CS469 Assignment 2

SVM for Landmark Observability Prediction

Author: Conor Hayes

Dataset: DS1 - http://asrl.utias.utoronto.ca/datasets/mrclam/index.html

Part A

A1 - Describe the reasoning behind your learning aim. Include any diagrams or plots to support your reasoning.

In this study, we attempt to learn a predictive model for landmark visibility given robot state. Formally, we wish to train a classifier on **(X, Y)** that takes robot state **x** and predicts observation $\mathbf{y_{hat}}$, where **x := (x y θ)** , **y :=** $\mathbf{(y_{L1}\ y_{L2}\ \ldots\ y_{LN})}$, and $\mathbf{y_{Ln}} \in$ **{-1,1}** is 1 if landmark *n* is visible and -1 otherwise.

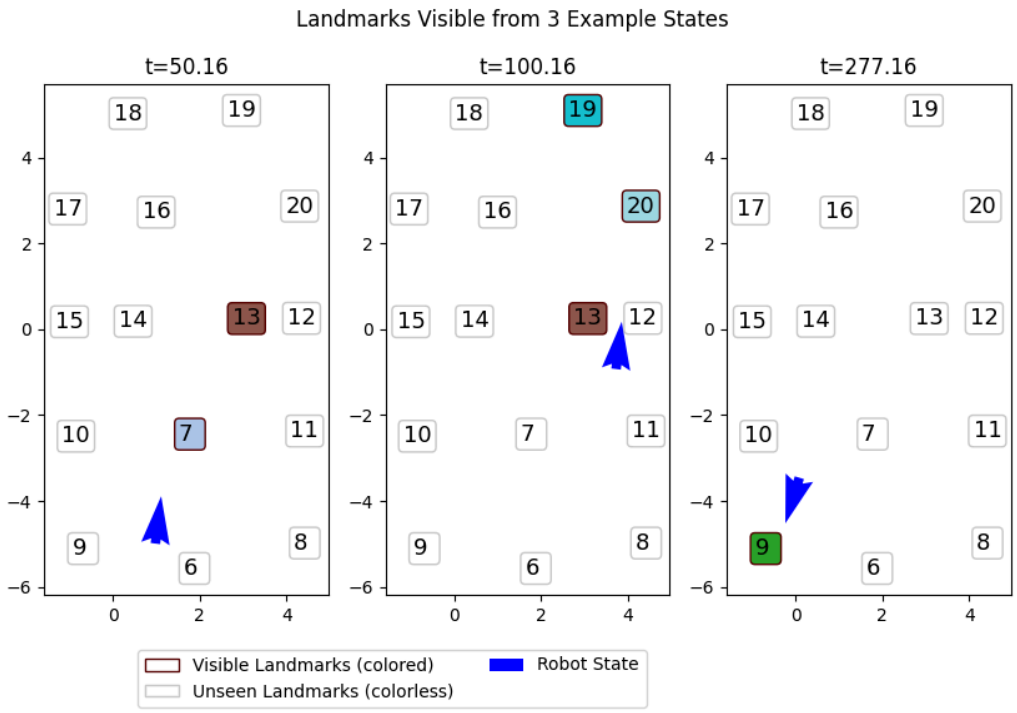


Fig A1.1 - Landmarks visible from 3 sample robot states selected from the ground truth robot dataset. Visibility at *t* is determined by whether the landmark is observed in *ds0_Measurement.dat* during the 2-second window prior to *t*. Intuitively, landmark visibility should be correlated with robot field of view and proximity, i.e. a function of orientation and position, in these examples; however, there are some surprises in each (14 is not visible in the left view, nor is 12 in the middle view), perhaps due to occlusion or rangefinder properties.

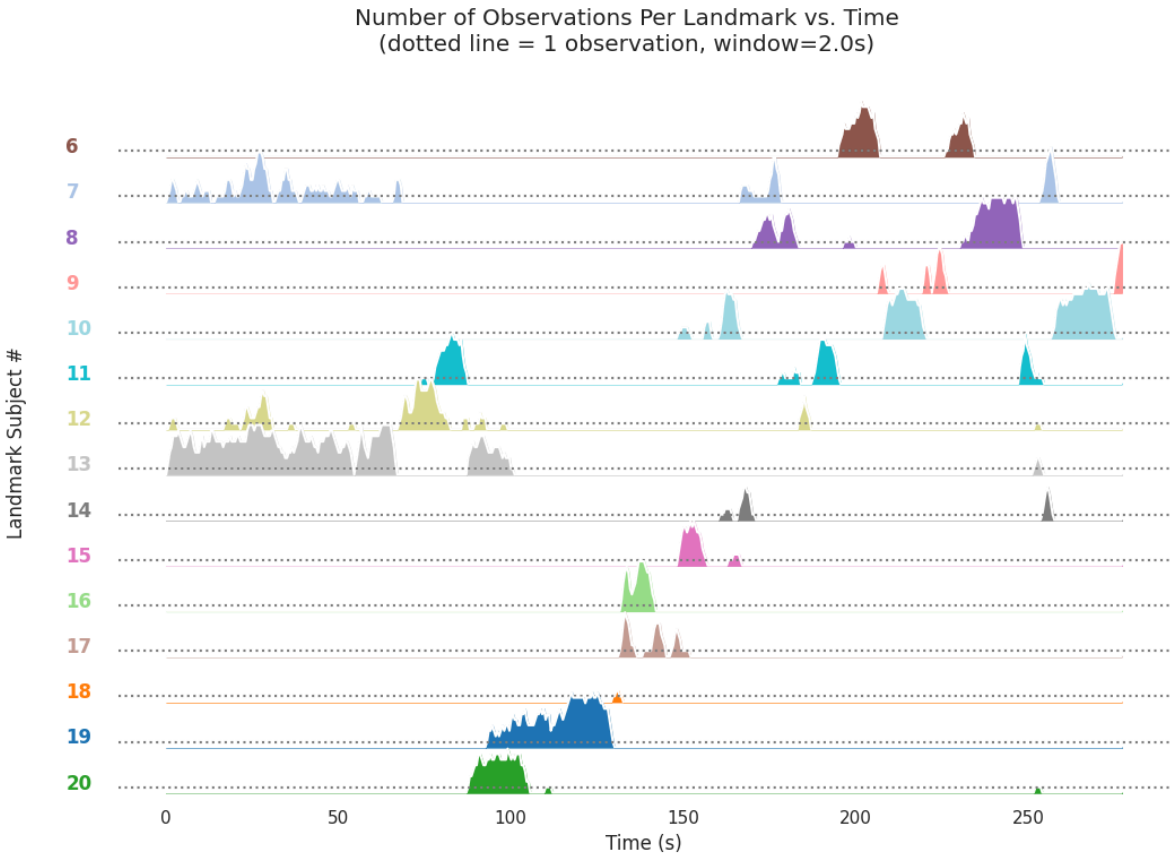


Fig A1.2 - Landmark visibility over time for the ground truth robot trajectory, using the same 2-second sliding window approach (the colors assigned to each landmark are also the same in all other figures, including Fig A1.1). In this plot, rather than simple binary classes, we use the *number of observations* we have of a landmark during the window as the y-axis of each landmark's plot. We can once observe clear structure in the data as the robot travels, indicating that predicting landmark visibility may be a viable target for a learning algorithm.

The tractability of this goal was first assessed qualitatively using DS1's ground-truth and measurement data (see Fig A1.1 and A1.2 above), indicating that there is structure that a successful learning algorithm may be able to take advantage of.
A landmark visibility predictor might be useful as a prior for a localization algorithm (such as the Bayesian filters explored in Assignment 0), or any other algorithms attempting to exploit onboard sensor data (i.e. live video) for scene understanding.

A2 - Justify (with data) why you expect this algorithm will be able to accomplish your learning aim.

We attempt to solve this classification task using a Support Vector Machine (SVM). In its most basic form, SVM is designed for simple binary classification on linearly separable data, but is often extended to handle data which is not linearly separable (via slack variables) or which may be separable by more complex surfaces in the input space (via the kernel trick), as well as univariate regression.

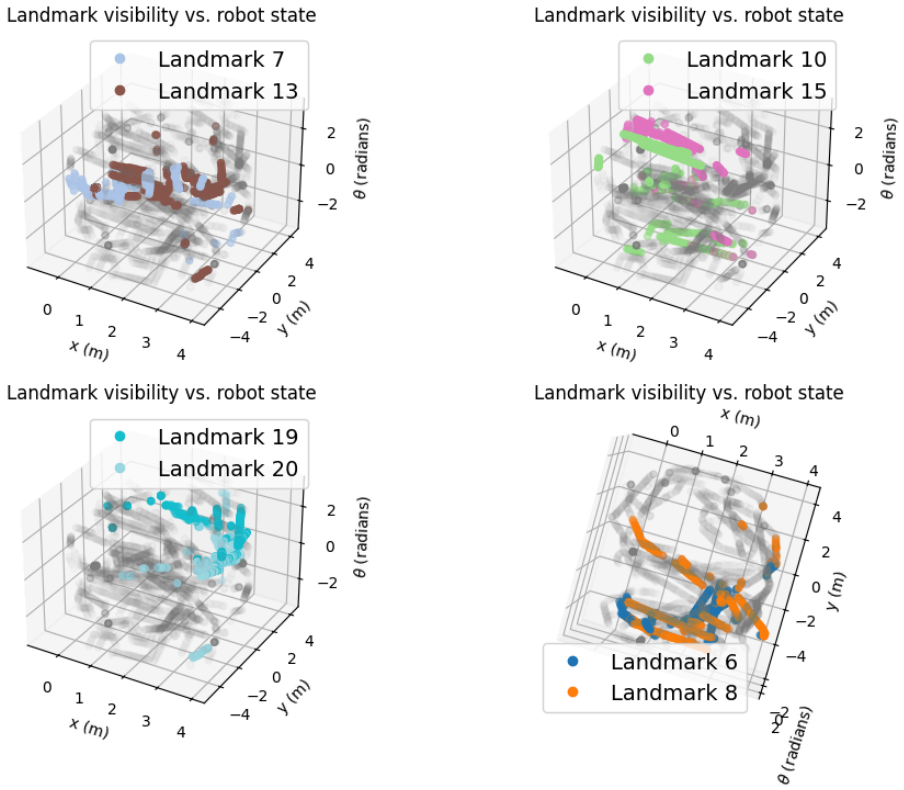


Fig. A2.1 - Visualization of the state space **x := (x y θ)** in 3D, with output labels $\mathbf{y_{Ln}}$ represented in color. Colored dots represent robot poses for which a given landmark is visible, while grey dots represent poses in which no landmark listed in the legend is visible. Each plot considers only 2 landmarks at a time for visual clarity. Seen from above, as in the bottom right, we can roughly see the trajectory of the robot on the xy plane from ds1, though the time element is removed here. We can also see that each landmark's visibility area partitions the state space into fairly distinct regions, once more indicating training an SVM classifier is possible. However, these regions don't look separable by any single plane in the input space, indicating a nonlinear kernel will likely be needed.

SVM's excel at learning with generality even on high-dimensional datasets with few training samples, where other techniques, such as MLP's, typically struggle. They are also considerably

easier to reason about and are generally more interpretable than neural networks. Their primary limitations are their computational complexity (training is bottlenecked by the need to solve a quadratic programming problem, which can be up to **O(**n^3**)** for n training samples[2]), and the fact that a single SVM can only perform binary classification or regression with univariate output; multi-class classification in which the outputs are correlated, or regression with higher-dimensional output, require groups of SVM's to be used. However, in the case of this learning objective, which can be formulated as a set of independent binary classification problems (see Section A3 below), none of these drawbacks present a challenge, while SVM's strong performance on small datasets provides a significant advantage, indicating that SVM may be a good fit for our learning objective.

A3 - Frame your learning problem. Define and describe your problem space. Make sure to clearly define any parameters, and how you will tune their values. As appropriate to your algorithm, define all necessary functions, and describe and generate any datasets.

We train a set of N classifiers given N landmarks, each with the inputs & outputs described below:

- Input: robot state **x := (x y sinθ, cosθ)**
- Output: robot visibility $\mathbf{y_{Ln}} \in$ **{-1,1} | 1** if landmark *n* is visible, **-1** otherwise.

Note also that instead of feeding in raw orientation **θ**, we use the pair **(sinθ, cosθ)**. This is to avoid the angle-wrapping problem–to ensure that points which are physically close in angle are also close in the input space, which may help it cover those points together in the same class.
To generate the training data, a sliding window was applied to the landmark range measurements provided in the dataset, such that for each $\mathbf{x_t}$, $\mathbf{y_{t, Ln}}$ is 1 if landmark *n* was observed in the last *w* seconds, and 0 otherwise. *w* was set to 2s for all experiments discussed here. The data generated in this manner was artificially produced at a frequency of 2Hz, and nearest-neighbor interpolation was used to generate input states at that frequency from the ground-truth data.
Note that this problem formulation contains no notion of time; we must deal with time to generate the training data from the dataset, but after that, time is discarded. The classifier is simply predicting classes based on the present robot pose (averaged over 2s due to preprocessing). This makes intuitive sense, as visibility at a given moment should be independent of prior state, velocity, or any factors other than current orientation and position.

An alternative framing was also considered, in which an SVM regressor would be trained to predict the *number of observations* of a given landmark measured within the sliding window, taking advantage of the idea that the more observations we've seen, the higher the confidence we have that this landmark will always be observed at that state. Then a final classification output would be produced by means of a threshold value (a hyperparameter which could be set automatically after fitting the regressor, via binary search). However, in practice, the simple classifier worked very well and so this alternative formulation was not implemented.

The most important hyperparameter for SVM classification is the kernel type; common kernels include linear, radial basis function (RBF), sigmoid, and polynomial. Each of these come with their own tuning parameters, and are adept at classifying with different kinds of surfaces. Linear kernels, as described above, split the data into two parts by a hyperplane in the input space. Roughly speaking, polynomial kernels of order *n* create a polynomial decision surface, and are best for separating data where the decision boundary is a smooth curve. Sigmoid kernels operate in a manner analogous to a sigmoid-activated 2-layer MLP. RBF's are the most flexible, and (speaking heuristically) create an arbitrary number of hyperspherical decision surfaces in the input space.[2]
To gain a better intuitive understanding of the operation of these different kernels, an initial exploration of the impact of kernel type on classification performance was performed using *scipy*'s *svm* module, leaving the default hyperparameters untouched for each. Based on this preliminary exploration, and also on the visual properties of the data as displayed in Fig A2.1 above, RBF was selected as the kernel to implement.

RBF kernels require the selection of 2 hyperparameters:

- σ - the standard distribution of the gaussian used in the RBF kernel; broadly, this sets the radius of the hyperspheres which will be generated to create decision boundaries across the input space. Small values may lead to overfitting (to the extreme case of perfectly enclosing each 1-point in the training set with a decision boundary), while large values will lose accuracy by restricting to larger regions
- *C* - controls the degree to which mis-classified points are punished when finding the optimal decision surface. Higher values of C create a tighter decision boundary and may result in overfitting, while lower values soften the decision boundary and may underfit.

The process of hyperparameter tuning for this particular problem is described in section B below.

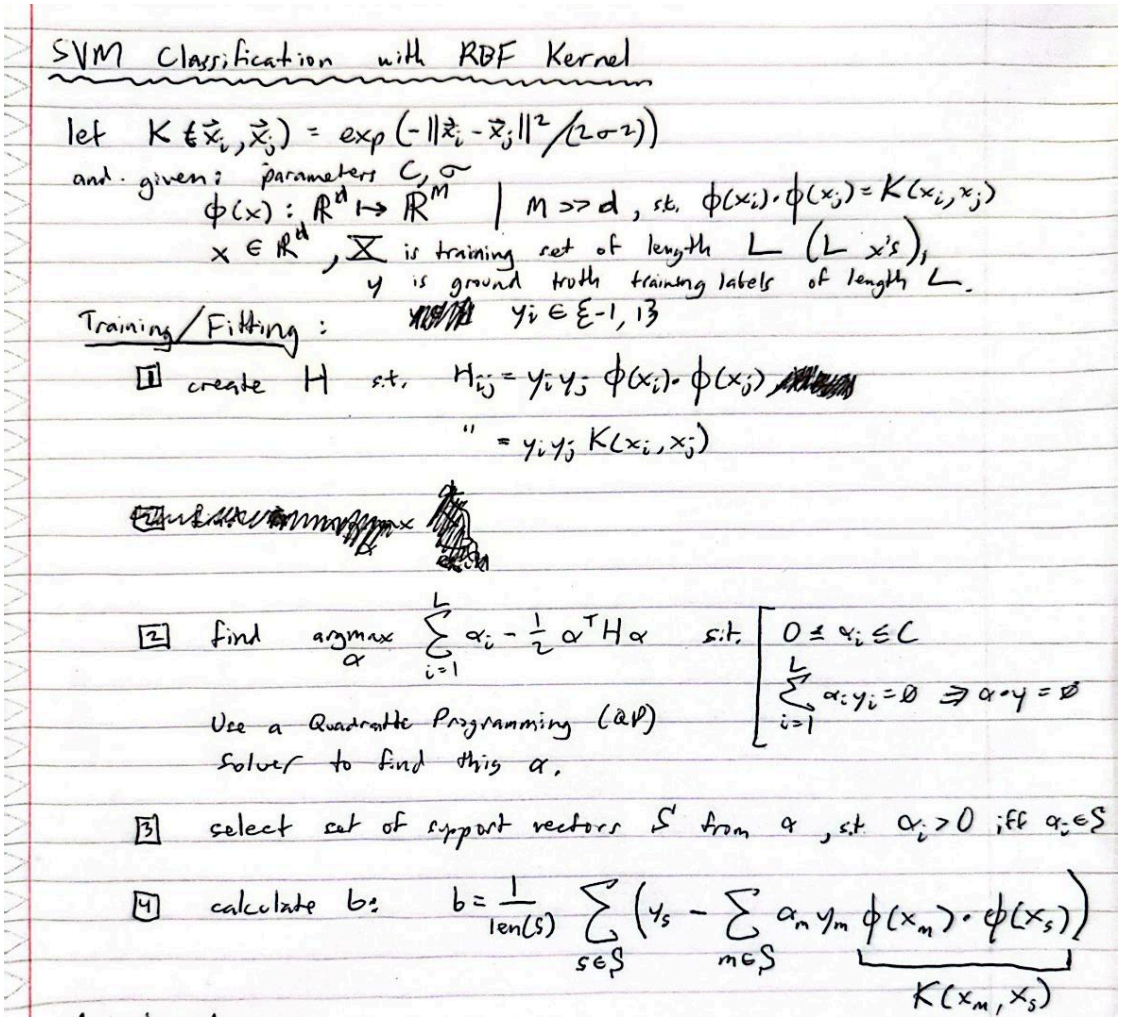
A4. Briefly describe the operation of your learning algorithm (including equations as appropriate)

SVM classification works by partitioning space into two regions (one for each class) by a hyperplane which optimally separates the two classes. In the case of non-linear SVM, this separating hyperplane exists in a higher-dimensional feature space (which may even be infinite-dimensional), which we access via a mapping $\phi(\mathbf{x})$ from input space to feature space. The hope is that while the training data may be poorly separated by a hyperplane in input space, it may be better-separated by a hyperplane in this high-dimensional feature space.

However, in general we cannot evaluate $\phi(\mathbf{x})$ directly; instead, we implicitly define $\phi(\mathbf{x})$ by means of a "kernel" function **K** over the input space, for which $\phi(\mathbf{x_i}) \cdot \phi(\mathbf{x_j}) = \mathbf{K}(\mathbf{x_i}, \mathbf{x_j})$. Since we can evaluate $\mathbf{K}(\mathbf{x_i}, \mathbf{x_j})$ numerically, the dot product $\phi(\mathbf{x_i}) \cdot \phi(\mathbf{x_j})$ is computable despite the fact that we do not have a computable representation of $\phi(\mathbf{x})$ itself.
Furthermore, the nonlinear SVM algorithm is constructed such that we only use $\phi(\mathbf{x})$ in the context of its dot product, meaning that we can perform all necessary operations in this

high-dimensional feature space (in which the classes are–hopefully–linearly separable) without ever having to numerically compute within it.

In this discussion we focus on the RBF kernel, in which **ϕ(x)** is a projection into an infinite-dimensional feature space implicitly defined by the kernel shown below, reminiscent of a gaussian parameterized by σ. The algorithm itself is as follows:



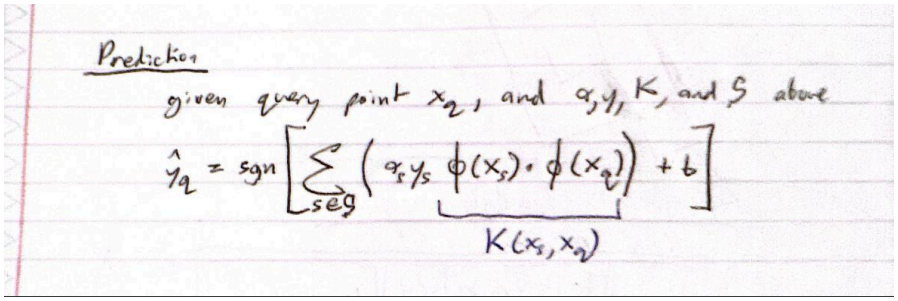


Fig A4.1 - pseudocode for SVM Classification with Radial Basis Functions. Note that the actual implementation is vectorized using numpy for performance (~4s per classifier over the training set), so many of these loops and sums are implicit.

Note: In order to solve the QP problem in step (2) in Fig A4.1, the *clarabel* solver was used via the python library *qpsolvers*[3]. This solver was selected for accuracy (based on results in *qpbenchmark*) and speed for this particular problem (based on empirical comparison on this dataset against several other common solvers).

A5. Demonstrate the algorithm's functioning on a simple dataset

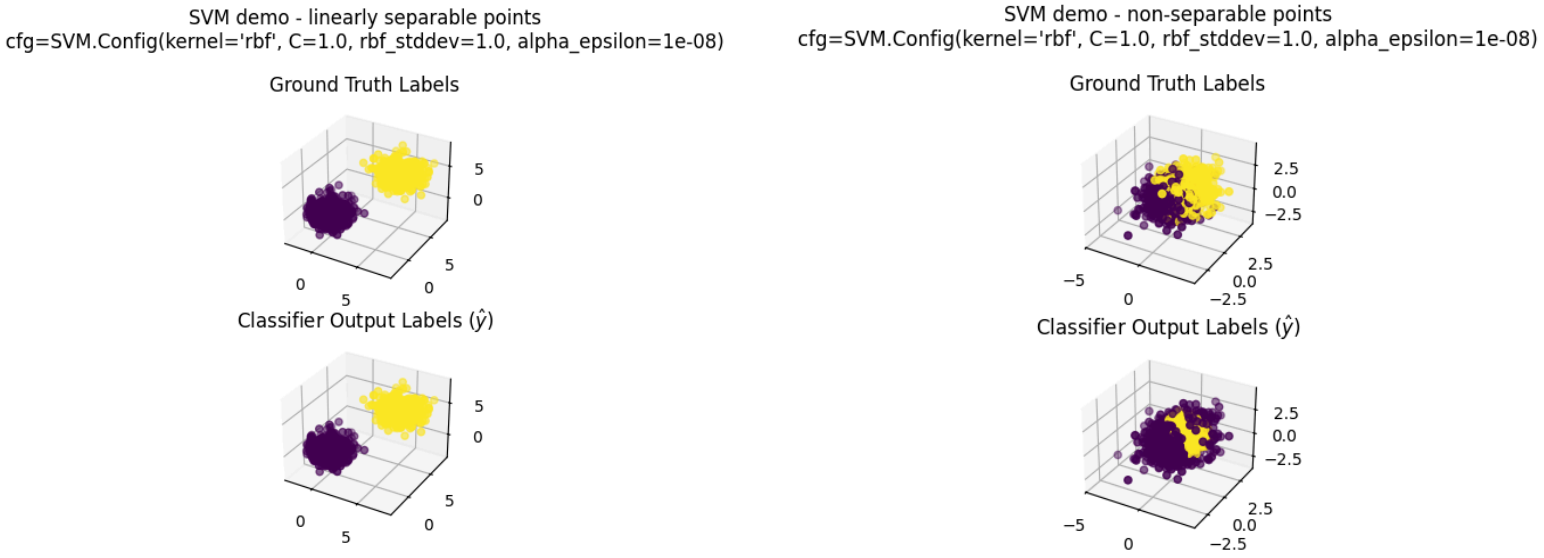


Fig A5.1 - Performance of the SVM implementation with parameters (C=1, σ=1) on 2 sample datasets created by sampling 500 points from 2 gaussians with different means, one for each of output class. In the left figure, the point clouds do not overlap, and we see the classifier performing with perfect accuracy relative to the ground truth. In the right figure, the point clouds overlap, and we see the classifier's accuracy suffer as a result. We can also roughly see that the shape of the cluster of y=1 predictions (yellow) in the bottom right is roughly spherical, which is typical of RBF.[1]

Part B

Apply your learning algorithm to your problem framing developed in Step 3. Evaluate its performance using at least two measures.

In order to train and evaluate the classifier, the dataset was split into training and test sets, by randomly selecting 80% of the dataset for training and leaving the remaining 20% out for test.

Random selection is important here because the dataset is time-ordered, and it takes time for the robot to drive around; meaning that we risk our train/test sets containing more disjointed sets of input states by taking the first 80/last 20 unshuffled (i.e. if the robot drives to a new area near the end of the recording).

The following two measures were used to evaluate the performance of the classifier (where **N** refers to the number of samples in the test set):

- Accuracy: $\mathbf{N_{correct}}$ / **N** , averaged over all classifiers.
- Recall: $\mathbf{N_{visible,\ correct}}$ / $\mathbf{N_{visible}}$, averaged over all classifiers. This statistic is valuable because each landmark is only visible for a relatively small region of the input space, meaning that a fairly high accuracy could be achieved by classifying all points to -1=invisible. Recall lets us observe how well landmarks are being detected when they are actually present.

Based on these 2 metrics, a grid search was performed to discover the best tuning, resulting in the selection of **(C, σ) =** (10, 0.5), which is used for the remainder of this analysis. For these parameters, the SVM + RBF implementation achieved 97% accuracy and 88% recall.

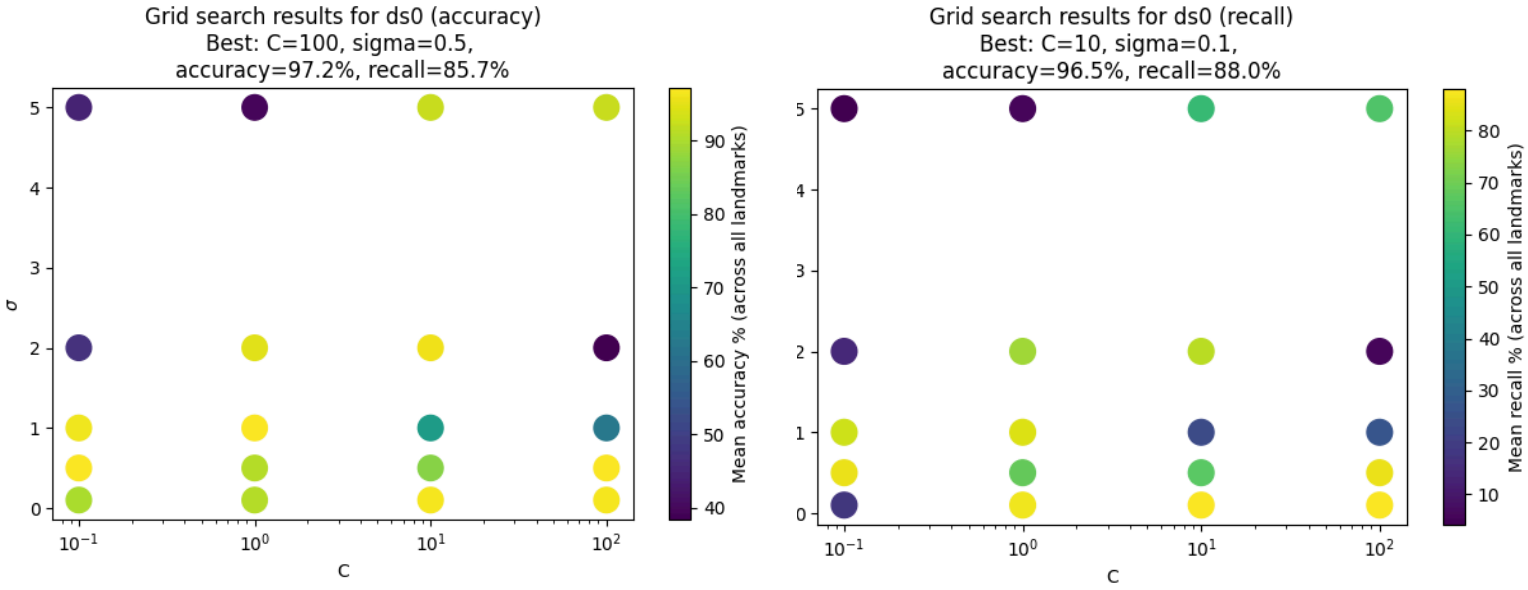


Fig B1 - Grid search over parameters **C** = {.1, 1, 10, 100}, **σ** = {.1, .5, 1, 2, 5}. For each combination **(C, σ)**, a new set of classifiers (one for each landmark) is fitted to the training set, then tested on the test set, and accuracy + recall scores are computed based on the results. In the plots above, we can see that recall is generally a stricter measure than accuracy for this task; several points such as (.1, .1) score well on accuracy but terribly on recall, indicating that they rarely predict that any landmarks are visible. The best scores on both come from the pairs (10, .1) and (100, .1), representing small decision boundary spheres and large punishment for outliers. (10, .1) was selected from these to minimize overfitting.

However, we should be somewhat wary of the conclusion that high-C and low-sigma will perform the best for arbitrary input states. This combination has a higher risk of overfitting, which is theoretically ameliorated by randomly splitting train and test sets. However, the nature of the dataset is that we only have data for a fairly restricted subspace of the input space, as the robot drove repeatedly over the same trajectories. It may be that this classifier would perform significantly worse given a test set which is more evenly distributed across the state space; however, we don't have access to that data, so it's impossible to know for sure.

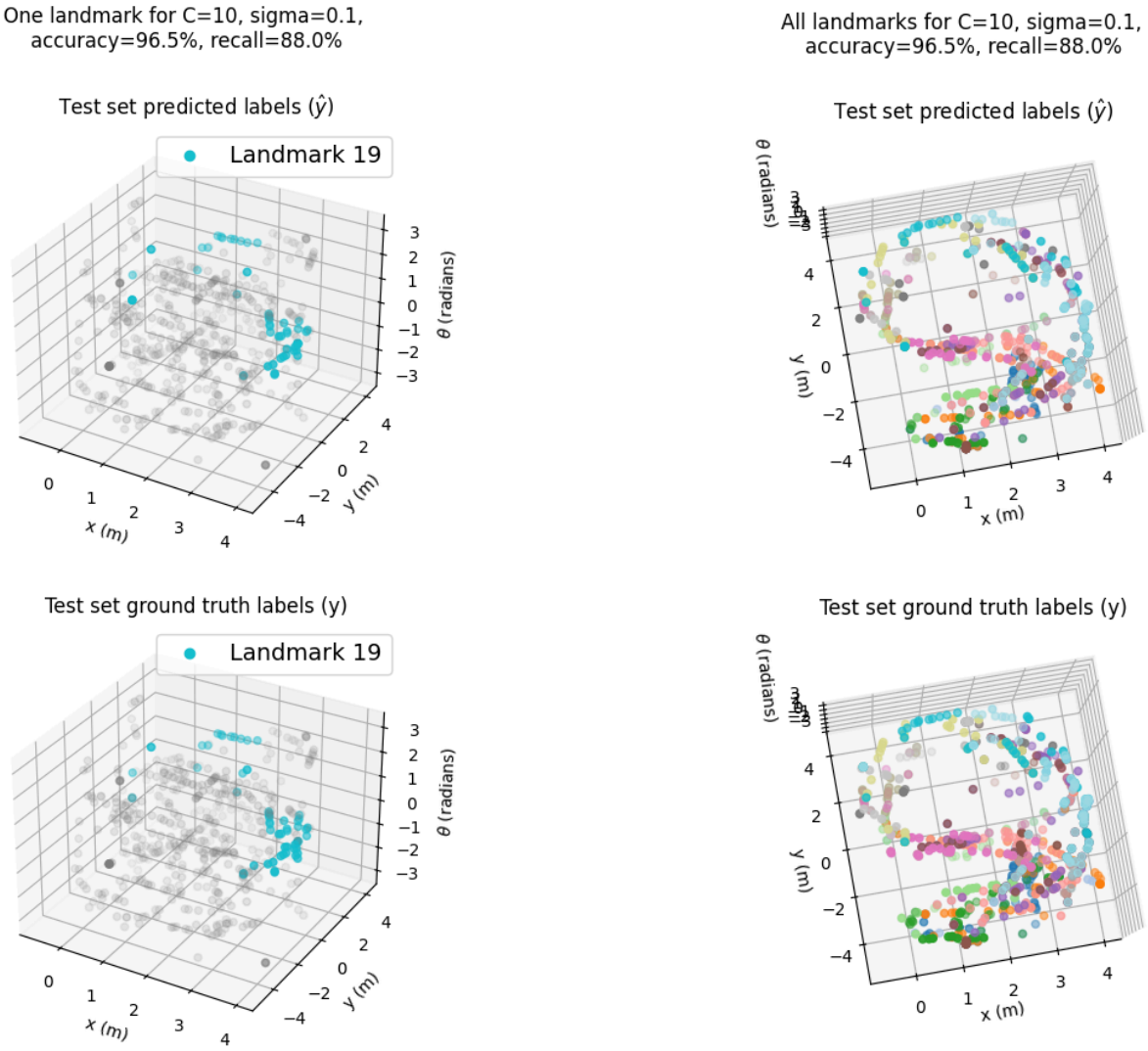


Fig B2 - classifier output (color) vs input (point coordinate) on the test set (top), compared with ground truth labels (bottom). The top and bottom classifications are visually very similar, in agreement with the high classification accuracy.

Left: results for a single landmark (19, selected arbitrarily). *Right*: results for all landmarks, viewed from above so that the familiar trajectory in xy space is visible. For points where multiple landmarks can be seen at once, the dot is (arbitrarily) colored by the landmark with the highest subject number.

Conclusion

In this paper, we successfully applied the SVM algorithm to the task of predicting landmark visibility for a mobile wheeled robot. An initial assessment of problem framings and kernel types was performed before implementing classification with the RBF kernel, which was then hyperparameter-tuned using grid search. The resulting classifier performed very well on the task, achieving accuracy and recall of 97% and 88%, respectively.

References

[1] Burges, C.J. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery* **2**, 121–167 (1998). https://doi.org/10.1023/A:1009715923555

[2] Fletcher, Tristan. *Support Vector Machines Explained*. 23 Dec. 2008. University of Western Ontario, www.csd.uwo.ca/~xling/cs860/papers/SVM_Explained.pdf.

[3] Caron, Stéphane, et al. *qpsolvers: Quadratic Programming Solvers in Python*. Version 4.8.1, 2025, https://github.com/qpsolvers/qpsolvers