ESE 650 Project 6: Semantic Map Layering

Alex Baucom, Sakthivel Sivaraman, Sai Krishnan Chandrasekar

I. INTRODUCTION

As part of robot navigation, is it helpful to have a map of the environment to aid the robot in localization and path planning; however, it is also important to be able to assign labels or names to locations in the map (and vice versa) so that users or operators that interact with the robot can communicate effectively about where to go and what objects are in the map.

This work is partially motivated by the design and development of a low-cost, service robot [3] that is being done at Penn. Currently, we have a module that will allow a user to manually drive the robot around and label locations in a map, but it is quite cumbersome to get set up and only labels individual points in the map.

This paper presents the design of a framework to allow automatic, probabilistic semantic labeling and clustering of objects, regions, people, or anything else in layers of the map. We also present test results from synthetic data and real data from the robot platform.

II. PROBLEM FORMULATION

The goal was to make this framework as flexible and extensible as possible and so we did not make any explicit assumptions on the use case for this labeling framework, only that there are some 'black box' detectors that give asynchronous streams of noisy observations consisting of a class label and an xy coordinate location in the map.

Using this incoming data we would like to estimate and maintain a model of the true locations of all observations and provide ways to make meaningful queries and inferences about the data.

For validating the framework on the real robot we used an off-the-shelf object recognition algorithm but it is easily extensible to any other detector. Other examples of possible use cases are maintaining locations of people by combining this with a face recognition algorithm or automatically classifying regions of the map with a scene recognition module.

III. TECHNICAL APPROACH

A. IGMM

After researching several other methods, the Incremental Gaussian Mixture Model [1] [2] seemed to fit our requirements for an internal model the best. It could maintain a GMM with incremental updates and add and remove clusters as necessary for the data. The IGMM maintains a standard mixture model representation

$$p(\mathbf{x}) = \sum_{j=1}^{M} p(\mathbf{x}|j)p(j) \tag{1}$$

where $p(j)$ is the prior, or weight, of the jth cluster and $p(\mathbf{x}|j)$ is the Gaussian probability that $\mathbf{x}$ belongs to cluster j, which is defined as:

$$p(\mathbf{x}|j) = \frac{1}{(2\pi)^{D/2}\sqrt{|\mathbf{C}_j|}} exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_j)^T\mathbf{C}_j^{-1}(\mathbf{x}-\boldsymbol{\mu}_j)\right) \tag{2}$$

where D is the dimensionality of the data, $\boldsymbol{\mu}_j$ is the mean and $\mathbf{C}_j$ is the covariance matrix of the jth cluster.

Then a *novelty criterion* is defined as:

$$p(\mathbf{x}|j) < \frac{\tau_{nov}}{(2\pi)^{D/2}\sqrt{|\mathbf{C}_j|}} \forall j \tag{3}$$

where τ_{nov} is a user defined fraction that determines how novel a new data point must be to warrant creating a new cluster. If the *novelty criterion* is met, then a new cluster is created and if it is not met, then an update step is performed. The full update step and component creation equations are presented in Section 2 of [2].

A key feature to note of this algorithm is that, in addition to maintaining the standard GMM values of $\boldsymbol{\mu}_j$, $\mathbf{C}_j$, and $p(j)$ for all j, two other values v_j and sp_j are maintained and updated for each cluster. The value of v_j keeps track of how many updates have occurred since the cluster j was created and the value of sp_j is a running sum of $p(j|\mathbf{x})$. In essence, v_j measures how long the cluster has existed and sp_j measures how important the cluster is. These values are used for both updating the clusters during the update step and pruning out clusters that are not important. A cluster is pruned when $v_j > v_{min}$ and $sp_j < sp_{min}$ where v_{min} and sp_{min} are user defined values.

B. Layering

For our purposes, a *layer* is defined as a collection of IGMMs which are all updated from the same observation stream. When a new observation is received, its label is compared to previously observed labels. If this label has previously been seen, the observation data is added to the corresponding IGMM as a new data point which will either trigger an update or new cluster for that model. If this label has not previously been seen, a new IGMM is created and initialized with the given observation data.

This is a fairly simple layer implementation as each class is treated as essentially separate. It could be possible to do cross-class updates to address noise in the class labels or remove intersecting clusters but that is not currently implemented. See the Discussion and Future Work sections for more details.

In order to integrate well with ROS, we wrote a wrapper for the layer code that allows any number of layers to be created, observation updates to occur via ROS topics, and parameters to be set via the ROS parameter server. We also implemented lookup requests as a ROS service (see next section). One point to note is that the layer and IGMM framework make no assumptions about the dimensionality of the observations - only that observations with the same label must have the same dimension. It is only the ROS wrapper that assumes a two dimensional coordinate in the "map" frame. Extending the IGMM algorithm to high dimensions can be very computationally expensive since the covariance matrix must be inverted with every update, but the authors discuss a faster version of the IGMM (and provide a very clear and concise explanation of the standard IGMM) in [2].

C. Lookup Requests

While the IGMM and layering system maintains a model of the observations, we would also like to extract specific information for planning, navigation, or other purposes. For this reason, we created a ROS service that allows a user or a different part of the code to request specific information about a particular layer or class.

We have currently implemented two types of lookup requests that use the same service. The first will take a layer name and a class name and return the *most likely* location for that class (which will likely be the cluster that has had the most observations added to it). The second will take a layer name, a class name (or 'any'), and a pose and return the *closest* location (using standard Euclidean distance) of the specified class (or of any class) to the given pose.

There are many other types of lookup or inferences that could be made and returned and the hope would be that more could be added in the future but we started with a couple simple ones for now as a proof of concept. See the Discussion and Future Work sections for more details.

D. Object Detection

In order to validate our framework of automatic, probabilistic semantic labeling and clustering of objects, regions etc., we wanted to use an off-the-shelf object recognition algorithm. After testing several algorithms, the Faster-RCNN model seemed to fit our requirements.

Recent advances in object detection are driven by the success of region proposal methods and region-based convolutional neural networks (RCNNs). Most object detection algorithms have a 2 stage process: Region proposals and Classification which are both independent of each other. They usually use an external regional proposal algorithm like grouping of super pixels, sliding windows etc. Here , the region proposal step still consumes as much running time as the detection network as they run on CPU.

Faster-RCNN eliminates these problems [4]. It computes proposals with a deep convolutional neural network leading to an elegant and effective solution where proposal computation is nearly cost-free given the detection networks computation. It uses a Region Proposal Networks (RPNs) that share convolutional layers with state-of-the-art object detection networks. By sharing convolutions at test-time, the marginal cost for computing proposals is small. RPNs are designed to efficiently predict region proposals with a wide range of scales and aspect ratios. Both RPN and the Object detection network can be trained end-to-end with shared features. Also, it does object detection in a single forward pass making it really fast and near real-time.

Faster R-CNN, is composed of two modules as mentioned before. The first module is a deep fully convolutional network that proposes regions, and the second module is the Fast R-CNN detector that uses the proposed regions. The entire system is a single, unified network for object detection (Figure 1). The RPN module tells the Fast R-CNN module where to look.

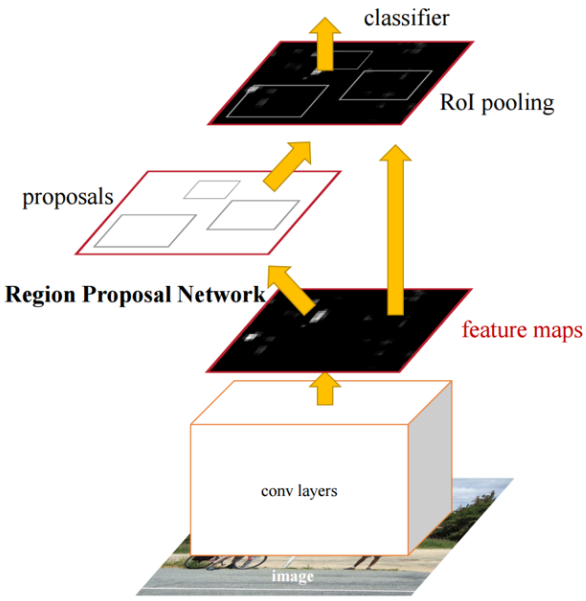

Fig. 1. Faster-RCNN framework

We integrated this algorithm with the turtle-bot and the map layering framework to be able to perform real-time detection, labeling and search requests. The Turtle-bot uses an Intel-NUC i5 processor for localization. Hence, we opted to set-up the Faster-RCNN on a ASUS Strix Laptop with NVIDIA GeForce GTX 1070. We built a ROS Wrapper for the Faster-RCNN that allowed us to receive the live rectified depth and rectified RGB image stream from the Kinect (connected to the Intel NUC) via Wi-Fi. The results from the object detection and the depth image are used to estimate the 3D coordinates of the object in Kinect frame. The robot's current pose is derived from the localization algorithm and used to transform the detected object's coordinates in Kinect frame to World frame (MAP coordinates). This, along with the labels are published as ROS messages which is used by the Map Layering Framework for labeling/clustering.

E. Path Planning

The overall flow of the framework is as follows: Detect objects using a detector, cluster them together using an IGMM and use lookup requests to allow the user to interact with the framework. The two lookup requests we have implemented as of now: The *most likely* location for a class and the *closest* location for a class. Once the framework returns the *most likely* location or the *closest* location, we wanted the robot to actually visit that location. For these simple lookup requests, the path planning problem isn't too complicated. An A* based path planner could be used for this problem.

However, we want our framework to be able to handle more complicated lookup requests. Ideally, if the object detector can detect people, including facial recognition, our lookup requests might involve people as well. For such requests, the path planning algorithm would have to be more optimal. For example, a complex lookup request could be something like: "Where is John Doe?". In this case, the planner would have to do the following tasks:

- Find the locations where the class "John Doe" has been observed.
- Identify the current location of the robot.
- Pull out the probabilities of observing the class "John Doe" at those locations.
- Estimate an optimal path that accumulates as high a cumulative probability as it can at each instant while minimizing the path length.

This was implemented by modeling the problem as a graph based Markov Decision Process.

1) Markov Decision Process: A Markov Decision Process (MDP) is a markov chain whose transitions are controlled. Thus, the current states depend only on the previous state and the action applied on the previous state.

More precisely, an MDP is a tuple $(\mathcal{X}, \mathcal{U}, p_a, r, \gamma)$ where:

- $\mathcal{X}$ is a fully observed state space (i.e., is known) is a control/action space
- $\mathcal{U}$ is a control/action space
- $p_a(\mathcal{X}_{t+1}|\mathcal{X}_t, \mathcal{U}_t)$, is a Markov transition kernel, specified by the motion model
- $r(\mathcal{X}_t, \mathcal{U}_t)$, is a reward/cost function, whose value is received after control is applied in state
- $\gamma \in [0, 1]$ is a discount factor

2) Reward: Reward is simply a scalar value that the agent receives for being in that particular state. Rewards are instantaneous and are confined to that particular state. In our world, the rewards are only non negative. The agent receives a reward if it reaches the goal. This reward is a weighted sum of 2 factors:

- A product between a maximum achievable reward and the probability associated with that location
- A product between the path length from the previous state to the current state and a negative factor.

The goal of this problem is to maximize the reward.

3) Policy: A solution to an MDP is a policy. Policy is defined as a mapping from a state $\mathcal{X}$ to an action $\mathbf{A}$. In other words, a policy tells the agent what action to take at a given state. Denoted by π.

The optimal policy π^*, denotes the policy that maximizes the long term expected reward.

4) Discount Factor: The discount factor is introduced to solve the infinite horizon problem with a finite reward. Denoted by γ. If $\gamma = 0$, the agent is myopic, i.e., maximizes only immediate rewards. If $\gamma < 1$ and $R(S, A)$ is bounded, then the infinite horizon problem has a finite reward.

5) Graph based environment: The map was modeled as a graph. In our case, the map of Levine 4th floor was modeled as shown in figure 3.

Each location where the object had been observed was mapped to a nearest node using euclidean distance. These nodes were termed as nodes of interest. From the graph, a subgraph of these nodes was computed. The subgraph's edges were estimated using the connectedness of the nodes in the original graph. In other words, if two nodes had a path between them in the original graph, they were connected using an edge in the subgraph. A sample subgraph is shown in figure 4.

The path lengths for these nodes were computed using information we knew about the real world and from the adjacency matrix.

From the aforementioned information, a reward matrix was computed using the following formula:

$$r[cs, ns] = maxr * prob[ns] - length[cs, ns] * factor$$

where:

- $r[cs, ns]$ is the reward for moving from current state (cs) to next state(ns).
- $length[cs, ns]$ is the path length of moving from current state (cs) to next state(ns).
- $maxr$ is the maximum reward possible. We set this to be 10.
- $prob[ns]$ is the probability of the object associated with the node, ns.
- $factor$ is the penalty applied on the path length.

This reward matrix was used to identify the rewards of going to the next state from the current state.

6) Policy Iteration: The Bellman equation in its original form is non linear due to the max function in the equation. V is the value function.

$$V(\mathcal{X}) = r(\mathcal{X}) + \gamma \max_a \sum_{\mathcal{X}'} (p_a(\mathcal{X}, A, \mathcal{X}')\mathcal{U}(\mathcal{X}')$$

Even though we have n unknowns and n equations, it's not possible to solve it in linear time, which is why we do a value update step until convergence. There is however, a

faster way to get to the optimal policy π^*. That method is policy iteration.

Let's take a look at the modified equation:

$$V(\mathcal{X}) = r(\mathcal{X}) + \gamma \sum_{\mathcal{X}'} p_a(\mathcal{X}, \pi_t(\mathcal{X}), \mathcal{X}') V_t(\mathcal{X}')$$

By changing the transition function from $p_a(\mathcal{X}, A, \mathcal{X}')$ to $p_a(\mathcal{X}, \pi_t(\mathcal{X}), \mathcal{X}')$, we eliminate the need for the max function, thus eliminating the non linearity. This new equation is now just a linear system of n equations and n unknowns in the form of $Ax = B$, with x being our policy pi. Thus, we compute the state-value function for the policy π.

It should be noted that if the same node is visited more than once, the cumulative probability doesn't change. The path length, however, does.

By policy iteration, the optimal policy for each state $\mathcal{X}$ given the reward matrix was computed.

IV. RESULTS

A. Synthetic Data

First, we ran several tests with synthetic data. We generated the data points by taking a list of 'true' objects and regions with their associated labels and creating several noisy observations of each 'true' data point by adding Gaussian noise. We also randomly shuffled the observation order in some cases. Figure 2 shows a plot of an IGMM with non-map-based data and Figure 3 shows observations from two layers from map-based data. Animations of both the non-map-based [1] and map-based [2] data are available online.

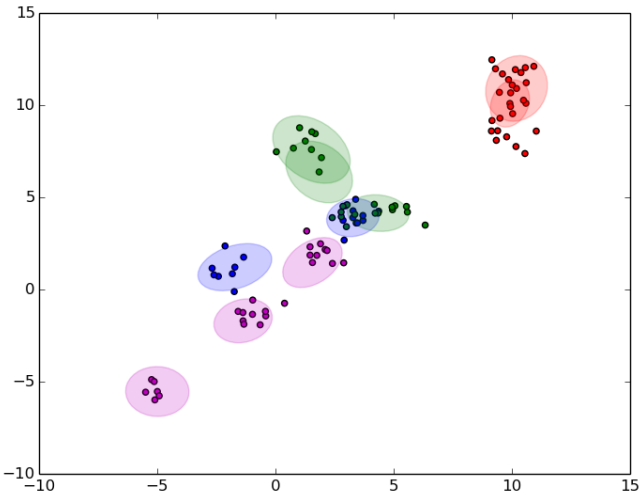

Fig. 2. Sample plot of IGMM with synthetic data augmented by Gaussian noise. Each class is represented by a different color.

[1] https://youtu.be/XGyHTZ6_YnM
[2] https://youtu.be/Nj-dLVJAze0

Fig. 3. Sample plot of two layers with map-based synthetic data augmented by Gaussian noise. Door observations from the object layer are shown and hallway observations from the regions layer are shown. Both are marked with red but the text near the markers denotes the layer and class.

B. Real Data from the Robot

Once the algorithm was verified to be working with synthetic data, we tested on the real robot using an object detector and a Microsoft Kinect.

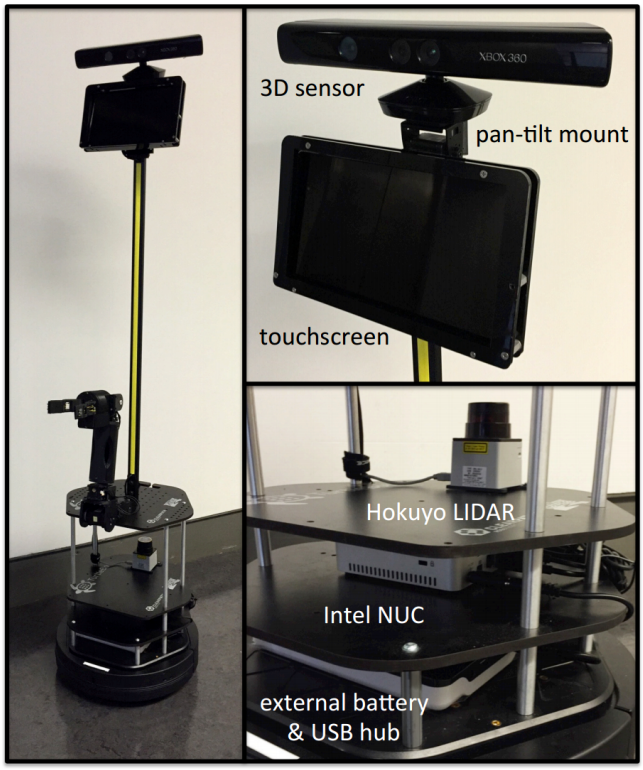

Fig. 4. Robot platform - TurtleBot base with Kinect v2.0

A video of both the camera footage with detections and the labeled map is available online [3].

[3] https://youtu.be/7h8EoGzfU4w

Fig. 5. Real-time Object detection and Map Labeling

C. Path Planning

- The map of Levine 4th Floor is shown in figure 3. The graph is representation of the map is superimposed on it.

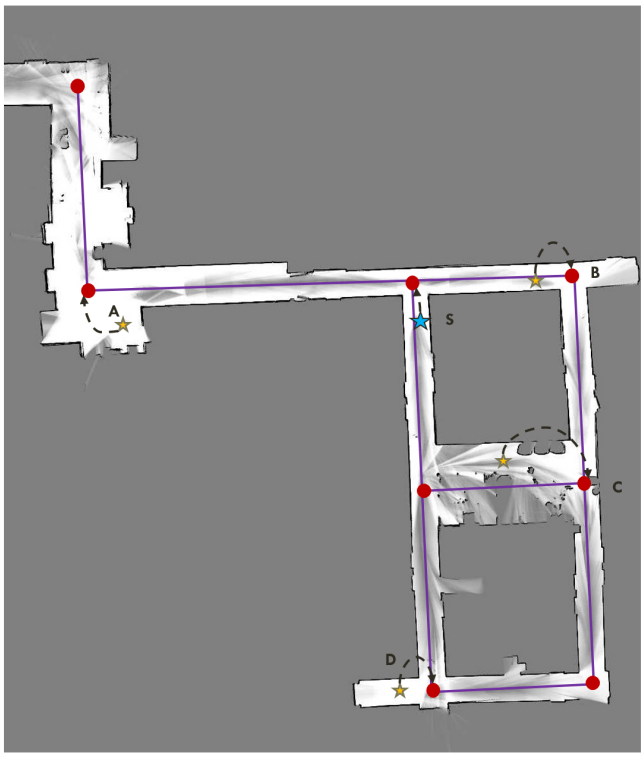

Fig. 6. The map of Levine 4th Floor represented as a Graph.

- The legend for the map is shown 4: The red dots are nodes in the graph representation of the map. The yellow stars represent the centers of the class clusters. The dotted curved arrows show the mapping of each cluster's center to the nearest node. The blue star represents the start node of the robot.

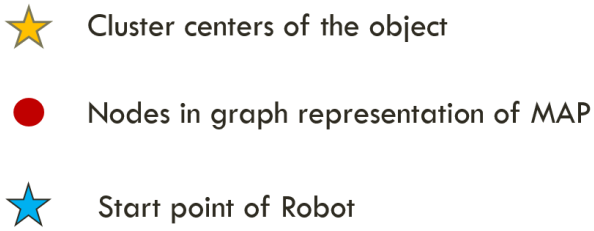

Fig. 7. Legend for figure 3.

- The subgraph from the sampled nodes is shown in figure 5. The numbers show the probability of the object being at that node. The thicker the arrow, the more the cost of traveling from the previous node to that node.

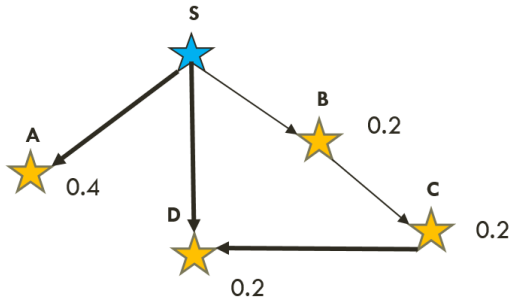

Fig. 8. The subgraph from the sampled nodes.

- The output of our path planning algorithm is shown below. This algorithm takes both the cumulative probability and cumulative path length into account. The path returned is shown using dotted arrows. The order in which the nodes are traversed is shown as well.

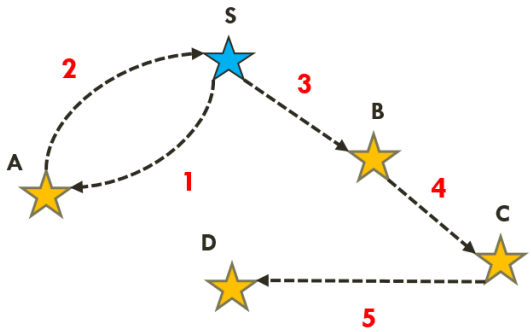

Fig. 9. The optimal path given by our planning algorithm.

V. DISCUSSION

A. IGMM

As seen in the Results section, the IGMM clustering algorithm works well for clustering observations online and the layer implementation allows for observations of different classes to easily be added to the model. However, the algorithm struggles to deal with the data association problem since it is so dependent on setting the *novelty criterion* correctly. This is particularly a problem if the expected noise in the observation is close to the expected distance between clusters. For example, if a object detector can identify chair locations with $\pm 0.5m$ accuracy, the IGMM model might not be able to distinguish two chairs sitting right next to each other if the *novelty criterion* is set fairly low, since observations have to be more unique to create a new cluster. Alternatively, if the *novelty criterion* is set higher, meaning that observations don't need to be as unique to create a new cluster, two observations of the same chair might be classified as coming from different clusters.

The IGMM algorithm is also sensitive to the parameters regarding cluster removal and the order in which the observations are received. Since the cluster removal occurs after a specified number of updates to the model, it is possible (and we saw it fairly often) to have a cluster created with a single observation, then a few updates occur that don't affect the new cluster so it is removed, only to have more observations in the future added to that same spot. In the limit of infinite observations, this isn't an issue, since there will eventually be enough observations consecutively to keep the cluster around, but with finite observations this sometimes created an area where there clearly should have been a cluster, but the data hadn't been observed in an order which preserved the cluster. This can be seen in red data points in Figure 2. Additionally, the opposite problem sometimes happened (which can also be seen in the red data points in Figure 2) where two clusters ended up very close to each other due to the order the data was observed in.

When testing on the real robot, it was clear that the pipeline from object detection all the way to map labeling worked great. However, the off-the-shelf object detection algorithm was very sensitive and detected a lot of false positives and sometimes even reported a single person as multiple people. In addition to the detection sensitivity, the depth estimation from the Kinect was just a rough estimate and would often lead to labels being placed in the wrong section of the map, or even off the map entirely. This noise also caused many spurious clusters to be created since detections of the same object would often be marked as coming from different locations.

Some proposed solutions to these problems are as follows. First, more accurate detection and location estimates will help in almost every aspect. If the detection noise is lower than the distance between observations of distinct objects, then the parameters can be tuned easily to maintain accurate and separate clusters for each object. If the detection noise cannot be reduced, then much more parameter tuning will likely be required for the system to perform accurately in its current state.

Second, some adjustments to the IGMM algorithm might also help resolve these issues. One possibility would be to handle cluster removal in a batch pruning step as opposed to an online, individual pruning step. So, instead of only looking at a single cluster and the v_{min} and sp_{min} values to determine if the cluster should be removed, the algorithm could consider all clusters and their relation to one another. If there are multiple clusters all very close together, they could possibly be combined into one larger cluster and if there are clusters that both have a low sp_{min} value *and* are distant from any other clusters, they could possibly be removed. Another possible improvement would be to add support for multi-detection observations. For example, if an object detector recognizes two distinct objects in the same image, these could be presented to the IGMM as a multi-detection observation. In this case, if both of these observations would normally be assigned to the same cluster, this could instead force the single cluster to split into two smaller clusters. If instead there were already two clusters that these observations would be assigned to, this would just further reinforce that those clusters are indeed representing distinct objects and should be kept separate.

B. Object Detection

Based on our research, YOLO was our first choice for Object detection. But due to it's compatibility issues with ROS (YOLO requires tensorflow and python3), we weren't able to integrate it into our model. Our attempts at installing ROS and other dependencies with python3 in a virtual environment were unsuccessful. But as future work, we would like to integrate the YOLO algorithm with our system as it has various advantages over the Faster-RCNN. Below is a brief discussion of its advantages over the other algorithms.

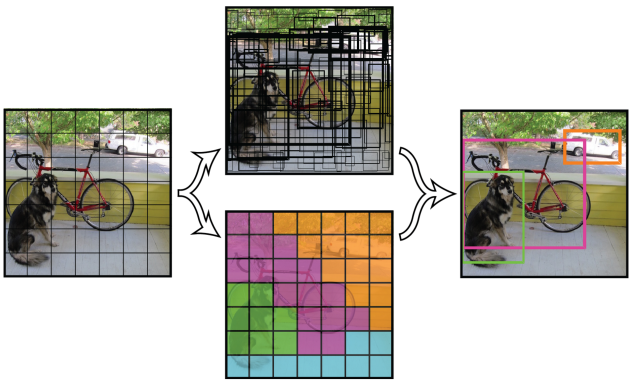

Fig. 10. YOLO: Real-Time Object Detection.

Most detection systems re-purpose classifiers or localizers to perform detection. They apply the model to an image at multiple locations and scales. High scoring regions of the image are considered detections. The YOLO algorithm uses different approach. It applies a single neural network to the full image which divides the image into regions and predicts

bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities [5].

It looks at the whole image at test time so its predictions are informed by global context in the image. It also makes predictions with a single network evaluation unlike systems like R-CNN which require thousands for a single image. This makes it extremely fast, more than 1000x faster than R-CNN and 100x faster than Fast R-CNN. It's processing speed on a NVIDIA GeForce GTX 1070 was about 15-20 fps. It performed better than Faster-RCNN on indoor objects.

Another advantage of YOLO is that the detection system could be set-up on the Intel-NUC as it runs at about 1 fps even on a CPU. This would eliminate the requirement of an additional GPU and it also goes well with our objective of building a low-cost service robot.

While the algorithm works well, it also gives a lot of false positives. To some extent, it can be eliminated by setting high confidence thresholds and by the IGMM based clustering algorithm which can get rid of some noise. But we would need a way to eliminate false detections as we want to make the whole process of map labeling completely automatic. This could be done by taking into account the object to object relations and their location. Using a scene recognizer would allow elimination of objects with low probability of co-existence in the given scene. As discussed before, this requires drawing layer-to-layer inferences.

C. Path Planning

An alternative method was initially implemented to solve the path planning problem. The whole problem was solved using graph search, where the path depended on the *Next nearest node*. The *Next nearest node* was decided using a cost function:

$$\mathcal{C} = a * length + b * (1 - \sum probability)$$

where $\mathcal{C}$ is the cost. a and b are parameters.

This cost function might have been overly simple to capture the complexity of the problem. This approach failed on quite a few edge cases. A few of them are listed below:

- Complete traversal of one branch of the tree, while leaving the other branch(es) untraveled.
- Repeated traversals over the same node.
- Nodes that were *equally near*

There were other edge cases that occurred. We felt there were too many edges to handle, which was why we implemented the reinforcement learning based path planner.

With regards to the current method, the approach is quite similar to solving a frozen lake using MDP, so there weren't too many unforeseen issues. Care had to be taken to ensure two things:

- Modify the probability once a node had been visited. We had to ensure that the agent would receive no more cumulative probability by repeatedly traversing over a node.
- All nodes would have to be visited.

This algorithm is a complete algorithm. If the object exists in the area defined by the map, the robot will find it. If not, the robot will report that the object does not exist in the area.

We also updated the overall probabilities of the map over each "search" too. If the robot was/wasn't seen on each iteration at a given node, the probability of that object being in that node would change. This way, this serves as a continuous learning approach as well.

VI. CONCLUSION AND FUTURE WORK

In this paper we presented an extensible framework for creating layers of semantic labels that can be grounded in the map coordinate system with Incremental Gaussian Mixture Models. We showed the results of both synthetic and real-world data and discussed some of the pros and cons of this approach as well as possible solutions to some of the problems with the IGMM algorithm.

We primarily focused on developing and validating the framework of the system and there are still plenty of extensions, improvements and updates that could be made to this system to make it more user-friendly, robust, and accurate. We mentioned some possible improvements in the Discussion section. A few other possible extensions to the system are: enabling cross-class updates to address issues with class label noise, designing and implementing more interesting or useful inference and lookup requests, taking the structure of the map (i.e. free space, walls, doors) into account when adding observations, or simply applying this system to other types of observation and detection algorithms.

In terms of optimal path planning, modifying the environment to accomodate dynamically changing probabilities *during* each iteration seems like an interesting problem. Right now, we're restricting our reinforcement algorithm to the subgraph. However, if the object is seen in a new node *during* search, our algorithm will not handle that, especially if the robot cannot see the object during the search. One possible solution to this algorithm would be to initially check the subgraph. If the object is not there in the subgraph, the robot should visit all other nodes in the graph, i.e., non nodes of interest. This can be achieved by using pure waypoint navigation for all non nodes of interest. This would make the algorithm a true complete algorithm. This seems like another interesting idea to explore.

While we might not get the chance to work on all of these improvements we welcome and encourage anyone else who is interested to work on them. All of the code, along with detailed installation and usage instructions is available on GitHub [4].

REFERENCES

[1] Engel, Paulo Martins, and Milton Roberto Heinen. "Incremental learning of multivariate gaussian mixture models." Brazilian Symposium on Artificial Intelligence. Springer Berlin Heidelberg, 2010.

[2] Pinto, Rafael Coimbra, and Paulo Martins Engel. "A fast incremental gaussian mixture model." PloS one 10.10 (2015): e0139931.

[3] Eaton, Eric, et al. "Design of a Low-Cost Platform for Autonomous Mobile Service Robots."

[4] https://github.com/alexbaucom17/MapLayer

[4] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. ”Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.”
[5] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi. ”You Only Look Once: Unified, Real-Time Object Detection.”