely distributed over the object surface.

Figure XXXX shows a surface with a series of normal vectors located at discrete points over the surface as described. Finally, deciding on the type data to capture is only one part of the problem when considering model features. Once we have the data, how should it be represented. One effective way used in computer vision applications is to build histograms of the HSV colours and surface normals profiles to obtain an estimate of the distributio. Section 3.2 and Section 3.3 discuss the process of using histograms to create features in more detail.

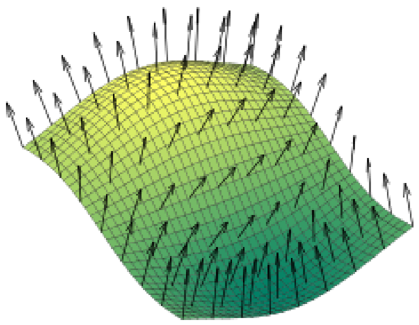


Figure 18: text

3.2 Colour Histograms

Once the RGB image data has been captured from the RGBD camera, it is converted into HSV values for each pixel in the image. A histogram is then created for the HSV data using bin ranges from 0 to 256, with approximately 70 bins. An example of a continuous HSV histogram can be seen in Figure XXXX. The function implementation used to create colour histograms for the PR2 can be seen in Listing XXXX.

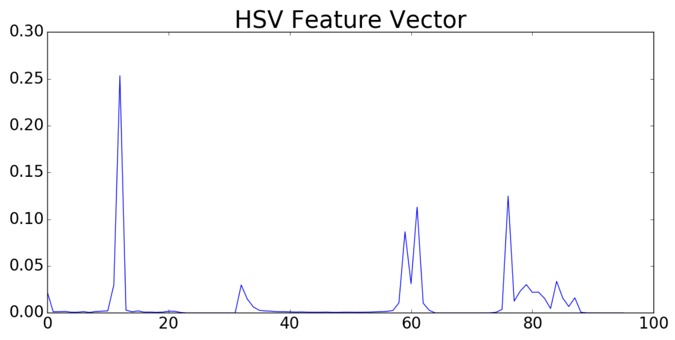


Figure 19: text

Listing 7: insert text

```
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
        h_hist = np.histogram(channel_1_vals, bins=70, range=(0, 256))
        s_hist = np.histogram(channel_2_vals, bins=70, range=(0, 256))
        v_hist = np.histogram(channel_3_vals, bins=70, range=(0, 256))

        # TODO: Concatenate and normalize the histograms
        features = np.concatenate((h_hist[0], s_hist[0], v_hist[0])).astype(np.float64)
        # Generate random features for demo mode.
        # Replace normed_features with your feature vector
        normed_features = features/np.sum(features)

        return normed_features
```

3.3 Surface Normal Histograms

Once the surface normal data has been captured a histogram is created using bin ranges from -1.1 to 1.1, given that the surface normals are confined to this range. The function implementation used to create surface normal histograms for the PR2 can be seen in Listing XXXX.

Listing 8: insert text

```
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)
        x_hist = np.histogram(norm_x_vals, bins=50, range=(-1.1,1.1))
        y_hist = np.histogram(norm_y_vals, bins=50, range=(-1.1,1.1))
        z_hist = np.histogram(norm_z_vals, bins=50, range=(-1.1,1.1))

        # TODO: Concatenate and normalize the histograms
        features = np.concatenate((x_hist[0], y_hist[0], z_hist[0])).astype(np.float64)
        # Generate random features for demo mode.
        # Replace normed_features with your feature vector
        normed_features = features/np.sum(features)

        return normed_features
```

3.4 Capturing Data

In order to generate enough meaningful data for our SVM classifier model, a script called `capture_feature.py` was implemented. This script, in conjunction with a ROS simulation of the RGBD camera, removes gravity so that a known object model floats in front of the RGBD sensor. The object is assigned randomised Roll, Pitch, and Yaw configurations for a set number of iterations. At each new orientation, the simulated RGBD sensor will capture point cloud data. From each of these point clouds we extract RGB, and surface normals to create histogram features. The object model label (e.g. biscuits) is saved, along with the HSV and surface normal histogram features. The full implementation of `capture_features.py` can be found in Appendix XXXX. Figure XXXX shows an example of data being captured for an object model.

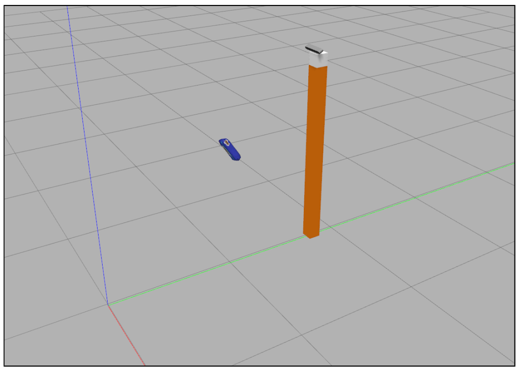


Figure 20: text

3.5 Training the SVM

Once the dataset has been captured the Python library `sklearn` was used to create a SVM classification model. This was implemented in a script called `train_svm.py`, which can be seen in full in Appendix XXXX. There are a couple of choices that can be made when optimising SVM performance. The first choice is between RGB or HSV colour spaces for our colour histogram feature (this choice is actually implemented in `capture_features.py`, but discussion of this choice is more suited to this section). Table XXXX shows reported model accuracy for two identical scenarios - the iterations was kept small at five, and the kernel for the SVM was linear, however, one scenario used RGB and the other HSV. There is an observable performance increase to the SVM classifier when using HSV.

Table 2: text

Iterations	**Colour Space**	**Kernel**	**Accuracy**
5	RGB	Linear	57.5%
5	HSV	Linear	77.5%

The other choice to make is the kernel function that is used for the SVM. Python library `sklearn` provides four off-the-shelf choices of kernel: linear, polynomial, rbf, and sigmoid. The best choice of kernel is not immediately clear. Figure XXXX shows the performance of the four kernels for for an increasing number of iterations.

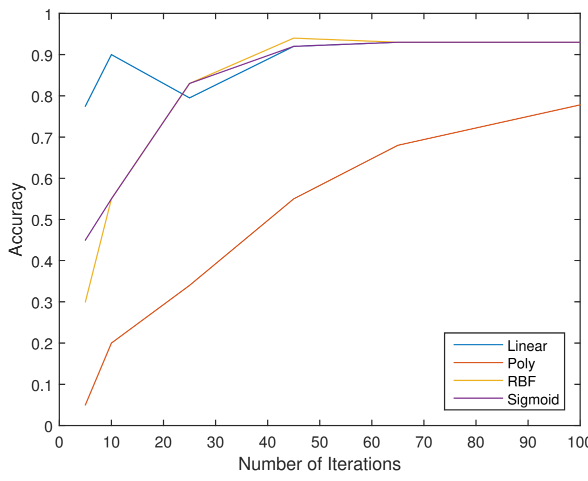
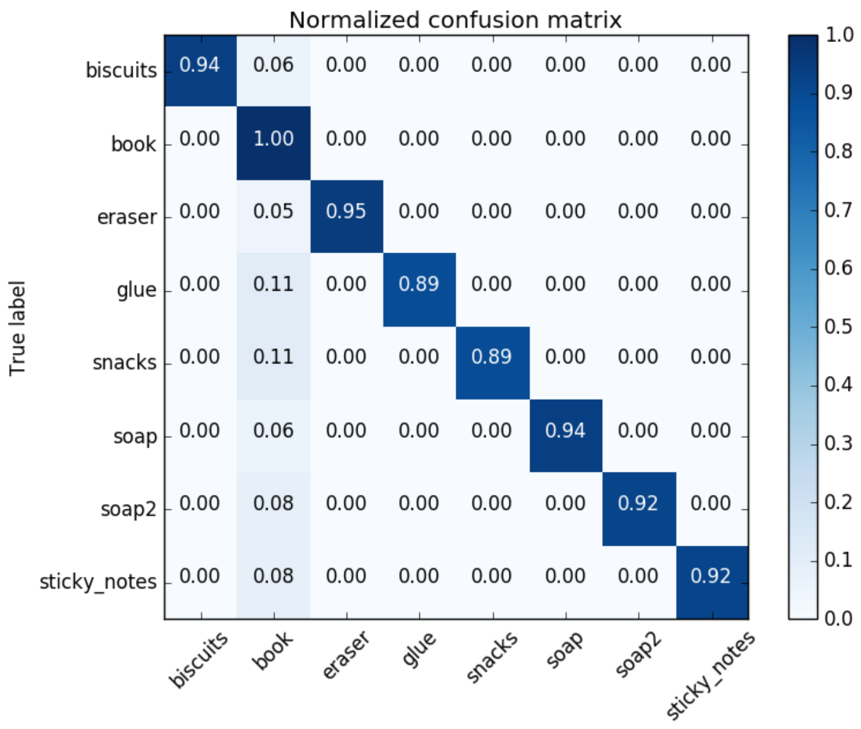


Figure 21: text

At 100 iterations for each object, there is no immediate distinction between the linear, rbf, and sigmoid models - most likely due to overfitting. One avenue to improve the performance further would be the use of regularisation, however, this has not been used in this instance. Experimentation with the PR2 robot implementation of the sensor revealed that the sigmoid kernel provided the most reliable performance. The normalised confusion matrix for 100 iterations using the sigmoid kernel can be seen in Figure XXXX. Notably, the model finds it difficult to distinguish between the box of snack and the book.



True label	biscuits	book	eraser	glue	snacks	soap	soap2	sticky_notes
biscuits	0.94	0.06	0.00	0.00	0.00	0.00	0.00	0.00
book	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
eraser	0.00	0.05	0.95	0.00	0.00	0.00	0.00	0.00
glue	0.00	0.11	0.00	0.89	0.00	0.00	0.00	0.00
snacks	0.00	0.11	0.00	0.00	0.89	0.00	0.00	0.00
soap	0.00	0.06	0.00	0.00	0.00	0.94	0.00	0.00
soap2	0.00	0.08	0.00	0.00	0.00	0.00	0.92	0.00
sticky_notes	0.00	0.08	0.00	0.00	0.00	0.00	0.00	0.92

Figure 22: text

4 PR2 Implementation & YAML File Output

The full implementation of the `object detection.py` Python script, which is designed to be run in conjunction with a ROS simulation of the PR2 robot, can be found in Appendix C. The first part of the code to be executed initialises the publishers and subscribers to ROS topics, loads the trained SVM model, and puts ROS into a persistant loop with `rospy.spin()`. The ROS subscriber that is set up makes a call to the `pcl_callback()` function everytime it receives point cloud data. This section of the code is shown Listing XXXX. The initial part of the `pcl_callback()` function simply implements to segmentation demonstrated in Section XXXX. Once the final stage of segmentation is completed, the processed point cloud data is published to the previously established topics ready for object recognition - this is shown in Listing XXXX.

The object recognition part of the code receives

4.1 Test Case 1

4.2 Test Case 2

4.3 Test Case 3

5 Results & Conclusion

6 Further Enhancements

Listing 9: text

```
if __name__ == '__main__':

    # TODO: ROS node initialization
    rospy.init_node('classification')

    # TODO: Create Subscribers
    pcl_sub = rospy.Subscriber('/pr2/world/points', pc2.PointCloud2, pcl_callback, queue_size=1)

    # TODO: Create Publishers
    pcl_table_pub = rospy.Publisher('/pcl_table', PointCloud2, queue_size=1)
    pcl_objects_pub = rospy.Publisher('/pcl_objects', PointCloud2, queue_size=1)
    pcl_cluster_pub = rospy.Publisher("/pcl_cluster", PointCloud2, queue_size=1)
    object_markers_pub = rospy.Publisher("/object_markers", Marker, queue_size=1)
    detected_objects_pub = rospy.Publisher("/detected_objects",
                                             DetectedObjectsArray, queue_size=1)

    # TODO: Load Model From disk
    model = pickle.load(open('model.sav', 'rb'))
    clf = model['classifier']
    encoder = LabelEncoder()
    encoder.classes_ = model['classes']
    scaler = model['scaler']

    # Initialize color_list
    get_color_list.color_list = []

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

Listing 10: text

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


##############################################################################
# TODO: Convert PCL data to ROS messages
##############################################################################
    ros_cloud_table = pcl_to_ros(cloud_table)
    ros_cloud_objects = pcl_to_ros(cloud_objects)
    ros_cluster_cloud = pcl_to_ros(cluster_cloud)


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

7 Appendix XXXX

```python
#!/usr/bin/env python
import numpy as np
import pickle
import rospy

from sensor_stick.pcl_helper import *
from sensor_stick.training_helper import spawn_model
from sensor_stick.training_helper import delete_model
from sensor_stick.training_helper import initial_setup
from sensor_stick.training_helper import capture_sample
from sensor_stick.features import compute_color_histograms
from sensor_stick.features import compute_normal_histograms
from sensor_stick.srv import GetNormals
from geometry_msgs.msg import Pose
from sensor_msgs.msg import PointCloud2


def get_normals(cloud):
    get_normals_prox = rospy.ServiceProxy('/feature_extractor/get_normals', GetNormals)
    return get_normals_prox(cloud).cluster


if __name__ == '__main__':
    rospy.init_node('capture_node')

    models = [\
       'biscuits',
       'soap',
       'book',
       'soap2',
       'glue',
       'sticky_notes',
       'snacks',
       'eraser']

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

    for model_name in models:
        spawn_model(model_name)

        for i in  range(100):
            # 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 < 5:
                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=True)
            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_set.sav', 'wb'))
```

8 Appendix XXXX

```python
#!/usr/bin/env python
import pickle
import itertools
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn import cross_validation
from sklearn import metrics

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion␣matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm. sum(axis=1)[:, np.newaxis]
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange( len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    thresh = cm. max() / 2.
    for i, j in itertools.product( range(cm.shape[0]),  range(cm.shape[1])):
        plt.text(j, i, '{0:.2f}'. format(cm[i, j]),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True␣label')
    plt.xlabel('Predicted␣label')

# Load training data from disk
training_set = pickle.load( open('training_set.sav', 'rb'))

# Format the features and labels for use with scikit learn
feature_list = []
label_list = []

for item in training_set:
    if np.isnan(item[0]). sum() < 1:
        feature_list.append(item[0])
        label_list.append(item[1])

print('Features␣in␣Training␣Set:␣{}'. format( len(training_set)))
print('Invalid␣Features␣in␣Training␣set:␣{}'. format( len(training_set)- len(feature_list)))

X = np.array(feature_list)
# Fit a per-column scaler
X_scaler = StandardScaler().fit(X)
# Apply the scaler to X
X_train = X_scaler.transform(X)
y_train = np.array(label_list)

# Convert label strings to numerical encoding
encoder = LabelEncoder()
y_train = encoder.fit_transform(y_train)

# Create classifier
clf = svm.SVC(kernel='linear')

# Set up 5-fold cross-validation
kf = cross_validation.KFold( len(X_train),
                            n_folds=5,
                            shuffle=True,
                            random_state=1)

# Perform cross-validation
scores = cross_validation.cross_val_score(cv=kf,
                                          estimator=clf,
                                          X=X_train,
                                          y=y_train,
                                          scoring='accuracy'
                                         )
print('Scores:␣' +  str(scores))
print('Accuracy:␣%0.2f␣(+/-␣%0.2f)' % (scores.mean(), 2*scores.std()))

# Gather predictions
predictions = cross_validation.cross_val_predict(cv=kf,
                                          estimator=clf,
                                          X=X_train,
                                          y=y_train
                                         )

accuracy_score = metrics.accuracy_score(y_train, predictions)
print('accuracy␣score:␣'+ str(accuracy_score))

confusion_matrix = metrics.confusion_matrix(y_train, predictions)

class_names = encoder.classes_.tolist()


#Train the classifier
clf.fit(X=X_train, y=y_train)

model = {'classifier': clf, 'classes': encoder.classes_, 'scaler': X_scaler}

# Save classifier to disk
pickle.dump(model,  open('model.sav', 'wb'))
```

```
# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(confusion_matrix, classes=encoder.classes_,
                      title='Confusion␣matrix,␣without␣normalization')

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(confusion_matrix, classes=encoder.classes_, normalize=True,
                      title='Normalized␣confusion␣matrix')

plt.show()
```