CS520 Project 3: Machine Learning - What Is It Good For?

Aditya Girish, Rishik Sarkar

November 10 2023

1 Model 1

This section is dedicated to our training and testing of Model 1, which is used to predict the best next move for the bot based on the current grid configuration. The subsections shall include the following essential information: the general pipeline of Model 1, a thorough analysis of the training process, and a presentation of the testing process along with any relevant results.

1.1 Model 1 Pipeline

I have outlined the general pipeline of Model 1 below, along with any crucial steps taken:

1. Import relevant packages and run the *Bot1.ipynb* notebook to gain access to all the setup functions used for Bot 1 in Project 2. None of the functions have been modified, and they allow us to set up the 30x30 grid, randomly place one bot, alien, and crew member, initialize alien and crew probability matrices, move the bot and aliens, update probability matrices, and deterministically determine the best move for the bot. Note that we fixed the ship's layout in terms of the default blocked and open cells, and only the locations of the bot, alien, and crew member change in any given simulation
2. Simulate Bot 1 similarly to how it was done in Project 2, but collect relevant feature and output vectors to save in a Pandas dataframe (and eventually store as *model_data_raw.csv* to be used in training Model 1. I shall go into more detail about the exact feature and output vectors in the next section.
   (a) To collect enough data, we ran 6000 bots (i.e., until the bot saved the crew or got captured): 30 iterations per simulation and 200 simulations
   (b) Through these simulations, we collected a total of 4,132,796 data points
   (c) The following hyperparameters were used: alpha = 0.004, k = 3
   (d) These were the metric results from the simulation:
   - Average Rescue Moves: 585.05
   - Probability of Crew Rescue: 0.71
   - Average Crew Saved: 21.3

(e) Note that these values only reflect the final simulation for our data collection phase. Since we modified the input and output vectors several times after the training/testing phase, our simulation hyperparameters, data size, and metrics constantly changed

3. Preprocess the collected data and perform train/test splits for the input matrix X and output vector y. We also generated a validity mask for the output vector at this pipeline step, but more later. The following general preprocessing steps were taken:

   (a) Read the data from *model_data_raw.csv* and ensure that the shape is consistent and the data is loading correctly

   (b) Remove duplicates from the data to streamline the dataset and remove redundancy. Without this step, the model might memorize frequently occurring data points or become skewed/biased. After this step, the total data points are reduced to 1,787,373

   (c) At this point, perform any necessary adjustments to the feature vectors (i.e., dropping/adding columns, normalizing data, etc.). Since we were confident about our features during our final collection process, we did not make any modifications to the collected data

   (d) Separate the data into the input matrix X and the output vector y (with the output being the last column and the input being all the other columns). Note that PCA or other feature engineering methods could also be performed in this step

   (e) Divide the X and y dataframes into train and test sets with an 80/20 split, respectively. At this point, we also checked that all five classes were equally represented in each set

   (f) Finally, generate the validity mask vector dataframes for the train and test sets. Each row in the validity mask corresponds to the same row in the X_train and X_test datasets, and essentially represents the neighbors that the bot can move to given the board's open and closed cells. The following process was used to generate this validity mask:

      i. For any given datapoint, use the bot_x and bot_y features (that represent the bot's current coordinates) along with an empty grid state to determine what neighbor cells count as valid moves for the bot

      ii. Each datapoint in the valid dataframes is represented as a list in the form of $[c_1, c_2, c_3, c_4, c_5]$ depicting the up, down, left, right, and bot cells, respectively. Each c_i is 0 if the corresponding relative neighbor is an invalid move and 1 if the bot can move to it. The significance of this vector will become clear in the next section

4. Train the logistic regression model on the X_train + y_train dataset and save the updated W and b values for Model 1. Since I will go into the input, output, model spaces as well as the learning techniques in the next section, this point is dedicated to discussing the final training hyperparameters:

   - alpha (learning rate) = 0.01
     - We additionally tried 0.001 and 0.05, but 0.01 seemed to be optimal. A value of 0.001 caused the model training to be prolonged, whereas 0.05 caused the loss to fluctuate due to overstepping

- epochs = 50
  - We tried several other epoch values, but 50 seemed an ideal number for the loss to decrease until it became asymptotic

5. Test Model 1 on the testing dataset with the updated W and b values and calculate the accuracy of predicting the best move. I shall discuss our findings in more detail in the Results section, but we generally saw no indication of overfitting and achieved decent training and testing accuracy

   - Additionally, to confirm that the decrease in loss caused an increase in accuracy (i.e., the model was improving), we calculated the training and testing accuracy using randomized weights and biases and averaged the results over 100 calculations. Overall, we were able to find a significant improvement in both training and testing accuracy using the trained weights than the randomized weights, thus indicating that the model truly learned

6. Simulate Model 1 through a new bot called Mimic-Bot1 that utilizes the logistic regression model's predictions to determine the next move for the bot by utilizing the same features as the trained model. Here is a general outline of the process:

   (a) Create the Mimic-Bot1 simulation function identical to the Bot1 simulation function, but instead of using the deterministic *determine_move()* function defined in *Bot1.ipynb*, predict the next move at each timestep by using Model 1 with a dataframe input containing features extracted from the current ship configuration (similar to the data collection function)

   (b) Test the Mimic-Bot1 simulation out against the Bot1 simulation function similar to how the testing was performed in Project 2 with Bot 1 and Bot 2, and store the collected average metrics for each bot

   (c) We ran 400 bots for both Bot 1 and Mimic-Bot1: 20 iterations per simulation and 20 simulations

   (d) The following hyperparameters were used: alpha = 0.004, k = 3

   (e) Importantly, we calculated a validity mask for every X input, and used a prediction function that applied the validity mask to the output to ensure that the bot would remain within bounds

   (f) Later, we added a stochastic element to the prediction provided by Model 1. In essence, we generated a random number between 1 and 5 and returned a random valid move if the number was 1 (i.e., 20% of the time). This modification accounted for situations where Mimic-Bot1 got stuck in an endless loop. Consider the following scenario: The bot is in a cell from which the highest probability neighbor is "up," but following the highest probability from "up," the bot is taken back to the original cell. By adding slight randomness, Mimic-Bot1 can escape such loops by taking unexpected actions

1.2 Training Process

1.2.1 Input Space

1.2.2 Output Space

1.2.3 Model Space

1.2.4 Loss

1.2.5 Training Algorithm

1.3 Testing Process and Results

2 Model 2

3 Model 3