Machine Learning for Data Science - Competition 2

Antoine Klopocki, Jaydev Kshirsagar, Travis Westura, Vishisht Tiwari
Kaggle Team Name: Antravishjay

1 Introduction

In this competition we are given a robot that is moving around in $\mathbb{R}^2$. The robot makes 10,000 runs, and for 1,000 timesteps we observe the angle θ that the robot's position makes with the x-axis. But are not given the robot's precise (x, y) location, and the angle itself does not determine directly the robot's position. The true position could be anywhere along the line passing through the origin and making angle θ with the x-axis.

Our task is to determine the robot's (x, y)-position on the 1,001st timestep as accurately as possible, with accuracy determined by Root Mean Square Error. Our methods achieve an accuracy score of **0.26233**.

2 Determining the Observer's Position

The competition specification describes the robot moving in the first quadrant, and the angles given in the observations file contain the angle made between the robot's location $(x_{r,t}, y_{r,t})$ and the x-axis. However, the coordinates given in the label file contain coordinates with negative values. We'll denote to these values as $(x'_{r,t}, y'_{r,t})$. Since these values are negative, the observer of the locations is positioned in the first quadrant as well, and we call it's location (a, b). First we need to determine the coordinates of this point so that we can match up the data in the observation and label files.

Consider a labeled point $(x'_{r,t}, y'_{r,t})$. The angle θ that this point forms with the x-axis is given by

$$\tan\theta = \frac{b + y'_{r,t}}{a + x'_{r,t}}.$$

We know the value of θ from the observations file. Since there are two unknowns a and b, we pick two labeled points and solve the system of equations

$$\tan\theta_1 = \frac{b + y'_{r_1,t_1}}{a + x'_{r_1,t_1}}, \quad \tan\theta_2 = \frac{b + y'_{r_2,t_2}}{a + x'_{r_2,t_2}}.$$

We use the points from run 1 time steps 205 and 216. Solving the system of equations yields the position of the location observer as $(1.5, 1.5)$. Using this position, we consider the observation angles as being measured at $(0, 0)$.

Further, we can use the 600,000 labeled points to gain intuition about the robot's movement. We compute the average of radius of these points to $(1.5, 1.5)$ to see that this average is approximately 1. Plotting the labeled points in figure 1, we see that the robot is moving roughly in a circle of radius 1 centered at $(1.5, 1.5)$, with kinks occurring at the top, bottom, left, and right of the circle.

Every run starts the robot at $(1.5, 1.5)$. But we are not given any labels for the first 200 runs, and we see that the robot stabilizes into its standard path by the time we start obtaining labels for its location.

3 Hidden Markov Models (HMM's)

We represent this problem as a Hidden Markov Model. A Hidden Markov Model is a graphical model with two types of variables: latent variables S_t and observed variables X_t. The latent variables represent states that are not visible to the observer. Each observed variable X_t depends only on the corresponding state variable S_t. And each state variable S_t depends only on the previous state variable S_{t-1}. We depict latent variables as shaded vertices.

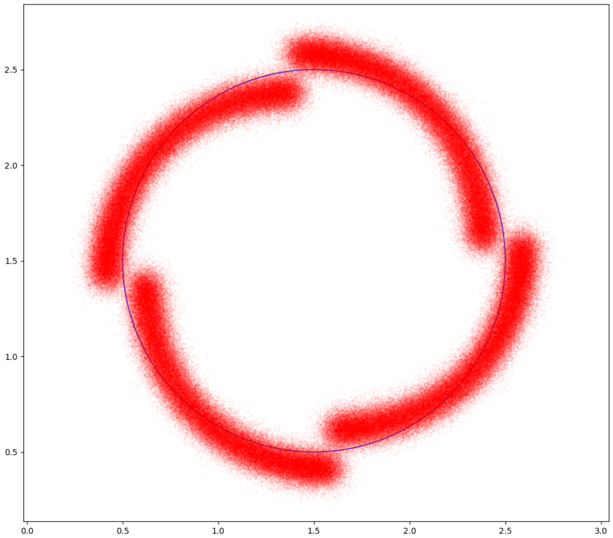


Figure 1: Locations of the robot given as labels.

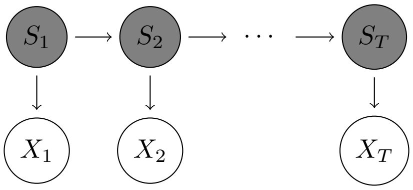


Figure 2: Hidden Markov Model

In our problem the observed values are the angles that the robot's position makes with the x-axis. The robot is moving around in the plane $\mathbb{R}^2$. This space is continuous, so our observations are angles in the continuous range $\big(0, \dfrac{\pi}{2}\big)$. We discretize this space in order to use a Hidden Markov Model. We divide the first quadrant into K sectors, $\big[0, \frac{1}{K}\frac{\pi}{2}\big), \big[\frac{1}{K}\frac{\pi}{2}, \frac{2}{K}\frac{\pi}{2}\big), \ldots, \big[\frac{K-1}{K}\frac{\pi}{2}, \frac{\pi}{2}\big)$, where K is a parameter that we choose. The observation, rather than being a value in $\big(0, \dfrac{\pi}{2}\big)$, is instead given by an integer $1, 2, \ldots, K$ representing the segment of the interval in which the angle lies. As a further refinement of this technique, we find the minimum and maximum angles that occur in the observations file, $\theta_{\min}$ and $\theta_{\max}$, and divide the shorter interval $[\theta_{\min}, \theta_{\max}]$.

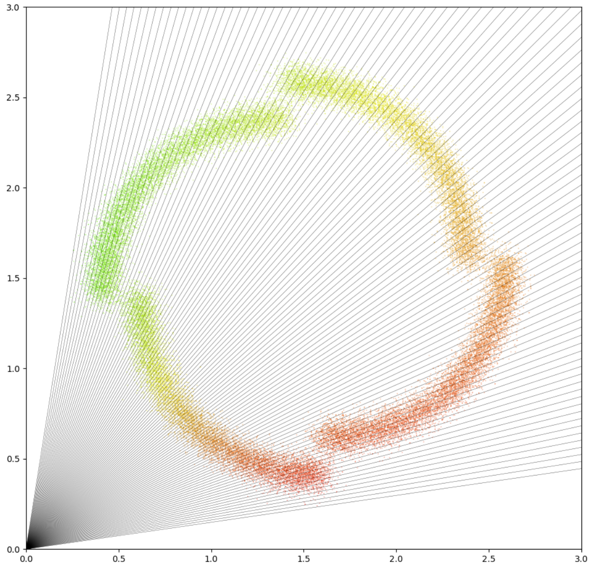


Figure 3: Labeled points colored based on the segment in which they lie with $K = 100$.

Given an observation and the corresponding state, we need to map the state to the robot's position. We outline a procedure for doing this in .

Given the observations, we need to estimate the transition matrix A and emission matrix B of the Hidden Markov Model. The transition matrix is defined by

$$a_{i,j} = \mathbf{P}(S_t = j \mid S_{t-1} = i),$$

that is, each entry $a_{i,j}$ gives the probability of being in state i given that the previous state is state j. With N states, A is an $N \times N$-matrix. The emission matrix is defined by

$$b_{i,k} = \mathbf{P}(X_t = k \mid S_t = i),$$

that is, each entry $b_{i,k}$ gives the probability of the observation k being emitted given state i. With N states and K possible observations, B is an $N \times K$-matrix. In section 5 we describe our process for estimating these matrices using the Baum Welch algorithm.

4 Algorithms and Implementation

We model this problem as an HMM learning problem, where we need to use and algorithm to learn the transition, emission, and initial probabilities. Our main tool is the Baum Welch algorithm.

5 Baum Welch

The Baum Welch algorithm is an Expectation Maximization (EM) algorithm for learning the parameters of Hidden Markov Models. We use a forward-backwards algorithm to perform inference for the expectation step and then update the HMM parameters in the maximization step. We thereby find the maximum likelihood estimate of the parameters of the model. The algorithm takes as parameters a triple (A, B, π), where A and B are the transition and emission matrices and π is the initial state distribution, that is, the probability of the first state being state i is given by $\pi_i = \mathbf{P}(S_1 = i)$.

5.1 Our Implementation of Baum-Welch

To achieve a high accuracy, we develop our own Baum-Welch algorithm. After using the basic implementation, we tweak the alogrithm to achieve better results.

5.1.1 Description of Algorithm

Let N be the number of states, K be the number of observations, and T be the number of time steps.

For the forward procedure we calculate $\alpha_i(t) = \mathbf{P}(X_1 = x_1, X_2 = x_2, \ldots, X_t = x_t, S_t = i \mid A, B, \pi)$, which is the probability of obtaining observations $y_1, y_2, \ldots, y_t$ and being in state i at time t. We recursively compute

$$\alpha_i(t) := \pi_i b_{i,x_1},$$
$$\alpha_i(t+1) := b_{i,x_{t+1}} \sum_{j=1}^{N} \alpha_j(t) a_{j,i}.$$

For the backward procedure we calculate $\beta_i(t) = \mathbf{P}(X_{t+1} = x_{t+1}, \ldots, X_T = x_T \mid S_t = i, A, B, \pi)$, the probability of the observations $x_{t+1}, \ldots, x_T$ occurring given the tth state is state i. Again we compute recursively to set

$$\beta_i(T) := 1,$$
$$\beta_t(t) := \sum_{j=1}^{N} \beta_j(t+1) a_{i,j} b_{j,x_{t+1}}.$$

Before performing the updates, we first calculate two temporary variables. We define $\gamma_i(t)$ to be the probability of being in state i at time t given a set of observations $X = (X_1 = x_2, \ldots, X_T = x_T)$. Applying Bayes's theorem we have

$$\gamma_i(t) := \mathbf{P}(S_t = i \mid X, A, B, \pi) = \frac{\mathbf{P}(S_t = i, X \mid A, B, \pi)}{\mathbf{P}(X \mid A, B, \pi)} = \frac{\alpha_i(t)\beta_i(t)}{\sum_{j=1}^{N} \alpha_j(t)\beta_j(t)}.$$

Next we define $\xi_{i,j}(t)$ to be the probability of being in state i at time t and state j at time $t+1$ given a sequence of observations X with parameters A, B, and π.

$$\begin{aligned}\xi_{i,j}(t) := \mathbf{P}(S_t = i, S_{t+1} = j \mid X, A, B, \pi) &= \frac{\mathbf{P}(S_t = i, S_{t+1} = j, X \mid A, B, \pi)}{\mathbf{P}(X \mid A, B, \pi)}, \\ &= \frac{\alpha_t(t) a_{i,j} \beta_j(t+1) b_{j,t+1}}{\sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i(t) a_{i,j} \beta_j(t+1) b_{j,t+1}}.\end{aligned}$$

We use these temporary variables to update the parameters of our Hidden Markov Model. The vector π is updated so each entry π_i is the probability of being in state i at the first time $t = 1$.

$$\pi_i := \gamma_i(1)$$

The values $a_{i,j}$ are updated to be the expected number of transitions from state i to j divided the total number of transitions from i (including from i back to itself).

$$a_{i,j} := \frac{\sum_{t=1}^{T-1} \xi_{i,j}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)}$$

And finally the values $b_{i,k}$ is set to the expected number of times the observation k is emitted from state i over the total number of times state i occurs.

$$b_{i,k} := \frac{\sum_{t=1}^{T} \mathbf{1}_{X_t = k} \gamma_i(t)}{\sum_{t=1}^{T} \gamma_i(t)}$$

5.1.2 Implementation on a small dataset

We implement this algorithm in Python and first test it with a small data set of 2 states and 3 observation types. We randomize the initial values of our parameters A, B, and C and run for 100 iterations.

5.1.3 Implementation on the Robot Challenge

After testing our implementation with a small parameter size, we attempt to use it to determine the HMM parameters of the robot challenge. Because the robot moves in a continuous space, we discretize the space so that our algorithm, which accepts discrete inputs, can handle it. We first tried using large parameters, setting the number of states $N = 3{,}000 and the number of observations$ K = 1,570. We round all observation angles to 3 decimal places to fit our discretization model. The algorithm runs as follows

1. Randomly initialize the transtiion, emission, and initial probability matrices using 3,000 states and 1,570 observations.
2. Use the Baum Welch algorithm on the first run of the robot with a maximum of 100 iterationsl
3. Obtain the new transition, emission, and initial probability matrices as the result.
4. Repeat for all 10,000 runs, updating the parameters with each run.

Results This experiment fails to produce a suitable result. The algorithm was not even able to complete the first iteration of the first run in half an hour. Modifications such as reducing the number of iterations, reducing the number of states and observations, and subsampling the time steps to use fewer than all 1,000 steps for each run fail to produce a suitable reduction in running time.

5.2 Baum-Welch Using External Pythin Code

With our own implementation failing to yield viable results on the data set, we next use the Python library `hidden_markov`, available here: `http://hidden-markov.readthedocs.io/en/latest/functions.html#baum-welch-algorithm`. We follow the same steps for training the model: randomly initialize the parameters, run the algorithm on one run at a time to update the parameters, and use the updated parameters for the number run.

Results However, we again face problems with running time when using this library, with convergence taking over an hour even for small parameter sizes and subsampling of the timesteps. Further we encountered numerical underflow with this implementation. As the algorithm involves multiplying together many small probabilities, this multiplication produces smaller and smaller numbers, eventually yielding 0 despite the multiplication occuring between nonzero numbers. Although the documentation of the library did mention that the Baum Welch algorithm takes the numerical underflow problem into consideration when computing the probability matrix, in our case, the library was unable to proceed beyond the second run without this problem occurring.

5.3 Baum-Welch using Python Libraries

We continue searching for a library to use for running Baum-Welch and further try running the implementation available here:
`http://www.katrinerk.com/courses/python-worksheets/demo-the-forward-backward-algorithm`. The level of performance of this algorithm on the small database previously tests is equivalent to our algorithm. However, this program is marginally faster in the bot challenge than our Baum-Welch implementation. Numerical underflow, however, continues to be a problem. The program performs well for fewer than 70 timesteps, but struggles when using 70 or more.

5.3.1 Algorithms

The term numerical underflow is used when the result of a calculation is smaller than the smallest minimum value that a computer can store in memory. In Baum-Welch numerical underflow can occur in very large observations sequences because of the multiplication of probabilities. A description of this problem and possible implementations to overcome it are described in this write-up: `https://pdfs.semanticscholar.org/54dc/c2a758e7fa34b8c2ef19826f39f16c4d1731.pdf`.

One solution involves using the log of probabilities to convert the multiplications into addition. This process succeeds with the α's and β's computed in the forwards-backwards algorithm, but not for the γ's we compute as temporary variables, as they are already a sum of the α's. Hence to avoid underflow, values of α and β are normalized. We compute

$$\hat{\alpha} = \frac{1}{\sum_{i=0}^{N} \alpha_i(t)}, \qquad \hat{\beta} = \frac{1}{\sum_{i=0}^{N-1} \beta_i(t)}$$

where $\hat{\alpha}$ and $\hat{\beta}$ denote the normalized values. The γ values are calculated as before, since the normalizers are cancelled. But the ξ values now have a slightly different formula.

$$\gamma_i(t) = \frac{\hat{\alpha}_i(t)\hat{\beta}_i(t)}{\sum_{j=1}^{N} \hat{\alpha}_j(t)\hat{\beta}_j(t)}, \xi_{i,j}(t) \qquad = \frac{\hat{\alpha}_i(t)a_{i,j}b_{j,t+1}\eta_{t+1}\hat{\beta}_j(t+1)}{\sum_{j=1}^{N} \hat{\alpha}_j(t)\hat{\beta}_j(t)} = \frac{\gamma_i(t)a_{i,j}b_{j,t+1}\eta_{t+1}\hat{\beta}_j(t+1)}{\hat{\beta}_i(t)}.$$

We use this algorithm we same way we have the previous implementations: randomly initialize the parameters, train on the first run to obtain new parameter values, and continue using the algorithm to update hte parameters for all 10,000 runs.

Results Unfortunately the normalization steps again reduce the efficiency of this Python code, and it again takes an unreasonably long time to complete the iterations. Again we try various modifications to the parameters, such as reducing their sizes and subsampling time steps, but we still cannot reach a reasonable level of efficiency with our code.

5.4 Baum-Welch in Matlab

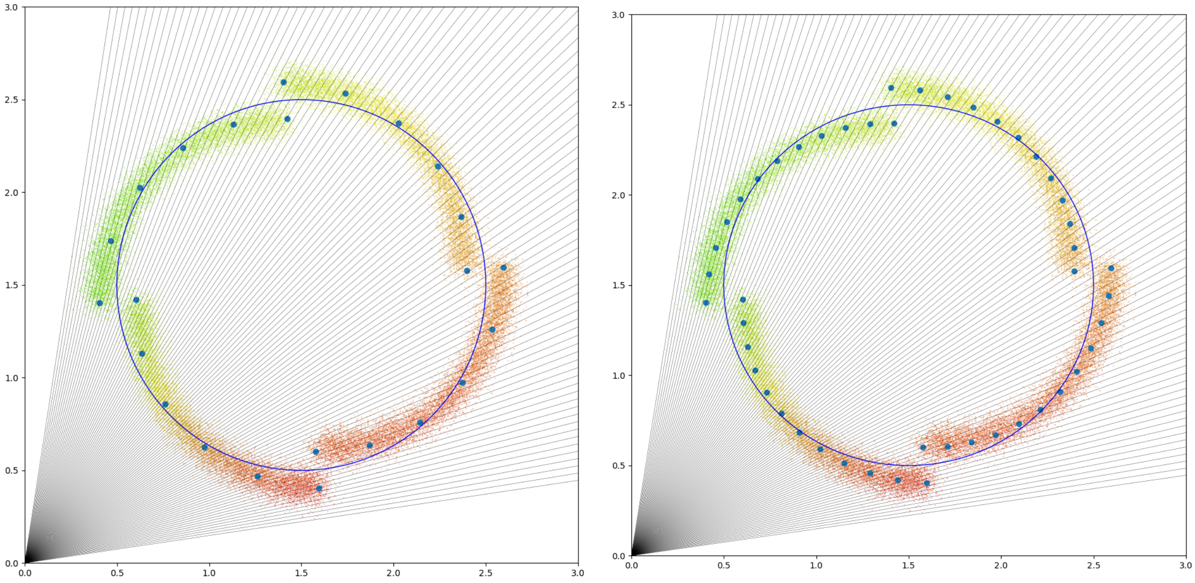
With unsuccessful results form Python libraries, we next use the `hmmtrain` function of Matlab. This function uses the observation sequence, transition matrix, and emission matrix to learn about the HMM and to predict the new transition and emission matrices. This implementation is considerably faster than the Python implementations we used. The transition and emission matrices converges about processing the first 200 runs and took only about 3 hours to run.

5.4.1 Implementation and Choice of Parameters

One way to initialize the parameters A, B, and π of the Baum Welch algorithm is to do so randomly. However, doing so means the algorithm takes a long time to converge. By choosing initialization parameters close to what we expect the algorithm's output to be, we can decrease the number of steps that the algorithm requires to converge.

To initialize the transition matrix, we take advantage of the fact that the robot is moving rather slowly around the circle. That is, in only one step, the robot is likely either to stay in the same state or move to a state with a location close to its previous state's location. It will not make large transitions, e.g. from the bottom to the top of the circle, in a single step. Thus it makes sense to initialize many values of the transition matrix to be 0, as there are only a small number of states to which the robot may transition from any given state.

Further, the Baum Welch converges to a local maximum. And by changing the initial parameters we can change the local maximum to which the algorithm converges. Thus we choose to initialize our transition matrix so that the states correspond to sections of the robot's path. In figure 4 we show how we pick locations to guide our construction of the transition matrix. We pick points along the robot's path and choose transition probabilities either that the robot stays in the same state or moves to the next state. For example, for 4 states, our initial transition matrix is



(a) N = 24.

(b) N = 48.

Figure 4: Initial state positions for different numbers of states.

$$\begin{bmatrix} 0.75 & 0.25 & 0 & 0 \\ 0 & 0.75 & 0.25 & 0 \\ 0 & 0 & 0.75 & 0.25 \\ 0.25 & 0 & 0 & 0.75 \end{bmatrix}$$

Further, we decide not to use any of the first 200 time steps in our model. The robot starts at $(1.5, 1.5)$, but by the time we start receiving labels at time 201, the robot has already reached its general orbit around its path. We don't have any labels for the first 200 time steps, so using these values ends up making it more difficult for our algorithms to estimate the behavior of the robot at the tail of its runs, which is the behavior in which we are most interested to predict the 1,001st location. When we try using these initial observations, our Baum Welch algorithm outputs a state to keep track of the values in the middle. Figure 5 shows how these initial observations result in points being assigned to a state that does not correspond to a useful location in the tail of the robot's path, but instead results in just a few points being assigned to this state instead of to a more reasonable location around the ring of the circle.

We also attempt to initialize the emission matrix. After placing the initial state centers along the arcs of the robot's path, we assign each labeled point to the closest state based on the Euclidean distance between the point and the state's center. Since we discretize the angles, we now determine the number of labeled points in each observation angle out of the total number of points assigned to the state. This process gives us the initial values of our emission matrix. Again, many of the values of this matrix are 0, since a state that corresponds to the bottom right of the circle will never emit observations that correspond to angles close to $\frac{\pi}{2}$.

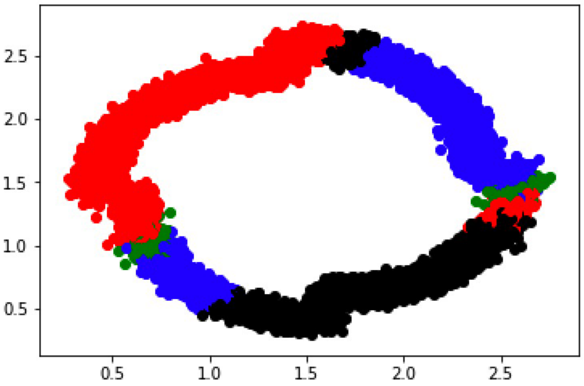


Figure 5: The green points are assigned to a state representing the middle of the circle through which the robot finishes moving by time 200.

However, this initialization of the emission matrix does not produce good results. The large number of 0's present in the matrix tend to result in numerical errors when running the Matlab implementation. We attempt to overcome this issue by adding an extremely small value, such as 0.00000001, to the entries of the matrix, but we were unable to remove completely the numerical issues. Randomly initializing the emission matrix does not create these numerical issues, so in our final implementation we use the randomly initialized matrix.

Results This implementation produces the best result, and we are able to achieve distinct states in a reasonable amount of computation time. In the final mode, the Baum-Welch Matlab code is used to compute the transition and emission matrices before using the Viterbi algorithm to determine the most likely states.

6 Viterbi

Viterbi is a dynamic programming algorithm that is used to predict the most likely hidden states. However, Viterbi training can also be used for the training of a HMM. We consider Viterbi training as an alternative to Baum-Welch training, which are computationally slow. We describe both uses of Viterbi in this section.

6.1 Algorithm

The algorithm is used for predicting the hidden states for a path $X = (x_1, x_2, \dots, x_T)$. The hidden states are denoted as $x_n \in S = \{S_1, s_2, \dots, s_K\}$, and the observations are denoted by $Y = (y_1, y_2, \dots, y_T) \in \{1, 2, \dots, N\}^T$. Again, we use N for the number of states, K for the number of observations, T for the number of time steps, and A and B with elements $a_{i,j}$ and $b_{i,k}$ for the transition and emission matrices, respectively.

The algorithm uses two tables of size $K \times T$.

- In the first table, each element $T_1[i,j]$ of T_1 stores the probability of the most likely path so far, which we denote by $\hat{X} = (\hat{x}_1, \dots, \hat{x}_j)$. Here $\hat{x}_j = s_j$, which generates $Y = (y_1, \dots, y_j)$.

The entries in the tables T_1 and T_2 are filled in by

$$T_i[i,j] := \max_k \{T_1[k, j-1] a_{k,i} b_{i,y_1}\},$$
$$T_2[i,j] := \arg\max_k \{T_1[k, j-1] a_{k,i} b_{i,y_1}\}.$$

6.2 Implementation of Viterbi for Predicting States

Because Viterbi is used for predicting states, it is the obvious next step after predicting the transition and emission matrices. Hence after predicting the probabilities with Baum-Welch, we use this algorithm to predict the states. We use Matlab's `hmmviterbi` function. This algorithm performs efficiently for us and has no major problems in computing the hidden states. The steps for using this algorithm are as follows:

1. Obtain the transition and emission matrices from the Baum-Welch algroithm.
2. Provide the transition matrix, emission matrix, and observations from the first run to the Viterbi algorithm. In this case we use 1,570 observations.
3. The Viterbi algorithm outputs an array containing the hidden states that correspond to the observations.
4. Repeat for all 10,000 runs.
5. At the end obtain a matrix of size 10,000.

Note here that we use only 800 observations (from 200 to 1,000) for every run since these points lie withing the orbit and not in the middle of the circle.

Results We use three different numbers of states: 16, 24, and 48. After obtaining the states, the labels for the first 100 runs are plotted to study if the Viterbi algorithm is successful in determining the states. The results are in . The best results are with 24 states, so we decide to use 24 states in our final model.

6.3 Implementation of Viterbi Training

Viterbi training, also knows as segmental K-means, works on the same principle as K-means. The Viterbi algorithm is first used to predict the states of hte observations using randomly initialized transition and emission matrices. After deriving the hidden states, the transition and emission matrices are then derived using the hidden states and observations. These two steps are repeated until the matrices converge. Unlike Buam-Welch, Viterbi training doesn't give the full conditional likelihood by only the maximum conditional likelihood, and hence is 1 to 2 orders faster than Baum-Welch.

We implemented this algorithm from scratch in Python. The steps of our implementation are as follows:

1. Randomly initialized the transition and emission matrices.
2. Use the transition matrix, emission matrix, and run 1 observations as the input.
3. Derive the hidden states, then derive the new transition and emission matrices using the hidden state sequence.
4. Repeat until convergence.
5. Use the new matrices to run Viterbi training on the observations from run 2.
6. Repeat for all 10,000 runs.

Figure 6 gives a pictorial representation of the Viterbi training algorithm and show that the same observation is used for predicting the hidden states and the emission and transition matrices again and again. This process is repeated until convergence.

Results While optimistic that this algorithm would prove more efficient at training our HMM parameters than Baum-Welch, the results we obtained are not satisfactory. We attempted this method several times, making a variety of small alterations, but never received a promising result when using it with the robot challenge. The state sequence derived from this algorithm is not in any pattern. Because the bot moves a limited distance every time step, there is no possibility of drastic changes in state between the two consecutive time steps. Unfortunately, this algorithm produced large changes in state that did not follow any pattern. We were unable to find a suitable solution to this algorithm after many trials, and thus decided to continue with using the Matlab HMM tools for our model.

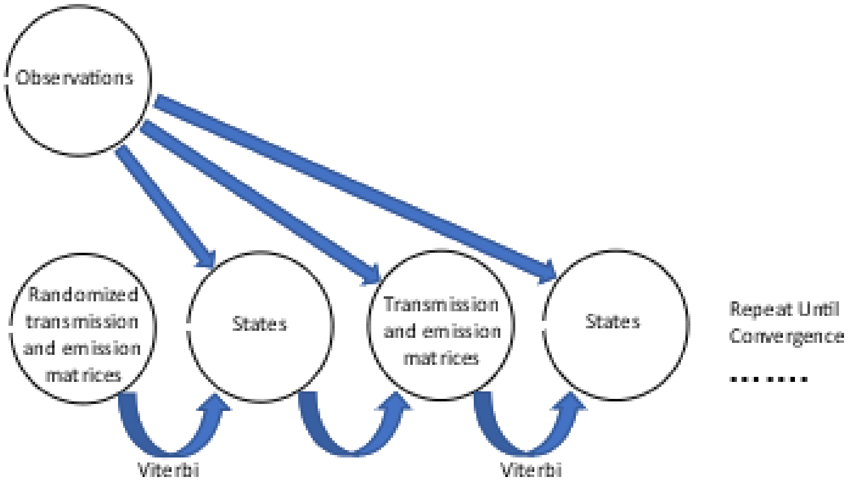

Observations
Randomized transmission and emission matrices
States
Transmission and emission matrices
States
Repeat Until Convergence
....
Viterbi
Viterbi

Figure 6: Viterbi Training

7 Finding the Location

After finding the right state, transition matrix, and emission matrix, the next step is to find the locations corresponding to the states. The project requires finding the location at the 1,001th point. However, because the bot does not move a large distance in every time step, we began by submitting simply the location of the 1,000th step to check the accuracy of our methods. Our first submission come from finding the 1,000th location of runs 6,001 to 10,000. After verifying our methods here, we then proceed to predict the location of the 1,001st point.

7.1 Finding the 1,000th Point Using Median and Projections

The biggest concern in finding the location corresponding to tan angle is that there are multiple of the circle that fall on the same angle. As seen in figure 7, the line determined by angle θpasses through two regions of the circle.

7.1.1 Using Labels to Map States to Locations

We use the labels for the first 6,000 runs to map the states produced from our algorithm to locations. We then use these state locations to determine the location of the 1,000th point in the final 4,000 runs.

1. Find the state of the 1,000th point for runs 6,001 to 10,000 using the Viterbi algorithm.

2. For each such point, find all labeled points that have the same state and discretized angle as this 1,000th point. This process solves the problem of every angle passing through multiple sections of the circle. Because the location of many points in the same state are given in the labels, it must be that region of the circle that is close to these datapoints.

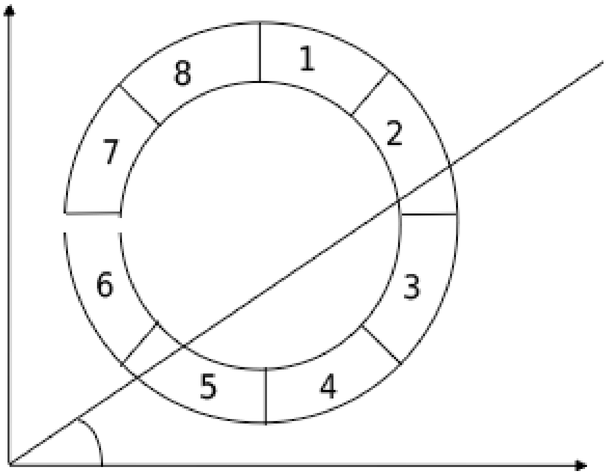


Figure 7: Multiple Intersections Determined by One Angle

3. If one or more labeled points are found with the same state and discretized angle, then the median of these points is determined and used as the location of the 1,000th point.
4. If no points are found at the same angle and state, the take the median of all the labeled points in the same state (not just same state and same angle) and project this median on the line determined by the observation angle. Use this projection as the location of the 1,000th point.
5. Repeat for runs 6,001 to 10,000.

Results The results from this algorithm are promising. By predicting the 1,000th point with the preceding algorithm, we obtain a Kaggle score of approximately 0.28. Next we proceed to use this 1,000th point to predict the location of the 1,001st point.

8 Final Model

9 Conclusions