CS 4786 Competition 2 Final Report

By Anilkumar Vadali(arv42), David Wang(dw472), Xinyu Ma(xm99)
Kaggle Team Name: Remember the Offset

# Kernelized SVM (Main Model)

### Motivation

Using HMM has a time issue. And given all observations (angles), an intuition is that if we could find a pattern between angles and corresponding locations then we solved our problem. Thus we try to use the kernelized SVM method to solve the regression problem.

### Model

Use of Data: We use data from both “Observations.csv” and “Label”. We use 10000 tuples of all 600000 tuples from “Label” and their corresponding angles from “Observations.csv” as target values and the training data. We use the last 4000 runs from “Observations.csv” as samples to perform regression on and use only the last 20 angles as features for each run sample. (use of data is explained in details above)

Parameters: We observe performances of different parameters Cs and gammas on predicting and choose the best ones. We also experiment on different subset size Ns to find the optimal one minimizing the error.

### Implementation Details

We choose the epsilon-support vector regression (SVR) with rbf kernel as our implementation. We will train two models for x-coordinate value prediction and y-coordinate value prediction separately. Since models produced by Support Vector Regression depend only on a subset of the training data ignoring any training data close to the model prediction, we can just use a subset instead of all angles to train and predict. Suppose the subset size was N, then each training vector consisted of N features. Taking a tuple [1, 265, 1.0557, -0.11706] of given labels as an example, we will show how to construct a training sample and its corresponding target value for an x-coordinate value estimator. Its target value is 1.0557 by the third element of the label tuple. Its features consist of N angles by retrieving entries observations[0][264-N+1] to observations[0][264]. Analogously, we can construct training samples and target values for a y-coordinate value estimator. We use 10000 samples from given 600000 labels to fit the kernelized SVM model finally. We fit our models by using function sklearn.svm.SVR(kernel=’rbf’, C, gamma).fit(training vectors, target values). After obtaining models, we perform regression on 4000 samples which correspond to the last 4000 runs, each sample has N features consisting of angles of the last N steps. We perform regression and get predicting results by using models obtained before and the function model.predict(samples).

**Experiments**

We use cross-validation method by sklearn.model_selection.cross_val_score function to adjust our parameters. C and gamma are two important parameters for an SVM model. We will adjust them repeatedly by splitting 20% of our original training data as test data. At first, we fix gamma to 0.1 to find the best parameter C. We experiment on C={100,200,……,1000} and take C=400 finally.

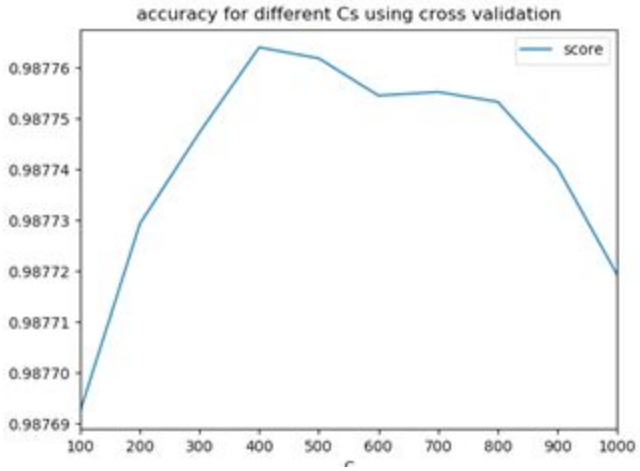


Figure 1 accuracies change with different parameters Cs when gamma=0.1

Then we fix C=400 and experiment on gamma={0.05,0.1,0.15,0.2} and take gamma=0.15 finally.

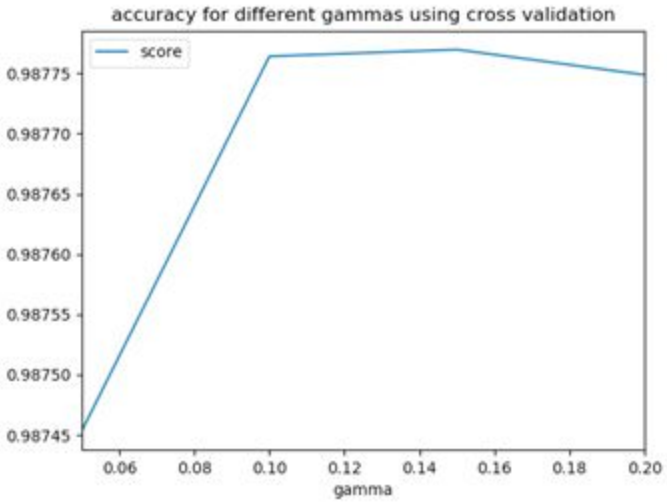


Figure 2 accuracies change with different parameters gammas when C=400

Then our task is to find a proper N when C=400 and gamma=0.15. We make a plot for the first run where the x-axis represents steps from 1 to 1000 and y-axis represents corresponding angles.

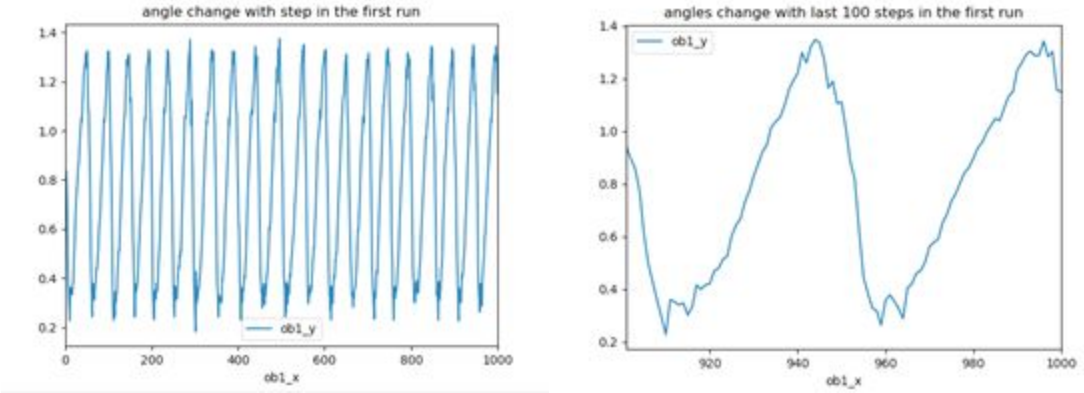


Figure 3 angles change with steps for the first run

We can observe from the plot that there is a pattern. Thus we experiment on N={30,25,20,15,10} and take N=20 finally.

| N | 30 | 25 | 20 | 15 | 10 |
|---|---|---|---|---|---|
| Error | 0.08333 | 0.07933 | 0.07847 | 0.07847 | 0.08240 |

Table 1 different subset size Ns with their corresponding errors

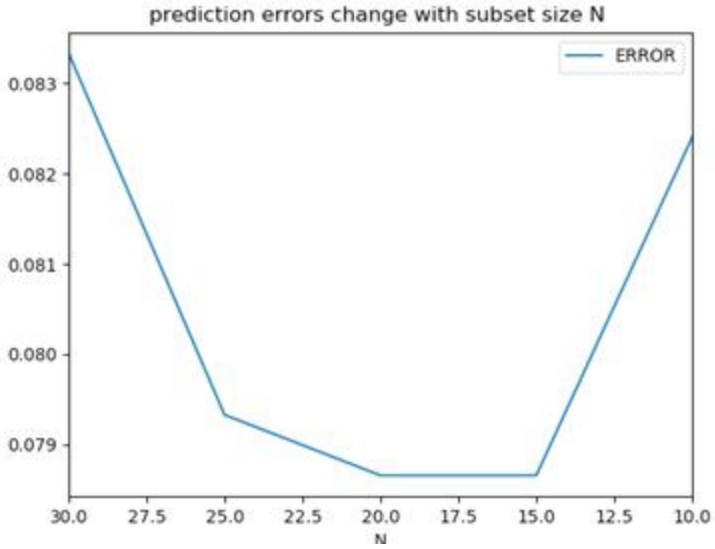


Figure 4 errors change with subset size Ns

**Results and Analysis**

This approach gives a public score 0.07847 on Kaggle which improves significantly compared to our HMM approach. The error is very tiny showing that the kernelized SVM works well.

Unfortunately, this score completed a little bit late after the deadline even though we started running the algorithm before the deadline.

# Hidden Markov Approaches (failed attempts)

**Understanding the data**

First, we realized, based on Piazza Post @608, that the positions of the points were offset by a certain amount alpha, beta, such that the angle was not being measured from the origin. To account for this, we had to solve a system of equations based on the first two lines of the labeled data set provided to us and the values corresponding to those two lines in the observations dataset. The observations dataset contains the angles for each run at each step.

The system of equations we solved was the following:

1. tan(1.2869) = (.234 - a) / (-.99408 - b)
2. tan(0.3409) = (-1.008 - a) / (-0.11297 - b)

We end up getting a = -1.5 and b = -1.5, which is how much each point is offset by. We compared these results with few other points in the labels in the dataset and they appear to work out. Thus we add 1.5 the x and y coordinates of each labeled datapoint so they align with their respective angles.

We take 2 approaches to constructing a hidden markov approach for bot localization. First, we use a continuous approach with an infinite state space, using visualization techniques to approximate the probability density function for a state's emission distribution. This was achieved by fully utilizing the labeled points, plotting them to reveal the locations that the bot was likely to be in throughout the course of its trajectory (Figure 5) based on angle (Figure 6).

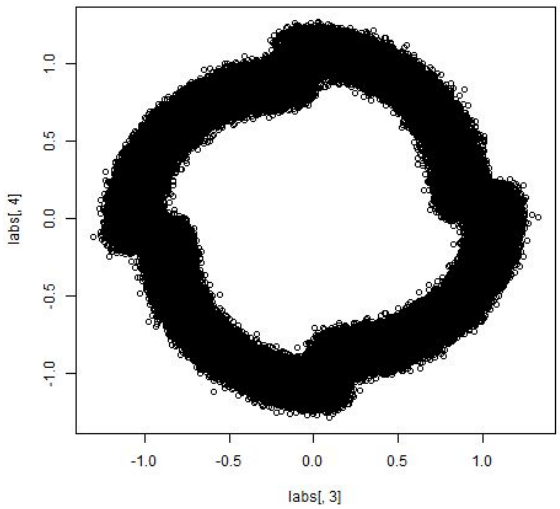


Figure 5: Location density of bot

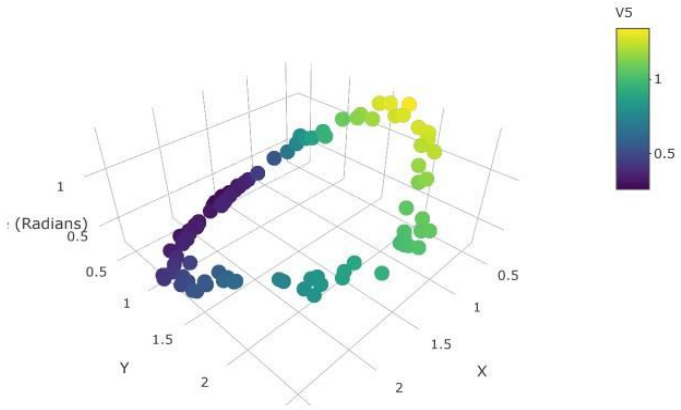


Figure 6: Bot location based on angle

The plots above reveals that the bot was essentially traveling in 4 arcs, each slightly offset from the others by a simple linear transformation. This formed the basis for the development of an approximate density function that is the essence of our continuous model. Furthermore, we notice that the average period of circulation decreases overtime (Figure 7). We did not take this into account in the discretized model and may have been a potential reason for the drop in accuracy when only using the last 60 data points of each run due to runtime issues related to running the Viterbi algorithm on all data points for each run

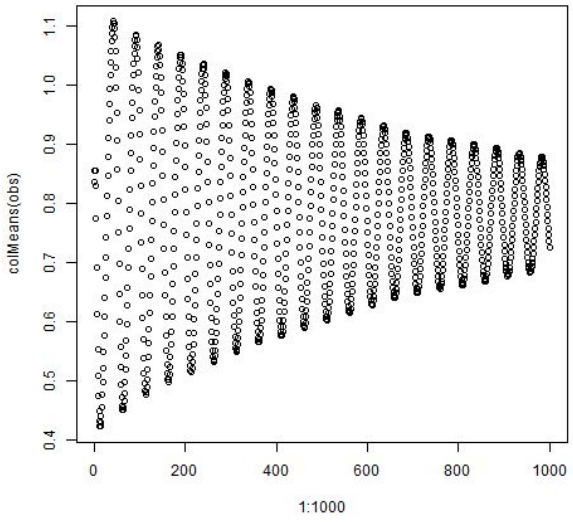


Figure 7: Average angle overtime

**Continuous Model**

Here we justify how our model handles the continuous case and justify our choice of parameters based on our understanding of the data described above. Our continuous HMM model is summarized in the graphical model below.

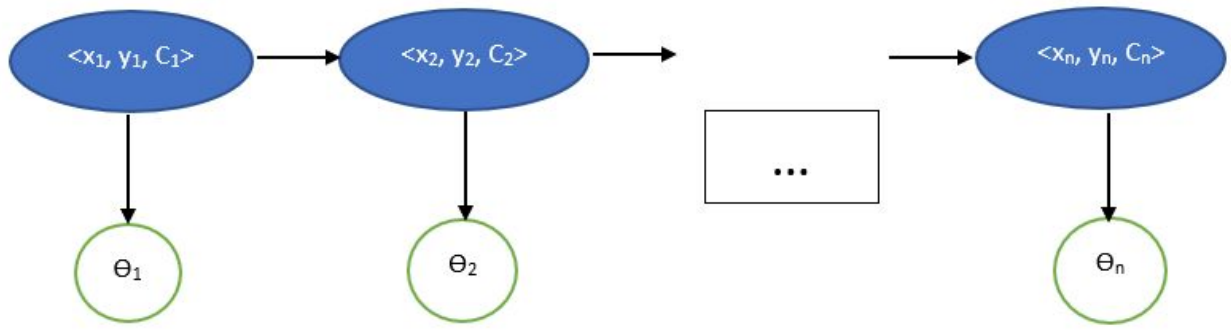


Figure 8: Graphic model for continuous HMM (where each state is the 3 element vector <x,y,c> described below)

Each state is abstracted as a 3 element vector containing a location(x and y coordinate) and a binary class c from {FRONT, BACK}, similar to the example from HW5, given by <x,y,c>. The class c assigned to a state is FRONT (red) if its coordinates are closer to the origin as shown in the diagram below and BACK otherwise (blue).

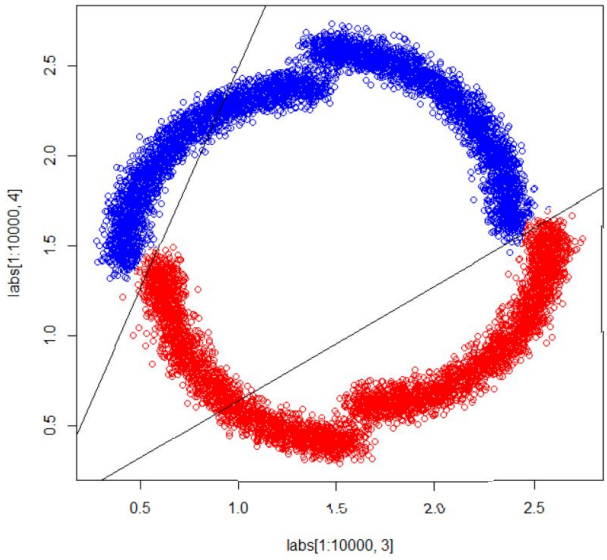


Figure 9: Coordinates by class

The possible values of x and y are only points along the periment of the 4 circles. The 4 circles are defined by a center C, offset O, and radius R where one circle is fixed at center C with

radius R and the rest are offset by O from the previous, also with radius R. The values of these 3 parameters were estimated through visualization. The figure below shows the result of our model, tested on the first run with all 1000 data points, to show the possible output coordinates.

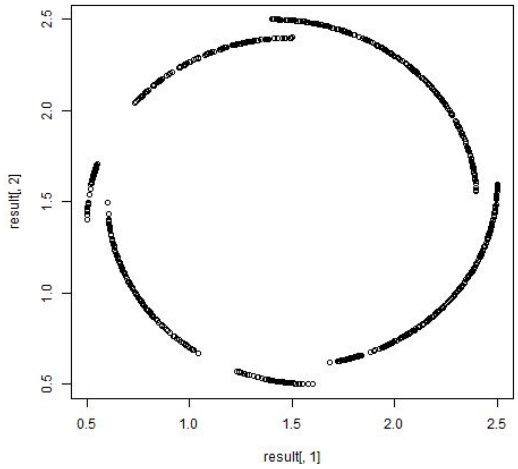


Figure 9: Output coordinate density example

To find the transition probabilities, we must find $P(x_t, y_t, c_t \mid x_{t-1}, y_{t-1}, c_{t-1})$. The intuition is that, knowing the coordinates of the last state and its class, we can determine the most probable coordinate pair and class for the current state. The density function that represents the probability of a coordinate pair is parameterized by angle θ, center C, offset O, and radius R, and the probability of a coordinate pair is simply 1/(#intersecting points) for each of the points where a line of angle θ intersects the 4 circles, limited to their respective quadrants, where one is described by C and R and the rest are just offset by O from one of the others. The equation we use for intersection is described using the code below:

```
#Calculation to find intersect
#x = x coordinate of center
#y = y coordinate of center
#r = radius
#slope = slope of line
a <- slope^2 + 1
b <- 2 * (0 - slope * y - x)
c <- y^2 - r^2 + x^2

bac <- b^2 - 4 * a * c

#line miss
if (bac < 0) {
    ...
```

```
} else {
    #The x and y for the 2 points of intersection
    x1 <- (-1 * b + sqrt(bac)) / (2 * a)
    x2 <- (-1 * b - sqrt(bac)) / (2 * a)
    y1 <- slope * x1
    y2 <- slope * x2
}
```

This returns a set of uniformly distributed states <x,y,c>. However, the likelihood of a point is still affected by the previous state's coordinate and class. Thus the most probable next state after transition should be the one with the closest euclidian distance to the previous state and have the same class. The probability of transitioning to a state with a different class is only non-zero for states with coordinates in the areas to the left and right of the lines in figure 9 above, but not between them. If a line does not intersect a circle (only occurs for extreme angles closest to 0 and 90), it is transformed such that it is tangent to its closest circle and the point of contact is treated as the only possible state. The equation we use for this process is described in the code below and is a continuation of the previous code block (where the ellipses are) with the same variables. The starting state is randomly chosen from among all the states with coordinates at the intersection of the line formed by the first angle at t=1 and the circles.

```
if (bac < 0) {
    #Find the point that is tangent to the circle
    x1 <- (x + slope * y) / (slope^2 + 1)
    y1 <- slope * (x + slope * y) / (slope^2 + 1)
    distToCen <- dist(rbind(c(x1,y1),c(x,y)))
    distToEdge <- distToCen - r
    xLine <- (x1 - x) * (distToEdge / distToCen)
    yLine <- (y - y1) * (distToEdge / distToCen)
    x1 <- x1 - xLine
    y1 <- y1 + yLine
}
```

It should be clear, as in HW5, that the emission probability for any observation is 1 because it is impossible to transition to a state that does not emit the observed angle. For example, if an angle observed is 0.8 radians at time t, there are only 2 possible states that the state at time t-1 can transition to and these states are the two with x and y coordinates that correspond to the intersection of the line with a slope of tan(0.8) and the circles.

**Results:**

First we run our model on the first couple of unabled examples and calculate the RMSE for each based on the labeled points. The purpose of doing this was to minimize Kaggle submissions by allowing us to evaluate the performance of our model on a smaller dataset. We then run this method on all 4000 runs for all data point (1:1000) observations. The result produces an accuracy of 0.29 on the private leaderboards. This serves as the baseline score that motivates decision for our discrete model.

**Discretized Model**

Learning from our previous model, we concluded that a simple HMM with roughly 20 states, as suggested in office hours, that can each be mapped to some coordinate on the perimeter of the circular path would produce similar results. Thus we implemented a 334 state model shown in the diagram below.

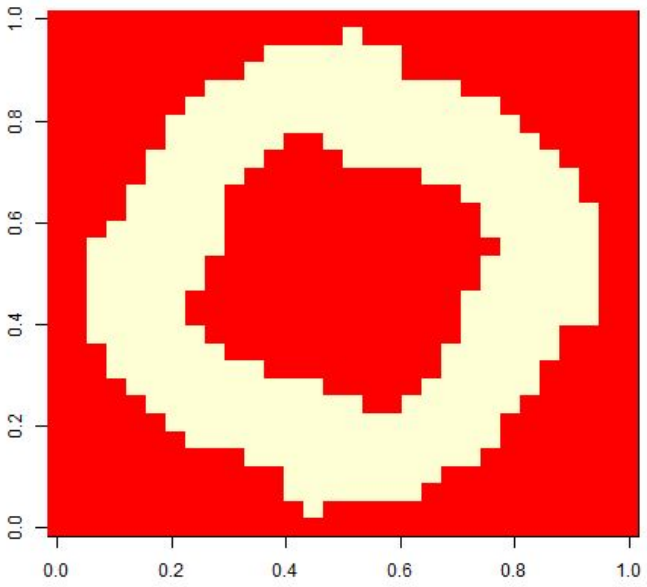


Figure 10: state space of discretized model

A summary of the graphical model is shown below:

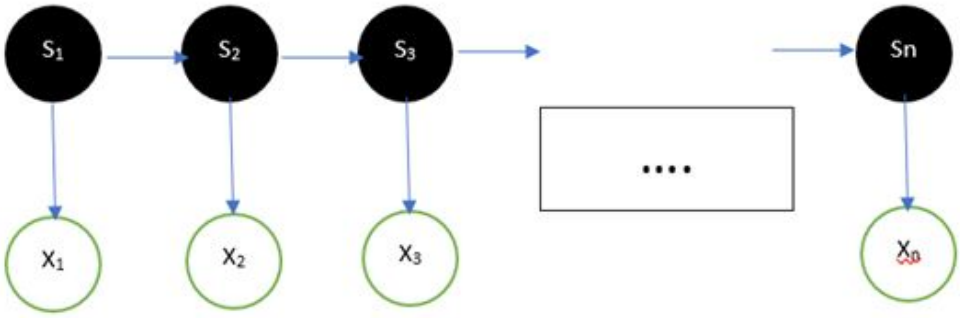


Figure 11: graphical model of discretized HMM; In this case the states are the cells, rather than a vector consisting of three different values. The observations are also discretized.

The figure 10 above shows a 30 x 30 grid, where each cell is a state. The red cells have no observed points in the labeled data, thus we remove these states reducing the state space from 900 to 334. We have also tried a 10x10 grid and 60x60 grid. The observations were also discretized by conversion to degrees and binned into 1, 0.5, OR 0.1 intervals (tried all 3). For example, if we binned in the angles into 0.5 intervals, there would be 181 possible observations per state (0 and 90 inclusive). Each state is mapped to the coordinate at its center (i.e. the center of the square).

First, we calculate maximum likelihood estimates for the emission probabilities for each state. This was obtained by simply dividing the total number of labeled points corresponding to a certain angle in a cell (obtained from the labeled data) by the total number of labeled points in the cell. This would give the emission probability of that angle from this state and the process is repeated for all state and all angles. It should be clear that the emission probabilities for each state sum up to 1 using this method, and the estimate is extremely precise. We apply add 1 smoothing to ensure that the probability of observing other angles at a particular state is not 0 to avoid overfitting.

Next, given this table of precisely initialized emission probabilities obtained using maximum likelihood estimation, we compute the transition probabilities using the Baum Welch algorithm for expectation maximization. The input emission probability table and starting probability vector were initialized to 1/(# of states) which in the case of a 30x30 grid, was 1/334. The input observations were all 1000 data points from the first run (discretized into the appropriate bins). Due to the large state space, the algorithm took an extremely long time (over 12 hours) to run and took over 250 iterations to converge. We repeated this experiment using only the last 100 data points and the total run time was about slightly under 2 hours with similar results. We used this technique when training the the 60x60 and 10x10 model.

**Results:**

With the results of Baum Welch, we constructed a completed HMM and used it to evaluate the first 10 labled runs using all 1000 data points for each by using the Viterbi algorithm to find the maximum likelihood path of states. Comparing to the labeled points, we achieve an average RMSE for 0.136. However, due to our large state space, it takes about 1 minute to run the Viterbi algorithm for each run. It would not be feasible to run our HMM on all 1000 data points for 4000 runs so we experimented using different subsets of point such as the last 100 and last 60. It took about 10 seconds to run using only 60 data points which is still not feasible for 4000 runs. Clearly using a 60x60 + reduction state space would not work, so we tried a 10x10 + reduction state space (54 states). This method achieve an accuracy of only 0.32 which is worse than our continuous model. Thus we conclude that our discretized 30x30 + reduction state HMM is a very admirable model in term of accuracy. However, due to runtime performance we could not submit its results to shine on the leaderboard.

In an attempt to devise a faster model, we turned to SVMs to learn the regression function that would allow us to predict a coordinate from several of the preceding angles. This method is described in the main model section above.