universidade de aveiro

Sistemas Inteligentes

Aprendizagem não Supervisionada Incremental

Docente

Luís Seabra Lopes

Mestrado em Engenharia Informática, 2016/2017

Autores:

José Horácio Fradique Duarte, Mec. 62406

Fernando José Fradique Duarte, Mec. 48257

Contents

1. Introduction

The objective of this assignment was to implement an application that "sees" object views and assigns those object views to object categories. The problem can be stated as:

Let:

V be the set of the object views.

C be the set of the object categories.

We want to find f, such that:

$\mathbf{f: V \rightarrow C}$

In practice, **f** should have certain properties of interest. In particular, the application should be capable of learning new object categories that it did not know of, the object categories are not known a priori and the application should use some sort of online/incremental k-means for efficiency. A memory management mechanism is also needed to control the number of features in memory, etc. A more detailed specification for such **f** is provided in the specification given for this assignment.

In the following sections of this report we present and discuss: issues, decisions, implementation details, the technologies used, architecture, optimization, etc.

2. Issues, Constraints and Decisions

In the implementation of the applications presented in this report, we tried to follow the specification that was given. We did not, however, follow it strictly. As an example, the main application is not capable of learning new object categories. The online/incremental k-means, in particular, is still a difficult problem to solve and we could not find well documented and explained solutions that could be implemented, in the literature. Therefore, we tried to simplify the problem by making some assumptions from which we constructed an application that can train and then assign object views to object categories.

Our main assumptions include:

- The object categories are known a priori.
- We use a custom implementation of K-means, the number of clusters is dynamic, and clusters can be split and merged.
- There is a memory management mechanism that includes the notions of: redundancy, memory decay and reinforcement and tries to control the number of features in memory by using a memory decay threshold, following from the specification that was given.

- The implementation uses the Point Cloud Library (PCL) and c++98. C++98 is a PCL requirement. We did successfully compiled a sample application using PCL with c++11, by using extra flags for the compiler, but the PCL is not guaranteed to work with c++11. Python bindings for the PCL also exist, but they provide only a subset of the PCL API, so we decided to use c++98.
- We use the Washington RGB-D Object Dataset [7] as our dataset and we assume that the object views that it provides are already cleaned up from outliers.
- The object views are represented by a set of spin images taken from a set of key points obtained using a voxel grid filter.

3. Discrete Model

In this section we present the discrete model of the main application.

3.1 Type System

The main application data types are: *feature*, *category*, *cluster* and *memory*.

Definition 1.
A feature represents the surface/geometry characteristics of an object near a key point. It has a unique identifier, local and global weights, it is assigned to a cluster and to a category and it has a representation as a histogram in R^{153}.

Given some $v \in V$, V is the set of object views. Then:

v = {*feature*}, *feature* $\in F$, F is the set of features,

feature = (id, histogram, lw, gw, cluster, category)

id	Unique identifier.
histogram	The features' representation as a R^{153} histogram.
lw, gw	Local and global weights, respectively.
cluster, category	The cluster and the category the feature is assigned to.

Definition 2.
A category represents an object category, e.g., apple, orange, pear, peach, etc.

Given some $c \in C$, C is the set of object categories and K, K is the set of clusters. Then:

c = (id, name, featuresIds, histogram, performanceMeasures)

id	Unique identifier.
name	Category name, e.g., apple.
featuresIds	The ids of the features assigned to the category.

	{ id : $\exists^1$ $feature_i$, id = $feature_i.id$, ∀i,j $id_i \neq id_j$,
	Assigned($feature_i$, *c*) }
histogram	Category histogram in R^N, N = $\|K\|$.
performanceMeasures	Performance measures: accuracy, precision, recall, etc.

Definition 3.

A cluster represents a set of features that are more or less close to each other.

Given some $k \in K$, K is the set of clusters. Then:

k = (id, centroid, features, sumFeatures, sumWeights, wss, bss)

id	Unique identifier.
centroid	The centroid of the cluster in R^{153}.
features	Set of features assigned to the cluster.
sumFeatures	Weighted sum of the features.
sumWeights	Sum of the weights of the features.
wss (Within Sum of Squares)	Intra-cluster distance measure.
bss (Between Sum of Squares)	Inter-cluster distance measure.

Definition 4.

The memory stores the set of features, the set of clusters and the set of categories.

Let C, be the set of object categories.

Let K, be the set of clusters.

Let F, be the set of features.

memory = (C, *K,* F)

4. Implementation

For this assignment we implemented 2 applications. In this section we discuss implementation details. The applications will be discuss in the next section.

4.1 Technologies Used

We used the following technologies:

- PCL - The Point Cloud Library Version 1.7.2
- C++98
- No other dependencies on external libraries.

PCL Class|Paremeters	Usage	Application Class
VoxelGrid • voxel size = 0.015	Point cloud downsampling and Key points extraction.	FeatureExtractor
SpinImageEstimation • *window width = 8* • *support length = 0.1 m* • *support angle = 90º* • *min points neighbors = 0*	Create features to represent the object views.	FeatureExtractor
KdTreeFLANN • *nearestKSearch*(...) • *radiusSearch*(...) • search *radius = 0.065*	Computation of the distance between object views: knn and radius search.	Searcher Memory

Table 1: PCL classes used by the main application classes.

Notes:

- The parameters/values shown in the table above, except for the search radius, follow from the ones used in [4].
- PCL Documentation, API, tutorials, etc. can be found at [3] [5] [6].

4.2 Dataset

We used the Washington RGB-D Object Dataset [7], [8], which has views of 300 physically distinct everyday objects, taken from different viewpoints, organized into 51 categories arranged using WordNet hypernym-hyponym relationships [9]. The dataset contains 3D point clouds of views of each object, in the PCD format:

- Each point is stored with 6 fields: the 3D coordinates (x, y, z), the color packed into 24 bits with 8 bits per channel (RGB), and the image coordinates of the point (imX, imY).
- In the PCD files, for a given point (x, y, z), the x-axis points to the right along the image plane, y points into the image plane, and z points upwards along the image plane.

Note: We assumed that the views from the dataset were already cleaned up and we did not use a Conditional Removal filter to remove outliers, as in [4].

4.3 Architecture and Main Application Modules/Classes Responsibilities

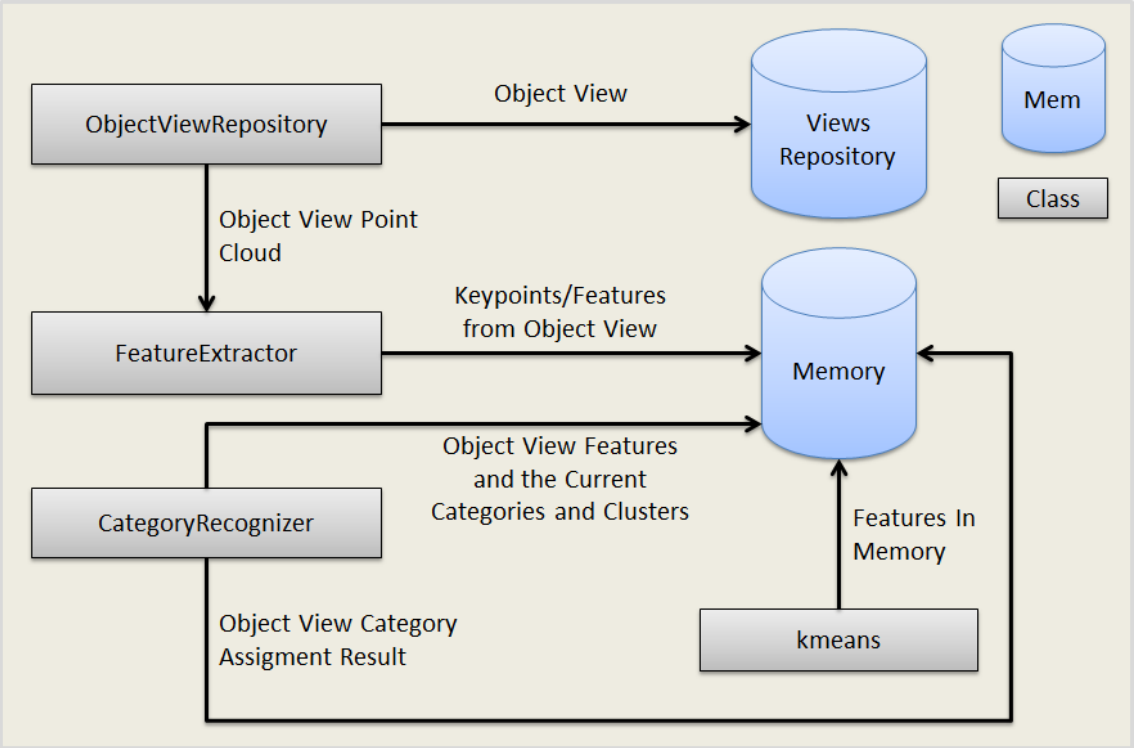

Fig 1: Main application architecture.

Module (Class)	**Main Responsibilities**
ObjectViewRepository	• Read object views stored on disk. • Shuffle categories and views of objects to use during training/testing.
FeatureExtractor	• Extract key points from the object views. • Compute representations of key points (Spin Images).
Searcher	• Knn search (based on the distance between Spin Images): including the distance in a specified search radius. • Compute the Euclidean Distance between histograms of views and categories.
Viewer	• Visualization of point clouds and histograms.
Memory	• Manage memory: forgetting, decay, redundancy, weight redistribution, etc. • Store application related data: features, categories, clusters, etc.
Kmeans	• Assign features to clusters. • Manage clustering quality: performing re-clustering via split/merge operations according to the WSS/BSS clustering quality measures.

CategoryRecognizer	• Assign each object view to a category.
Category	• Represents a category.
Cluster	• Represents a cluster.
FeatureMetadata	• Store all information related to a feature: local/global weight, cluster/category assignment, etc.
TestResult	• Store the performance measures obtained during a test run: accuracy, precision, recall, etc.
PerformanceManager	• Store all the performance measurements taken across test runs.

Table 2: Main application modules/classes responsibilities.

4.4 Application Main Work Flows

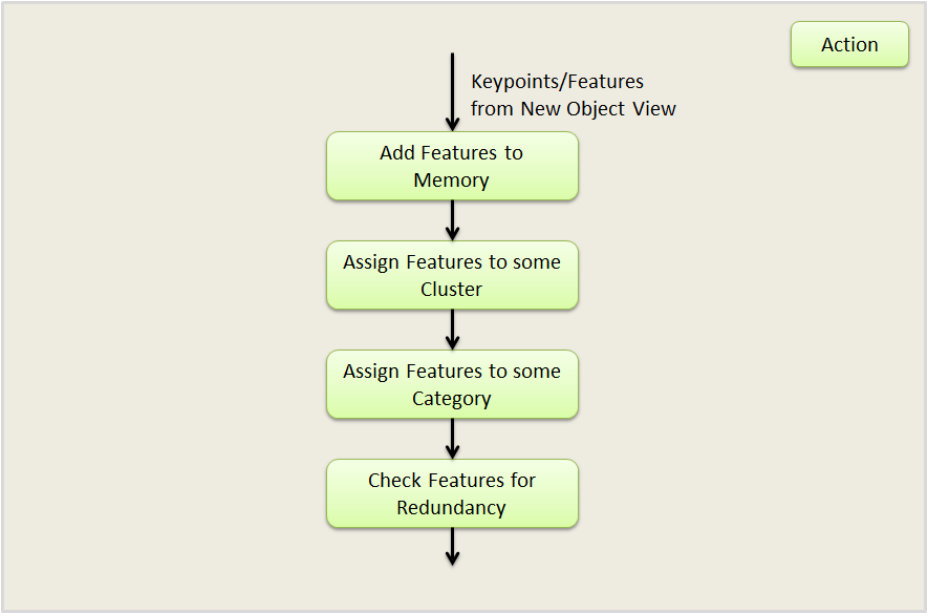

Fig 2: Processing the features from a new seen object.

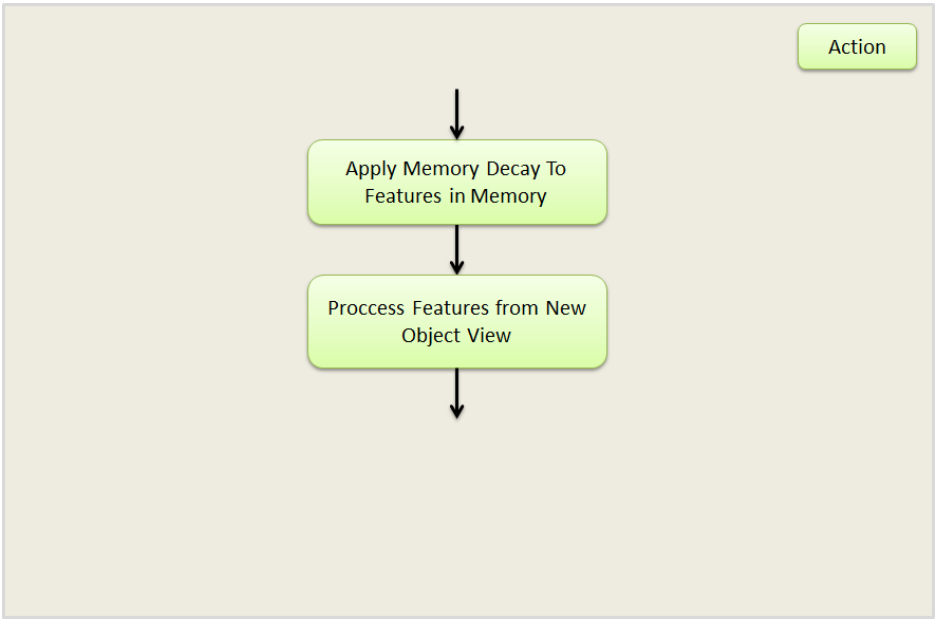

Fig 3: Applying the memory decay factor.

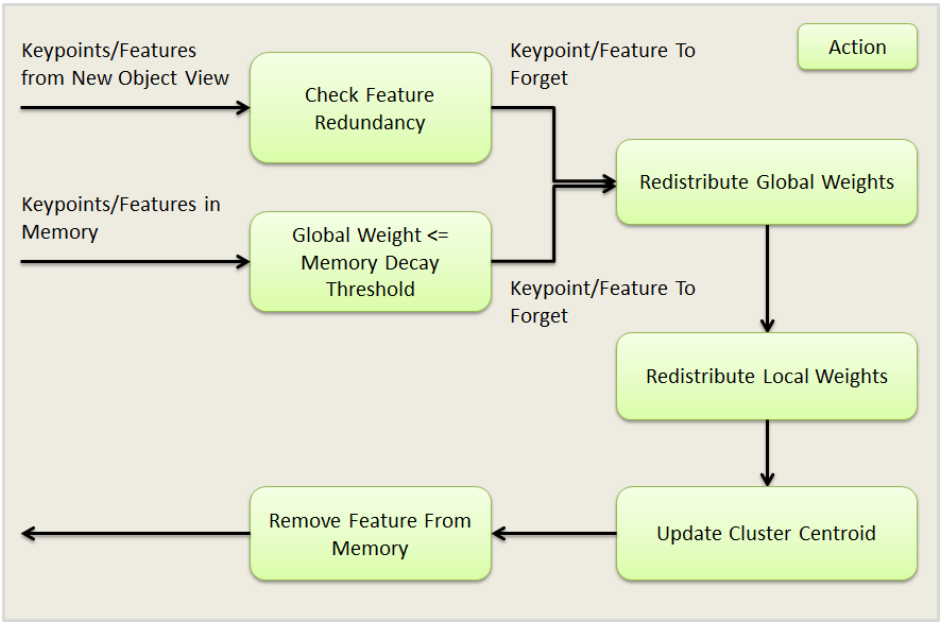

Fig 4: Forgetting a feature.

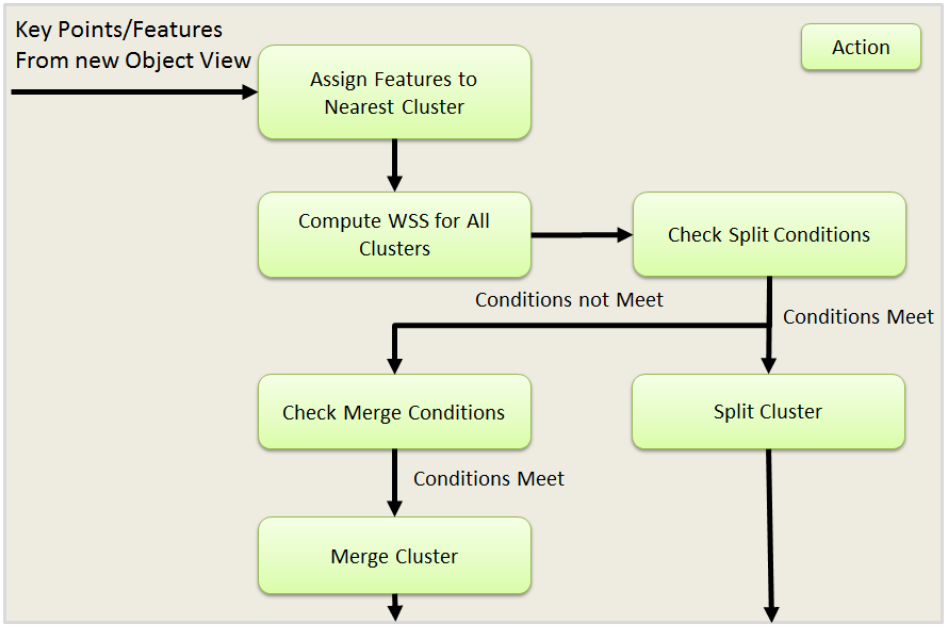

Fig 5: Assigning features to clusters and checking clustering quality.

4.5 Data Structures Implementation

The implementation of the data types follows the discrete model, except for the category. In our implementation the category does not own/manages its' histogram. This is future work. The categories histograms are computed by the CategoryRecognizer class.

Data Type	**Class Implementation**
feature	FeatureMetadata
category	Category
cluster	Cluster
memory	Memory

Table 3: Main data types class implementation.

C, K and F are stored in the Memory class using dictionaries, more specifically:

- unordered_map<int, Cluster> clusters
- unordered_map<int, Category> categories
- unordered_map<int, FeatureMetadata> featuresMetadata

Note: The use of the unordered_map data structure instead of the std::map c++ standard data structure allows us to have O(1) access to the dictionary keys. This has to do with the internal implementation of these data structures.

The features ids in the Category and Cluster classes are stored using the unordered_set data structure.

Note: The use of this data structure allows us to have O(1) access to the i-th feature id in the set. This situation is analogous to the unordered_map versus std::map situation described earlier.

We use a custom implementation of K-means, no dependencies on any external library, using c++98 to manage the clusters.

4.6 Categories and Object Views Histograms

Let V be the set of the object views and C be the set of the object categories.

The $\mathbf{V} \rightarrow \boldsymbol{C}$ mapping is obtained by computing the Euclidean distance between the histograms of each $v \in \mathrm{V}$ and each $c \in C$ and choosing Min(|histogram_v, histogram_c|) for every v, i.e., an histogram is computed for every c and every v.

To avoid problems of dominance, e.g., one category dominates the others, these histograms are normalized and they have the following main properties.

A histogram h of a view|category is based on the local weights of that view|category and it has the following properties:

- Dim(h) = N, N is the number of clusters, h is a vector in R^N.
- h is normalized using $\sqrt{\sum_1^{\mathrm{N}} \mathrm{lw}^2}$, lw_i is the sum of the local weights of all the features that belong to:
    - cluster i, i = 1, ..., N
    - that view|category.

As an example, consider the following table:

Histogram	**Cluster 0**	**Cluster 1**	**Cluster 2**	**Cluster 3**	**Cluster 4**
View	0	0.525311	0.642653	0.546846	0.109564
Category: bowl Id: 0	0.000596445	0.478811	0.746093	0.440463	0.141696
Category: ball Id: 1	0.9933	0	0	0.049048	0.0294286

Category Id:

Predicted: 0 || Real: 0

4.7 Memory Management Implementation

Let C be the set of the object categories.

Let K be the set of clusters.

Let F be the set of features in memory.

The application memory main data structures are: C, K and F.

The application memory:

- Implements a forgetting mechanism of features that includes the notions of: feature redundancy, memory decay and reinforcement. This follows from the specification given for this assignment.

The following rule φ:

$$\varphi = \begin{cases} |feature_i,\ feature_j| \leq \varepsilon,\ feature_i \wedge feature_j \in F,\ \mathrm{i} \neq j \ \rightarrow feature_i \text{is redundant} & (1) \\ GW(feature_i) \leq \delta \qquad \rightarrow forget\ feature_i & (2) \end{cases}$$

, where $GW(feature_i)$ represents the global weight of feature$_i$ and |a, b| represents the distance between a and b, establishes: in (1), the condition for feature redundancy using the Feature Redundancy Threshold ε and in (2) the condition to forget features using the Memory Decay Threshold δ. In both cases, (1) and (2), feature$_i$ will be forgotten.

Then, forgetting feature$_i$ will follow the algorithm:

Redistribute Global Weights

Redistribute Local Weights

*Update the cluster centroid to whic*h feature$_i$ *is assigned to*

Remove feature$_i$ *from memory*

4.7.1 Global Weight Redistribution

Let K *be* the set of clusters, $k \in K$, F is the set of features in memory, $f \in F$, f is the feature to use for weight redistribution and θ is the search radius to use for weight redistribution. Global weight redistribution follows the algorithm:

Assumptions: $|K| > 0$

Find the ≤ 3 *closest* k *to* f

A = {features from the ≤ 3 *closest* k *to f}* $\setminus \{ f \}$

D = $\{x \in A : |x, f| \leq \theta\}$

For $x \in D$

AssignNewGlobalWeight(x)

UpdateCentroid(k_i), $k_i \in K$, *Assigned*(x, k_i)

$\leq$ 3 means: at least 1 and at most 3.

In practice however, the domain search restriction in '*Find the* $\leq$ *3 closest* k *to f*' does not scale. This is because multiple point clouds are being copied and this does not scale well. Another concern is that, the effectiveness of such restriction is dependent on the number of clusters and their respective sizes. Therefore, we decided to use another approach that uses a snapshot of all the features in memory at a given instant in time and the algorithm becomes:

S_f = *Snapshot(F)*

D = {x $\in$ S_f *: |x, f|* $\leq \theta$*, x* $\neq$ *f}*

For x $\in$ *D*

AssignNewGlobalWeight(*x*)

UpdateCentroid(k_i), $k_i \in K$, *Assigned*(*x,* k_i)

Note: The snapshot is only computed once for each object view and this is not yet the final solution for the point cloud copy problem!

And again for performance reasons, the search radius computation rules changed from:

$$\theta = \begin{cases} avg(|f,\ k_j| + |f,\ k_i|) & ,\ |K| > 1,\ i \neq j, (k_i,\ k_j\ are\ the\ 2\ closest\ clusters\ to\ f) \\ |f, k| & ,\ |K| = 1 \end{cases}$$

- $\boldsymbol{avg}$ refers to the arithmetic mean.

To:

θ = 0.065

θ is a fixed value => less point cloud copies! Point cloud copies are a major cause for poor performance. This value has no theoretical support behind it. It is just a value we chose by looking at the distances computed by KdTreeFLANN.

Global weight redistribution is implemented in:

Class	**Source Files**	**Function**
Memory	Memory.h Memory.cpp	Using a snapshot: redistributeGlobalWeights(unordered_set<int> &toForget, KdTreeFLANN<Histogram<153> > &kdtree, vector<int> &mapping, double searchRadius) Not using a snapshot: redistributeGlobalWeights(

		int featureId, FeatureMetadata &featureMetadata, double searchRadius)

4.7.2 Local Weight Redistribution

Let C be the set of object categories, $c \in C$, F is the set of features in memory, $f \in F$, f is the feature to use for local weight redistribution . Local weight redistribution follows the following algorithm:

A = {*x* : *Assigned*(*x,* c), *Assigned*(f, c), $\{x, f\} \in F$, $x \neq f$}

For x ∈ *A*

AssignNewLocalWeight(x)

Note: Set A can be obtained from a snapshot of F or from F. In the case of new features a snapshot of F will be used since all new features from an object view belong to the same category.

Local weight redistribution is implemented in:

Class	**Source Files**	**Function**
Memory	Memory.h Memory.cpp	Using a snapshot: redistributeLocalWeights(unordered_set<int> &toForget, int categoryId) Not using a snapshot: redistributeLocalWeights(int featureId, FeatureMetadata &featureMetadata)

4.7.3 Feature Redundancy

The feature redundancy check follows the following algorithm:

Let K be the set of clusters, $k \in K$, F the set of features in memory, f a new seen feature : $f \in F$ and ε the Feature Redundancy Threshold.

Assumptions: $|K| > 0$

Find the $\leq$ *3 closest* k *to f*

A = {*features from the* $\leq$ *3 closest* k *to f*} \ {*f* }

Find f_i *: Min*($|f, f_i|$), f_i *in A*

CheckFeatureRedundancy(*f,* f_i)

CheckFeatureRedundancy(*f,* f_i)

$|f, f_i| \leq \varepsilon$

Redundant(*f*)

ForgetFeature(*f*)

≤ 3 means: at least 1 and at most 3.

In practice however, the domain search restriction in '*Find the* ≤ 3 *closest* k *to f*' does not scale. This is because multiple point clouds are being copied and this does not scale well. Another concern is that, the effectiveness of such restriction is dependent on the number of clusters and their respective sizes. Therefore, we decided to use another approach that uses a snapshot of all the features in memory at a given instant in time and the algorithm becomes:

Assumptions: N = 2

A = Snapshot(F)

B = $\{f_j : f_j \in F, f_j \in MinN(|f, f_i|)\}$, $B \subsetneq A$

CheckFeatureRedundancy(f, B)

CheckFeatureRedundancy(*f, B*)

For f_i in *B*

$f_i \neq f$ and ¬MutualRedundant(f_i, f) and $|f, f_i| \leq \varepsilon$

Redundant(*f*)

SaveDistance($|f, f_i|$)

ForgetFeature(*f*)

Where B has the N nearest features to feature *f*, *MinN*() finds such set B and MutualRedundant(A, B) refers to the case where features A and B are both new seen features and they make each other redundant, i.e.,:

Min(|A, C|) = Min(|A, B|),

Min(|B, D|) = Min(|B, A|), C and D $\in F$, C ≠ A, D ≠ B, Min(|A, B|) ≤ ε, Min(|B, A|) ≤ ε

Mutual redundancy is checked using the distance value and assuming that for two pair of features (A,B) and (C, D) where (A,B) ≠ (C, D) => |A,B| ≠ |C, D|.

The feature redundancy check is implemented in:

Class	**Source Files**	**Function**
Memory	Memory.h, Memory.cpp	checkRedundancy(vector<int>& featsIdx)

4.7.4 Memory Decay Factor

Let π be the Memory Decay Factor and F the set of features in memory. Then:

We assume π = 0.99985.

π is applied to F \ {features added for the first time}, i.e., applied to the features' weights (local and global), such that: some weight ω will have the new value $\omega = \omega * \pi$.

Again, there is no solid ground theoretical support for the value of π.

4.7.5 Remove Feature from Memory

Removing a feature f_i, assigned to cluster k_j and category c_l, from memory involves:

UpdateSums(k_j)

UnAssign(f_i, k_j)

UnAssign(f_i, c_l)

RemoveFromMemory(f_i)

RecomputeCentroid(k_j)

This is implemented in:

Class	Source Files	Function
Memory	Memory.h Memory.cpp	remFeatureFromMemory(int featureId, FeatureMetadata &featureMetadata)

4.7.6 Snapshots

Using a snapshot means that all features in memory will be collected at a time instant and they will be used to create 1 index that will be used for N operations of knn and radius search, among features. This means that the feature that is being checked for is also included in the snapshot and care must be taken do avoid redundancy by itself, mutual redundancy and the introduction of 0 distances in the calculations, which will lead to a division by zero. Snapshots are used for: feature redundancy check and global and local weights redistribution.

4.7.7 Exceptions and Error Conditions

We decided to use exceptions to deal with execution conditions that we think are not acceptable. This also gives us more confidence about the quality of the implementation and its' results. For example: any calls to the empty constructor of the classes: Category, Cluster and FeatureMetadata are considered a major fault and will be treated with an exception. This is because such call is injecting a malformed object into the system, empoisoning it over time, and such a call should not happen anyway.

Finally, a major challenge is to find appropriate ε, δ, θ and π.

4.8 K-means Implementation

A customized version of K-means was implemented in order to allow for the dynamic creation of clusters as opposed to the traditional implementation of K-means whereby the number of clusters is

defined a priori and remains unchanged thereafter. In this implementation the number of clusters is not specified a priori and may change as new features are added or forgotten. This is accomplished by using the Between Sum of Squares (BSS) and the Within Sum of Squares (WSS) clustering quality measures to decide on the need to perform a split or a merge operation. Ideally a Split operation should occur if the WSS measure of a cluster as reached a specified threshold. Similarly, a merge operation should be performed between 2 clusters if their WSS measures are below a given threshold and the distance between them, that is, their BSS measure is also below a given threshold.

The algorithm comprises 2 steps:

- Step 1 concerns feature assignment to clusters,
- Step 2 concerns clustering quality checking.

Step 1 (pseudo code)

```
Repeat until convergence
  For each feature f
    assign f to the nearest cluster c
    recompute the centroid of c
```

Step 2 (pseudo code)

```
For maxCheckIters
  compute WSS for all clusters
  For each cluster c
    If WSS of c > wssThreshForSplit
      k = choose nearest point to the centroid of c
      create new cluster c^2 using k
      unassign all features previously assigned to c
      perform Step 1
    else If WSS of c < wssThreshForMerge
      for remaining clusters c^2 (c != c^2)
        If WSS of c^2 < wssThreshForMerge
          bss = compute BSS between c and c^2
          if  bss < bssThreshForMerge
            k = compute Euclidean distance between c and c^2
            create new cluster c^3 using k
            unassign all features previously assigned to c and c^2
            delete c and c^2
            perform Step 1
            break
```

In terms of parameterization, the algorithm accepts 4 parameters. The values used for each of these parameters were obtained through experimentation and posed a real challenge. Future work should consider more suitable ways of finding the appropriate values. In [10], a more detailed overview of these measures is presented.

Parameter name	Default value	Parameter description
maxCheckIters	1	The number of iterations used to perform Step 2, that is, to check clustering quality
wssThreshForSplit	0.12	Threshold value considered for splitting. All clusters whose WSS measure fall above this threshold are considered for splitting
wssThreshForMerge	0.07	Threshold value considered for merging. All clusters whose WSS measure falls below this threshold are considered for merging. This value is used in conjunction with the value of bssThreshForMerge
bssThreshForMerge	0.14	Threshold value considered for merging. All clusters whose BSS measure falls below this threshold are considered for merging. This value is used in conjunction with the value of wssThreshForMerge

Table 5: Main parameters used to parameterize K-means.

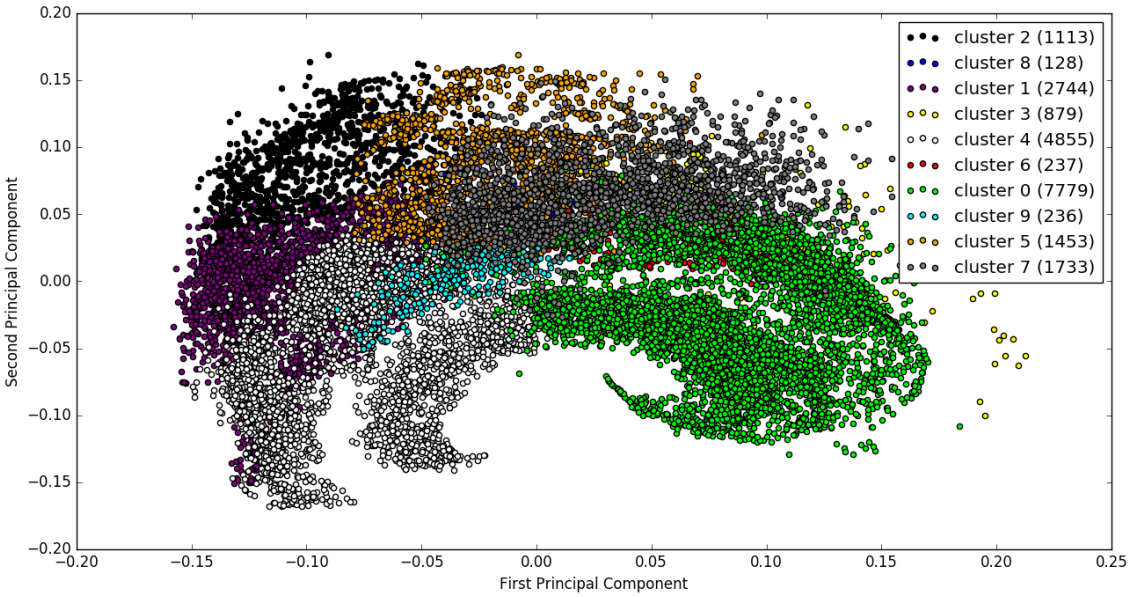

Fig 6: Resulting clustering for a test with 6 categories. Visualization using the first and second principal components obtained after applying PCA for 2d visualization.

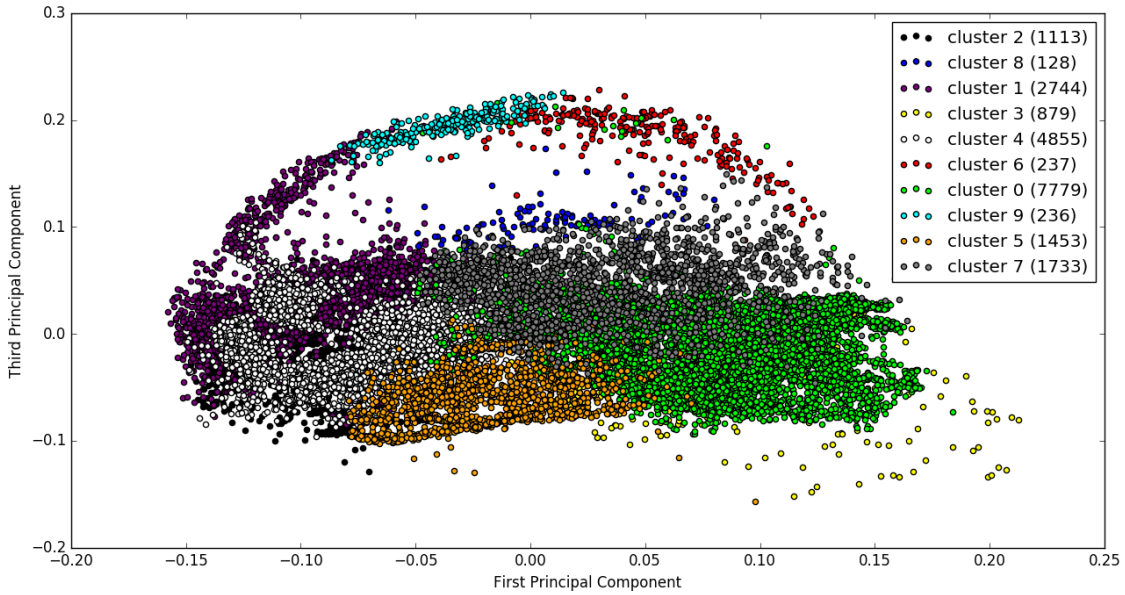

Fig 7: Resulting clustering for a test with 6 categories. Visualization using the first and third principal components obtained after applying PCA for 2d visualization.

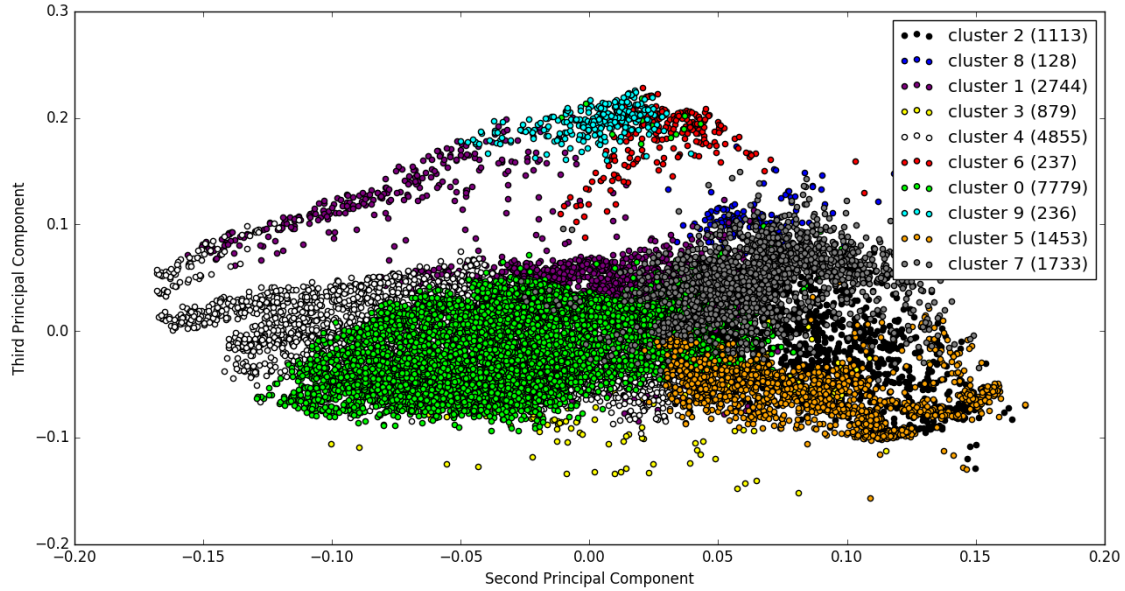

Fig 8: Resulting clustering for a test with 6 categories. Visualization using the second and third principal components obtained after applying PCA for 2d visualization.

5. Application Overview, Execution and Installation

We implemented 2 applications for this assignment. In this section we discuss among other things: the applications, how to execute them, how to install/compile the provided source code and its' dependencies, the applications' execution options, etc.

5.1 Application Overview

2 applications were implemented for this assignment:

Application Name	Purpose
Viz	An application for visualization and to experiment with different parameters for the creation of key points and spin images. Viz can be found in the SI_VIZ directory.
Pcd_read	The main application that trains and assigns views to object categories. Pcd_read can be found in the SI_CODE directory.

5.1.1 Viz

The viz application, found in the SI_VIZ directory, allows the visualization of object views, key points and spin images and has the following execution options:

point_cloud_path The path to the object views to load/view.

configuration_file The path to the configuration file.

./viz <point_cloud_path> <configuration_file>

Usage example: ./viz /home/u/pcl/SI_CODE/views/apple/apple_1_1_1.pcd config

How to use the application:

Step 1	Run the application in the terminal, e.g., with: ./viz /home/u/pcl/SI_CODE/views/apple/apple_1_1_1.pcd config
Step 2	The console will print information about: • The settings being used to create the key points and the spin images. • The number of points of the original point cloud and the number of key points obtained by the configuration. • The first spin image histogram contents. • The index of the 3 closest spin images to the spin image at index 3, in the key points point cloud, and their respective distances using the KdTreeFLANN. **Note**: Not all the information will be outputted immediately.
Step 3	A window showing the original point cloud points and the key points, side by side, will appear. Several commands can be used to interact with this window. With the window selected it is possible to: • zoom in/out, using the mouse wheel, • increase/decrease the size of the points using the +/- keys, for a better view of the object,

	• rotate the object by clicking in the window and dragging the mouse in the desired direction, etc.
Step 4	With the window selected, press the e key or click on the windows' close button and a new window will appear, showing the key points, marked green, in the original point cloud. The interaction with this window is similar to the previous one.
Step 5	With the new window selected press the e key or the windows' close button, and a new window will appear showing the histogram of the first key point. Interaction with this window is similar to the previous ones.
Step 6	All windows are showing and can be further analyzed. Also, all the information outputted by the application can now be seen on the terminal.

```
** \brief PCLVisualizerInteractorStyle defines an unique, custom VTK
 * based interactory style for PCL Visualizer applications. Besides
 * defining the rendering style, we also create a list of custom actions
 * that are triggered on different keys being pressed:
 *
 * -        p, P   : switch to a point-based representation
 * -        w, W   : switch to a wireframe-based representation (where available)
 * -        s, S   : switch to a surface-based representation (where available)
 * -        j, J   : take a .PNG snapshot of the current window view
 * -        c, C   : display current camera/window parameters
 * -        f, F   : fly to point mode
 * -        e, E   : exit the interactor\
 * -        q, Q   : stop and call VTK's TerminateApp
 * -       + / -   : increment/decrement overall point size
 * -        g, G   : display scale grid (on/off)
 * -        u, U   : display lookup table (on/off)
 * -  r, R [+ ALT] : reset camera [to viewpoint = {0, 0, 0} -> center_{x, y, z}]
 * -  CTRL + s, S  : save camera parameters
 * -  CTRL + r, R  : restore camera parameters
 * -  ALT + s, S   : turn stereo mode on/off
 * -  ALT + f, F   : switch between maximized window mode and original size
 * -        l, L           : list all available geometric and color handlers for
 *                           the current actor map
 * -  ALT + 0..9 [+ CTRL]  : switch between different geometric handlers (where available)
 * -        0..9 [+ CTRL]  : switch between different color handlers (where available)
 * -
 * -  SHIFT + left click   : select a point
 * -        x, X   : toggle rubber band selection mode for left mouse button
 *
 * \author Radu B. Rusu
 * \ingroup visualization
```

Fig 9: List of the available commands to interact with the windows in the viz application. These commands can be found in interactor_style.h, in the PCLs' source code.

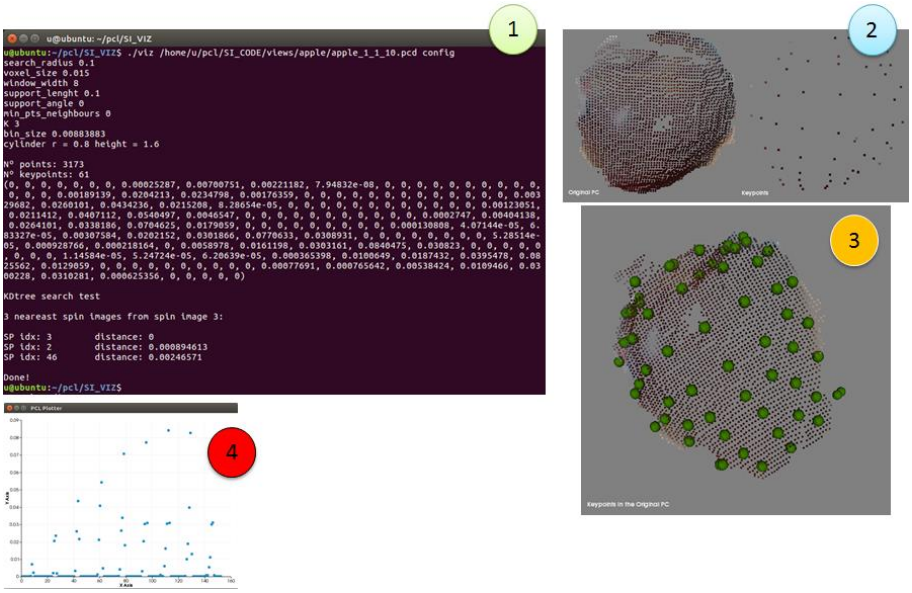

Fig 10: 1 - Viz console output. 2 - The original point cloud and the Key points. 3 - The key points on the original point cloud. 4 - Spin image histogram.

5.1.2 Pcd_read

Pcd_read is the main application, found in the SI_CODE directory, that trains and assigns object categories to object views. This application has the following execution options:

object_views_directory	The path of the object views to use.
num_categories_to_use	num_categories_to_use > 0, num_categories_to_use $\leq$ num_categories_available
%_views_4_training	$1 \leq$ %_views_4_training ≤ 100
forget_features	forget_features $\in$ {true, false}, true $\rightarrow$ application will forget features
Debug	debug $\in$ {true, false}, true $\rightarrow$ application will output debug information
number_of_tests	Number of tests to perform.
number_of_views_to_see_per_category	Number of views per category to use.
use_rand_shuffle	use_rand_shuffle $\in$ {true, false}, false $\rightarrow$ the tests will not be random. They will use always the same shuffle order.
configuration_file	The path to the configuration file.

Usage example: ./pcd_read views/ 20 65 true false 1 25 false config

How to use the application:

Step 1	Edit the configuration file and choose the values that you want to use.

Step 2	Run the application in a terminal. The application will output information about the configuration been used, e.g.,: Configuration voxel_size 0.01 spin_imgs_search_radius 0.065 window_width 8 support_length 0.1 support_angle 0 min_pts_neighbours 0 redistribute_GWeights_search_radius 0.05 Forget Features Off Number Of Tests To Perform: 1 Number Categories used in the tests: 10 % Views for training: 0.65 Views Dir: ../SI_CODE/views/ Memory Decay Factor: 0.99985 Memory Decay Threshold: 0.97775 Feature Redundancy Threshold: 0.000126 Max Number Features In Memory: 35000 bin_size 0.00574524 cylinder r = 0.52 height = 1.04 About the tests been performed, e.g.,: Test 1 Nº Categories used: 10 Nº Categories Available: 51 Nº Views per Categories: 20 Nº Views for Training: 12 Nº Views for Testing: 8 During the training phase the application will output the path of the object views being processed, e.g.,: Task: Training View: ../SI_CODE/views/apple/apple_1_4_167.pcd View: ../SI_CODE/views/pliers/pliers_6_1_107.pcd … During the test phase the application will output the path of the object views being processed and the assignment result, e.g.,: Training done! Starting Testing phase!

Task: Testing

View: ../SI_CODE/views/apple/apple_1_4_93.pcd

Category Id:

Predicted: 18 || Real: 0

Category:

Predicted: peach || Real: apple

View: ../SI_CODE/views/cereal_box/cereal_box_1_4_39.pcd

Category Id:

Predicted: 3 || Real: 3

Category:

Predicted: cereal_box || Real: cereal_box

…

When the tests are all performed, the application will output the results and the total time of execution, e.g.,:

Testing done!

RESULTS

Test 1

Test performance

21 cluster(s) used in the test

40 samples used: 36 correct classifications, accuracy: 90.00%

Category Measures

Category garlic ,precision 88.89% ,recall 100.00%

Category bell_pepper ,precision 80.00% ,recall 100.00%

Category dry_battery ,precision 88.89% ,recall 100.00%

Category cereal_box ,precision 100.00% ,recall 75.00%

Category comb ,precision 100.00% ,recall 75.00%

Memory Management

Nº forgotten features: 0

Nº redundant features: 0

Nº forgotten features by Memory Decay: 0

Nº seen objects: 100

Nº seen features: 23864

Nº features in memory: 23864

…

Time Elapsed: 00:04:59

Some of the values outputted might only be used with the forgetting mechanism ‘on’ and some values are for debug purposes only.

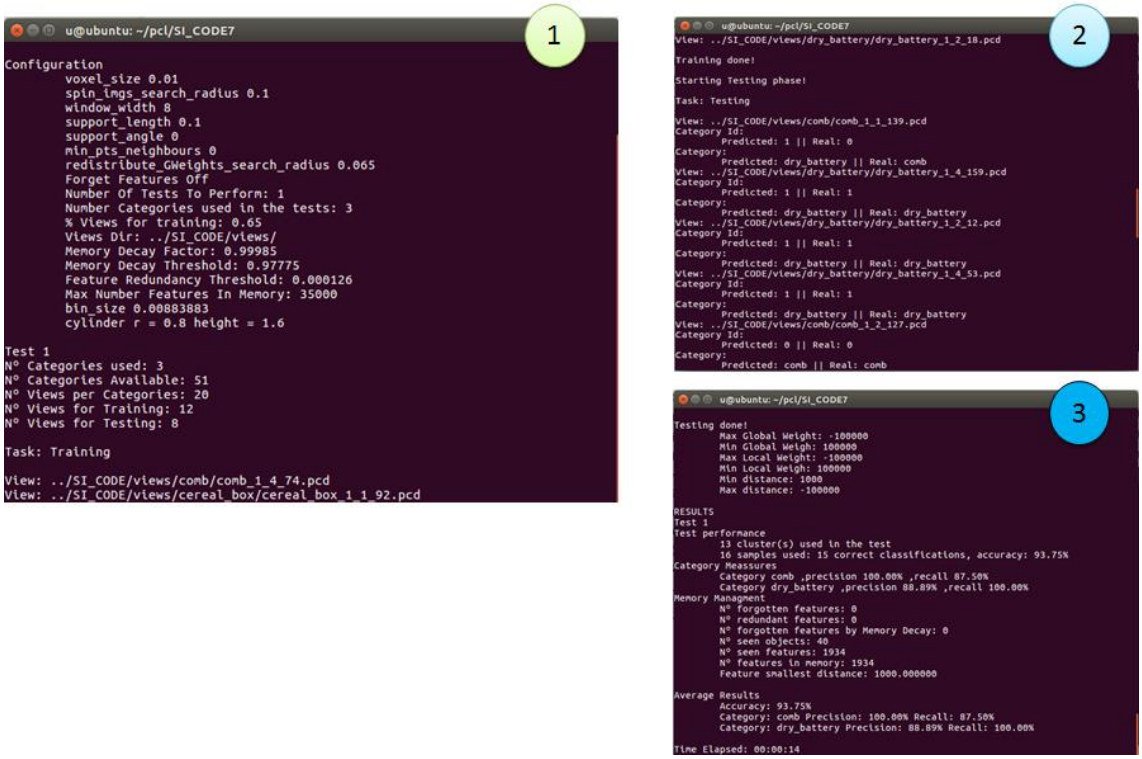

Fig 11: Main application. 1 - Settings and training phase. 2 - Test phase. 3 - Results..

5.2 Installing PCL, Application Compilation and Execution

Execute the following commands to install PCL on Ubuntu 16.04:

- sudo apt-get update
- sudo apt-get install libpcl-dev
- sudo apt install libproj-dev **(Use this command if you get an error related with libproj-dev.so during application compilation.)**

Run the following commands to compile and execute the applications:

Main application:

- cd SI_CODE
- cmake CMakeLists.txt
- make
- ./pcd_read views/ 20 65 true false 1 20 false config

Visualization application:

- cd SI_VIZ
- cmake CMakeLists.txt
- make
- ./viz /home/u/pcl/SI_CODE/views/apple/apple_1_1_1.pcd config

Interesting links related with some errors that might occur during this process:

http://stackoverflow.com/questions/38711172/pcl-point-cloud-library-1-7-on-ubuntu-16-04-lts-build-error

http://stackoverflow.com/questions/37369369/compiling-pcl-1-7-on-ubuntu-16-04-errors-in-cmake-generated-makefile

5.3 Documentation

The source code has been documented and we have generated the documentation for the main application. The documentation and the doxygen file to generate the documentation automatically can be found in the doc directory inside SI_CODE. To read the API just open the file SI_CODE/doc/html/index.html in a browser. There was a considerable effort to document the main functions of each class.

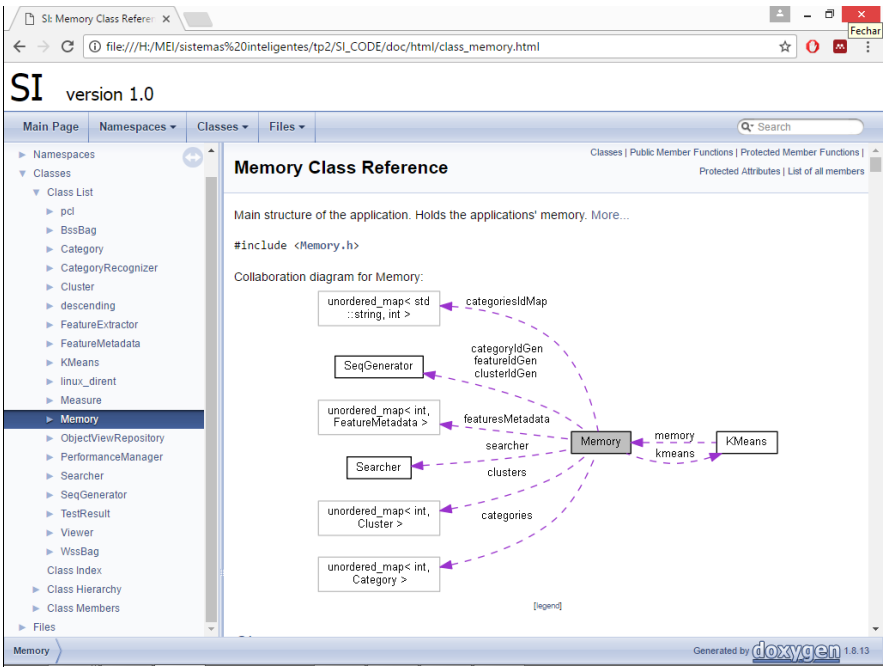

Fig 12: Documentation of the main application, found in the doc directory under SI_CODE.

5.4 Delivered Source Code

The delivered source code has the following structure.

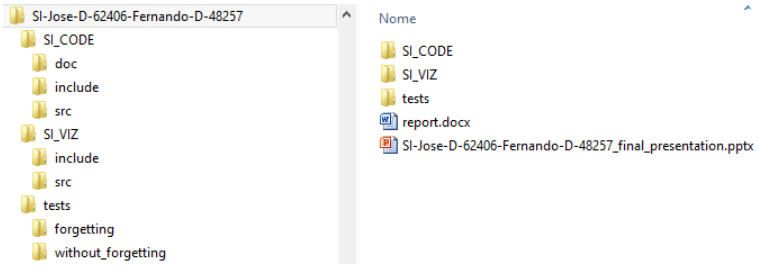

Fig 13: Source code directory structure.

Directory	Content
SI_CODE	The main application.
SI_CODE/doc	A file to generate the main application documentation, doxyfile. The generated documentation for the main application.
SI_VIZ	The visualization application.
tests	The files with the output of the tests performed and presented in this report.
tests/ without_forgetting	The tests with no feature forgetting.
tests/ forgetting	The tests with feature forgetting. • tests2 - The tests presented in this report. • tests1 - Tests performed on another machine at IEETA.
.	report.pdf - This report. final_presentation.pptx - Presentation.

6. Tests and Results

In this section we present the tests that were performed.

6.1 Tests Setup

	Characteristics	
The main application was tested using a VMware virtual machine in an external hard drive. VMware virtual machine setup:	RAM Nº Processors HD OS	4 Gb 1 Intel® Core™ i7-4510U CPU 2.00GHz 40 GB Ubuntu 16.04 LTS 32-bit
Host machine setup:	RAM Nº Processors HD OS	8 Gb 1 Intel® Core™ i7-4510U CPU 2.00GHz SSD Windows 8.1 64-bit
PCL	Version 1.7.2	

Main Parameters:

Nº views per category	20
Nº views for testing	8
Ratio training/testing	65%/35%
Redistribute Global Weights Search Radius (θ)	0.065
Memory Decay Factor (π)	0.99985
Feature Redundancy Threshold (ε)	0.000126
Memory Decay Threshold (δ)	0.97775

K-means Parameters:

maxCheckIters	1
wssThreshForSplit	0.12
minSplitClusterSize	50
degenerateClusterSize	20
wssThreshForMerge	0.07
maxSplitClusterSize	500
bssThreshForMerge	0.14

6.2 Tests Results

Nº Categories	Forget	Smashed Categories	Nº Clusters	Accuracy	Features In Memory	Nº Seen Objects	Execution Time
5	Off	None	21	90.00%	23864	100	04:59
10	Off	water_bottle	23	76.25%	41008	200	07:33
15	Off	water_bottle toothpaste	25	65.00%	56432	300	09:13
20	Off	Pliers water_bottle toothpaste	38	65.00%	65110	400	12:53
30	Off	water_bottle flashlight ball camera toothpaste	29	58.75%	119473	600	22:24
51	Off	Lot of smashed categories	50	40.20%	220136	1020	1:26:31

Nº Categories	Forget	Smashed Categories	Nº Clusters	Accuracy	Features In Memory	Nº Seen Objects	Execution Time
5	On	None	28	87.50%	M: 20271 F: 3593	100	28:19
10	On	water_bottle	79	72.50%	M: 33615 F: 7393 R: 3702 D: 3691	200	43:40
15	On		73	72.50%	M: 44664 F: 11768 R: 3862 D: 7906	300	2:05:41
20	On	pliers	108	68.12%	M: 48312 F: 16798 R: 3882 D: 12916	400	2:44:12
30	On	**These tests take too much time to perform.**					
51	On						

M: Number of features in memory.

F: Number of forgotten features.

R: Number of redundant features.

D: Number of forgotten features by applying the Memory Decay Threshold.

Smashed categories are categories with low precision and recall.

These tests are provided in the tests directory and the name of the files follows the value presented in the categories column in the table above.

6.3 Tests Results Discussion

Smashed categories are not necessarily categories with precision and recall equal to 0. And choosing them is somewhat subjective so the main conclusion about smashed categories is that its number tends to grow with the number of categories used in the tests.

The number of clusters seems to be somewhat independent from the number of categories used in the tests and the number of clusters is bigger when forgetting features. This is because when forgetting features the cluster centroids will change and this can induce splits and merges, since the number of clusters is dynamic, introducing more instability on the number of clusters.

The accuracy results seem to indicate that we can forget features and have results very similar to the results obtained when not forgetting features. It also seems that when forgetting features the accuracy results are slightly better compared to not forgetting features when using 15+ categories.

Execution time grows and performance drops significantly when using the forget features mechanism. This is due mainly because of the use of too many point cloud copies while checking features redundancy and forgetting features.

Execution time is also affected by the fact that the tests were performed using a virtual machine on an external hard drive. For instance:

Setup	Test Conditions	Execution Time
VMware virtual machine on an external hard drive.	20 categories forget features On	2:44:12
VMware virtual machine on the pc hard drive at IEETA.	20 categories forget features On	1:34:55

Note: These 2 tests are using the same 20 categories and object views with the same shuffle.

7. Performance and Optimization

Performance includes: memory management and k-means. Both can have a significant impact on performance.

At some point we realized that the application does not scale well and this is mainly due to the fact that we need to create a new point cloud to make knn and radius search operations using KdTreeFLANN. We wanted to have only one point cloud in memory and use that but:

- A point cloud in PCL is implemented using a std::vector<PointT>. Because the vector will be resized we can't have references to the features in the point cloud, references will be lost after a vector resize.
- Features will be added and deleted so the id of a feature can change over time, assuming the id of a feature = the features' position in the vector. Several classes in the system need to store some features ids. Therefore, some extra management is needed.

Also, we are using KdTreeFLANN but:
"pcl::KdTreeFLANN wraps flann::KDTreeSingleIndex which is optimized for low dimensional exact search and is not ideal for 153 dimensional descriptors. For high dimensional points you should be using an approximate nearest neighbor index from FLANN, such as flann::KDTreeIndex or flann::KMeansIndex."
Author: Marius Muja
Source: http://www.pcl-users.org/Kdtree-Flann-have-problem-search-nearest-neighbor-for-Spin-Image-Descriptor-i-e-Histogram-lt-153-gt-td4022815.html

We discussed several alternatives, but, because we already add an implemented infra-structure we maintained the option of having the features stored in the FeatureMetadata class and create a point cloud when needed. This does not scale well! Too many copies lead to poor performance!

A quick workaround, that does not solve the problem entirely, is to use a snapshot of all the features in memory. And this is the solution that the current version uses. Internally, KdTreeFLANN will create the index to use for knn and radius searches in the setInputCloud method. This means we could, e.g., collect all features in memory, create only one index and check all new features for redundancy using only that index and then use the same index for global weights redistribution. This means that only one point cloud is created, but with all features. This approach, however, has some dangers. Because all features are collected from memory, we need to avoid: redundancy by itself, i.e., a feature makes itself redundant and mutual redundancy, i.e., if a new feature, fA, is made redundant by another new feature, fB, and fB is made redundant by fA => fA and fB will be forgotten, but in this case, one of them should remain in memory!

Note: In this scneraio, even if fA is deleted from memory before redundancy for fB is checked, fA is still in the KdTreeFLANN index!

Another decision we made was to use KdTreeFLANN without sorting the results from radius and knn searches, to improve performance.

Finally, as mentioned before, the application uses snapshots whenever possible.

There are some alternatives to improve performance that we will discuss next.

- An alternative to KdTreeFLANN is to use KDTreeIndex like mentioned in [3]. We could also use the Euclidean distance directly.
- When checking feature redundancy we could use the radius search instead of the knn and collect all the neighbors to redistribute the global weights in case the feature is found redundant. This means that only one lookup is made in the index for a feature to check redundancy and to redistribute global weights. However, if feature redundancy is low, we are getting a lot of work up front. We could do this, of course, using snapshots.
- We could use spin images compression as in [1].
- We could use more efficient closest point search structures like in [2].
- We could change PCLs' source code to allow the use of point clouds of pointers to features in memory. This might be a non-trivial task.
- Finally, we could implement a infra-structure that has one point cloud and uses a lookup table to map the id of the features in the point cloud to a generated id that the classes in the system known about. This way, when a feature is deleted only the look up table needs to be updated. A diagram is shown below. This alternative can have some complications that are not immediately apparent.

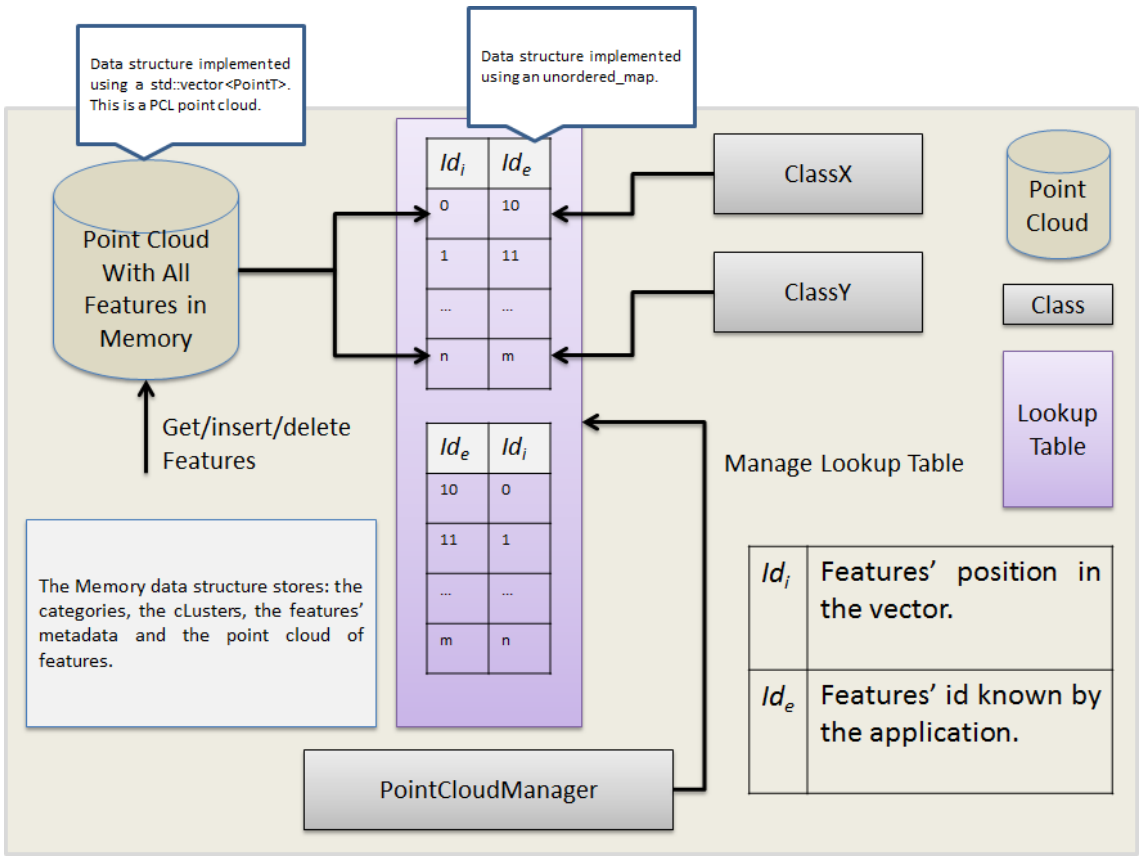

Fig 14: Alternative architecture to store the features in a point cloud in memory.

8. Issues and Future Work

- The application does not scale well! Optimizing the application is a major task for future work.
- Define appropriate values for the:
    - memory decay threshold,
    - memory decay factor,
    - feature redundancy threshold,
    - search radius for global weights redistribution,
    - split and merge thresholds.
    - etc.
- Further test the application with and without the forgetting features mechanism.

9. Conclusion

Further work can improve the results obtained and they suggest that the application can be used as a starting point for further investigation and experimentation, on the subject proposed for this assignment.

The implementation of K-means, in particular, could be further tested and fine-tuned to better understand its' performance and contribution in the results obtained. And the same can be said about memory management.

10. Bibliography and References

[1] Andrew E. Johnson, Martial Hebert (1998). Using Spin-Images for Efficient Object Recognition in Cluttered 3-D Scenes. IEEE Transactions on Pattern Analysis and Machine Intelligence.

[2] S. Nene and S Nayar. Closest point search in high dimensions. Proc. Computer Vision and Pattern Recognition (CVPR '96), 1996.

[3] Marius Muja, Julius Kammerl (2011). PCL Tutorial at RSS 2011 PCL::Search. http://www.pointclouds.org/media/rss2011.html.

[4] Miguel Oliveira et. al. 3D object perception and perceptual learning in the RACE project. Robotics and Autonomous Systems 75 (2016) 614–626. http://dx.doi.org/10.1016/j.robot.2015.09.019

[5] PCL Documentation: http://pointclouds.org/documentation/

[6] PCL API: http://docs.pointclouds.org/1.7.2/

[7] Washington RGB-D Object Dataset:
    - http://rgbd-dataset.cs.washington.edu/dataset/rgbd-dataset_pcd_ascii/
    - http://www.cs.washington.edu/rgbd-dataset/

[8] A Large-Scale Hierarchical Multi-View RGB-D Object Dataset Kevin Lai, Liefeng Bo, Xiaofeng Ren, and Dieter Fox IEEE International Conference on Robotics and Automation (ICRA), May 2011.

[9] L. Bo, X. Ren, and D. Fox. Unsupervised Feature Learning for RGB-D Based Object Recognition.

[10] Zhao Q., Xu M., Fränti P. (2009) Sum-of-Squares Based Cluster Validity Index and Significance Analysis. In: Kolehmainen M., Toivanen P., Beliczynski B. (eds) Adaptive and Natural Computing Algorithms. ICANNGA 2009. Lecture Notes in Computer Science, vol 5495. Springer, Berlin, Heidelberg