PROJECT 3: MACHINE LEARNING

Submitted By
William Chandler (wcc44)
Vignesh Krishnan (vk505)
Rohan Tripathi (rt767)

Introduction

This project uses machine learning to analyze the actions taken by a bot that utilizes the approach of Bot 1 crafted in project 2. For this purpose, we instantiate an ACTOR model and a CRITIC model. After training, these models are applied to the bot design and the result is compared with the original design of Bot 1. The detector size and α used are 2 and 0.09. We have also fixed the layout of our ship and thus restricted the learning process to focus on this particular layout.

Data Generation

The first step to creating these ACTOR/CRITIC models is preparing the dataset that these models will be trained on. We have identified the possible relevant attributes to the problem and added them to a dataframe which serves as the fundamental dataset for our models. The attributes are listed here:

- Current bot cell location at each step
- The neighboring cells (categorical values for UP, DOWN etc and UNREACHABLE if the cell is not open)
- Alien Probabilities for these neighbors (-1 if cell is closed)
- Crewmate Probabilities for these neighbors (-1 if cell is closed)
- X and Y coordinates for the max crewmate probability cell at any time step
- The X and Y of the next position of the bot (our target)

The combination of these attributes was chosen as the dataset because this set completely describes all the information available to the bot at any time-step. Thus, this represents the complete state of our input at any point in time. While some of these attributes may be redundant, we can always trim down attributes during feature selection when conducting our correlation analysis.

Overall, across multiple simulations, we have collected a train-test data set of 89,194 rows which has diverse target class values to ensure unbiased training. We split this data randomly while maintaining the rescued/caught ratio. Taking the X and Y coordinates separately, we have a total of 26 attributes for model 1.

Feature Selection

To identify attributes that are most relevant to the learning process of the models, we get rid of redundant or unnecessary attributes by performing Lasso's L1 regularization and snipping out the attributes whose coefficients reduce to zero. The results of regularization indicated that most of our attributes were indeed relevant. Interestingly, according to Lasso's regularization, the alien beep and crewmate beep boolean attributes can be safely pruned when formulating the feature set, most likely because the probabilities give a fairly clear indication of the state of these sensors. Thus, based on Lasso's regularization, our feature set is:

- Current bot cell location at each step
- The X and Y coordinates for each neighbor of the current bot cell
- Alien Probabilities for these neighbors (-1 if the cell is closed)
- Crewmate Probabilities for these neighbors (-1 if the cell is closed)
- X and Y coordinates for the max crewmate probability cell at any time step

Model 1 Description

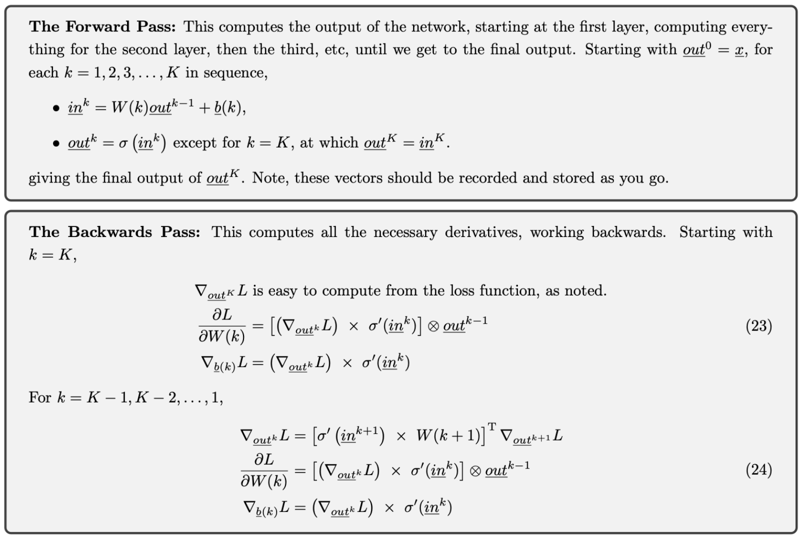
Input Space:
Our input space for model 1 consists of a 1-Dimensional vector of size 26 for each time-step in a particular simulation. This also includes the target attribute in our case which can be one of 5 values: UP, DOWN, LEFT, RIGHT and STAY.

Output Space:
The output space for model 1 will be a 1-Dimensional vector of size 5 (the number of target classes), consisting only of probability values corresponding to each of the target classes which indicate the likelihood of the bot rescuing the crewmate if it chose this particular move.

Model Space:
We use a Deep Learning based Neural Network Model with an input layer, an intermediate layer of size 100 and an output layer which gives 5 outputs - probabilities corresponding to the 5 prediction classes for our actor model. The maximum of these

probabilities is then mapped to the value 1 while the rest are set to 0 indicating that the max probability class is the one predicted by our model. The mathematical layout of the layers and forward pass and back propagation is formulated below:

- $\underline{out}^0 = \underline{x}$
- $\underline{out}^1 = \sigma(W(1)\underline{out}^0 + \underline{b}(1))$
- $\underline{out}^3 = \sigma(W(3)\underline{out}^2 + \underline{b}^2)$
- $\ldots$
- $\underline{out}^{K-1} = \sigma(W(K-1)\underline{out}^{K-2} + \underline{b}(K-1))$
- $\underline{out}^K = W(K)\underline{out}^{K-1} + \underline{b}(K)$


The Forward Pass: This computes the output of the network, starting at the first layer, computing everything for the second layer, then the third, etc, until we get to the final output. Starting with $\underline{out}^0 = \underline{x}$, for each $k = 1, 2, 3, \ldots, K$ in sequence,

- $\underline{in}^k = W(k)\underline{out}^{k-1} + \underline{b}(k)$,
- $\underline{out}^k = \sigma\left(\underline{in}^k\right)$ except for $k = K$, at which $\underline{out}^K = \underline{in}^K$.

giving the final output of $\underline{out}^K$. Note, these vectors should be recorded and stored as you go.

The Backwards Pass: This computes all the necessary derivatives, working backwards. Starting with $k = K$,

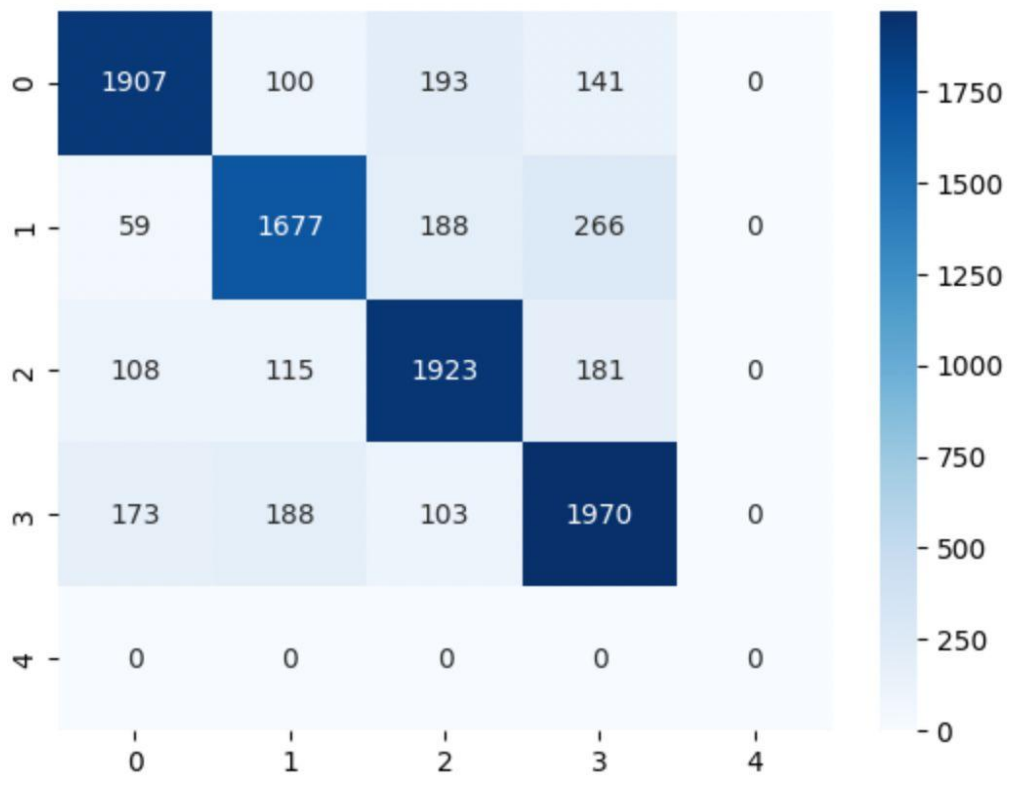
$$\begin{aligned} &\nabla_{\underline{out}^K} L \text{ is easy to compute from the loss function, as noted.} \\ &\frac{\partial L}{\partial W(k)} = \left[\left(\nabla_{\underline{out}^k} L\right) \times \sigma'(\underline{in}^k)\right] \otimes \underline{out}^{k-1} \\ &\nabla_{\underline{b}(k)} L = \left(\nabla_{\underline{out}^k} L\right) \times \sigma'(\underline{in}^k) \end{aligned} \tag{23}$$

For $k = K-1, K-2, \ldots, 1$,

$$\begin{aligned} \nabla_{\underline{out}^k} L &= \left[\sigma'\left(\underline{in}^{k+1}\right) \times W(k+1)\right]^{\mathrm{T}} \nabla_{\underline{out}^{k+1}} L \\ \frac{\partial L}{\partial W(k)} &= \left[\left(\nabla_{\underline{out}^k} L\right) \times \sigma'(\underline{in}^k)\right] \otimes \underline{out}^{k-1} \\ \nabla_{\underline{b}(k)} L &= \left(\nabla_{\underline{out}^k} L\right) \times \sigma'(\underline{in}^k) \end{aligned} \tag{24}$$

We also tested our model after adding some max pooling and convolutional layers, but neither the accuracy nor the consistency of the model showed any appreciable improvement across multiple tests. Infact, the accuracy even dropped at times and the time taken by the model to train also increased significantly. Given these problems, we restricted our model to use hidden and dropout layers only, which provided us with a fairly impressive accuracy of 80% for the parameters detailed above.

Loss Function:
We perform a comparison of the action taken by Bot 1 at a time step, and the corresponding prediction by our ACTOR model to measure the loss of our model in terms of categorical cross-entropy. We also plot a confusion matrix to compare the predictions with the actual actions taken. The values 0,1,2,3 and 4 are UP, DOWN, LEFT and RIGHT respectively. The matrix is illustrated below:

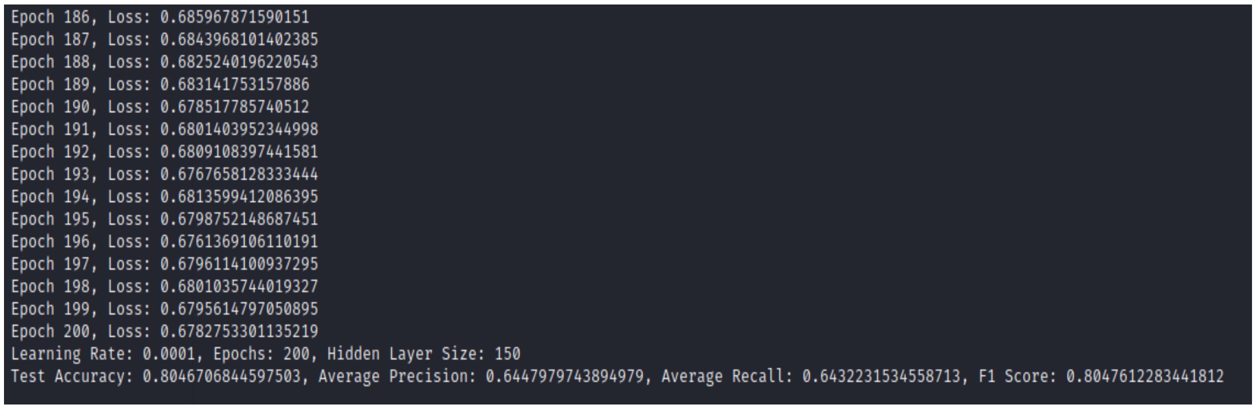


	0	1	2	3	4
0	1907	100	193	141	0
1	59	1677	188	266	0
2	108	115	1923	181	0
3	173	188	103	1970	0
4	0	0	0	0	0

Training Results:
To come up with the best combination of parameters, we performed a grid search varying over different numbers of epochs, learning rates and the hidden layer sizes. The parameters for the search were:

learning_rates = [0.0001, 0.0005, 0.001]
epoch_numbers = [100,200,300]
hidden_layer_sizes = [50, 100, 150]

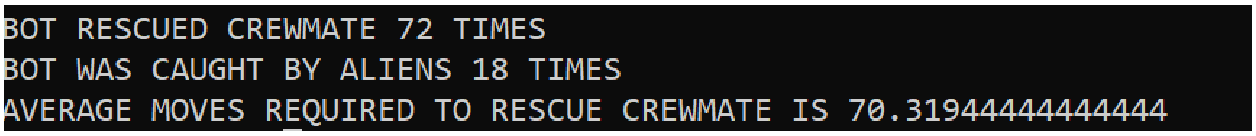
We picked the best set of values out of these based on an average of the accuracy and F1 scores.
We trained our model at a learning rate of 0.0001 over 200 epochs with the layer configuration described under the model space and arrived at an accuracy of 80% over the test data. The results showed an almost negligible difference between accuracy on training and test data. This suggests that the model is most likely not an overfit as it shows no exceptional performance on training data. Further, the size of the database is also large enough to stabilize any biases or skews in class representation (which we normalize nevertheless).



```
Epoch 186, Loss: 0.685967871590151
Epoch 187, Loss: 0.6843968101402385
Epoch 188, Loss: 0.6825240196220543
Epoch 189, Loss: 0.683141753157886
Epoch 190, Loss: 0.678517785740512
Epoch 191, Loss: 0.6801403952344998
Epoch 192, Loss: 0.6809108397441581
Epoch 193, Loss: 0.6767658128333444
Epoch 194, Loss: 0.6813599412086395
Epoch 195, Loss: 0.6798752148687451
Epoch 196, Loss: 0.6761369106110191
Epoch 197, Loss: 0.6796114100937295
Epoch 198, Loss: 0.6801035744019327
Epoch 199, Loss: 0.6795614797050895
Epoch 200, Loss: 0.6782753301135219
Learning Rate: 0.0001, Epochs: 200, Hidden Layer Size: 150
Test Accuracy: 0.8046706844597503, Average Precision: 0.6447979743894979, Average Recall: 0.6432231534558713, F1 Score: 0.8047612283441812
```

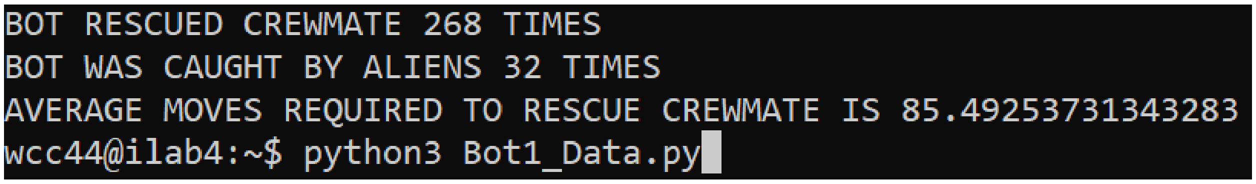
A comparative analysis of Bot 1 and our Model 1 based bot is given here:

151 Simulations ACTOR:



```
BOT RESCUED CREWMATE 72 TIMES
BOT WAS CAUGHT BY ALIENS 18 TIMES
AVERAGE MOVES REQUIRED TO RESCUE CREWMATE IS 70.31944444444444
```

300 Simulations Bot 1:



```
BOT RESCUED CREWMATE 268 TIMES
BOT WAS CAUGHT BY ALIENS 32 TIMES
AVERAGE MOVES REQUIRED TO RESCUE CREWMATE IS 85.49253731343283
wcc44@ilab4:~$ python3 Bot1_Data.py
```

Thus, we see that the actions recommended by bot 1 do reduce the average number of time steps that are required by the bot to rescue the crewmate.

Model 2 Description

Input Space:
The input space for model 2 is the same as model 1. The math involved is also explained along with model 1.

Output Space:
We append an additional attribute aimed to provide model 2 with an indication of the likelihood of success corresponding to a move in each simulation. To do this, we use the following strategy:

- If the bot successfully rescued a crewmate at the end of a simulation, we append a "RESCUED" value for this attribute to every time step of that simulation.
- If the bot was caught by the alien, we append a "CAUGHT" value to this attribute for every time step of that simulation.

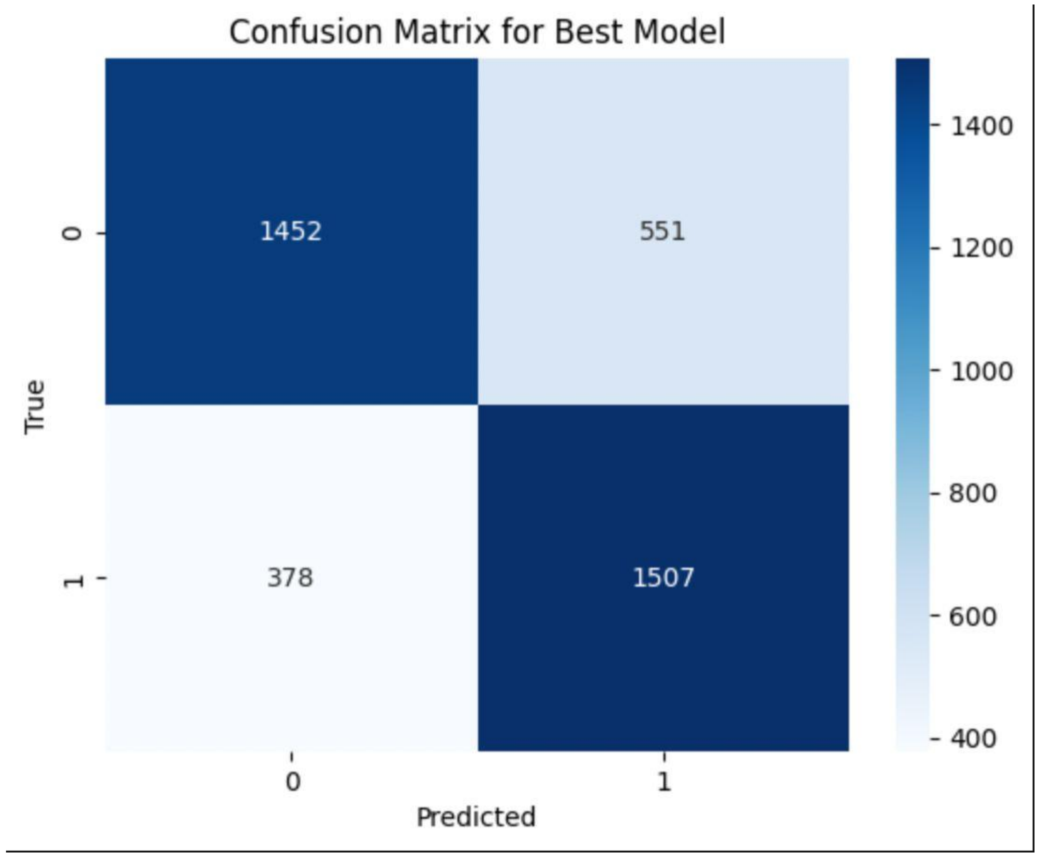
This allows the model to learn from the cases where the bot was able to rescue the crewmate. We do note that there might be cases where the bot was lucky and the alien’s movement was favorable enough to let it survive, our model could be learning undesirable patterns. But we rely on the sheer volume of our data to ensure that such cases are an insignificant part of the train-test data and will not impact our model significantly.

Our CRITIC model (model 2) will be providing 2 classes for probability of rescuing a crewmate or getting caught as an output in response to each time step it receives. That is - the output corresponding to a row of test data would be a float value pair between 0 and 1, indicating chances of rescuing the crewmate.

Model Space:
The overall layout of the layers for the neural network is identical to the ACTOR model. And the output layer only gives 2 output labels now.

Balancing the Rescued and Caught cases:
To make sure that a particular class is not over-represented in our data, we wait till the under-represented class instances (“caught” verdicts) have reached a certain threshold before we continue appending instances of our over-represented instance (“rescued” verdict) to our dataset.

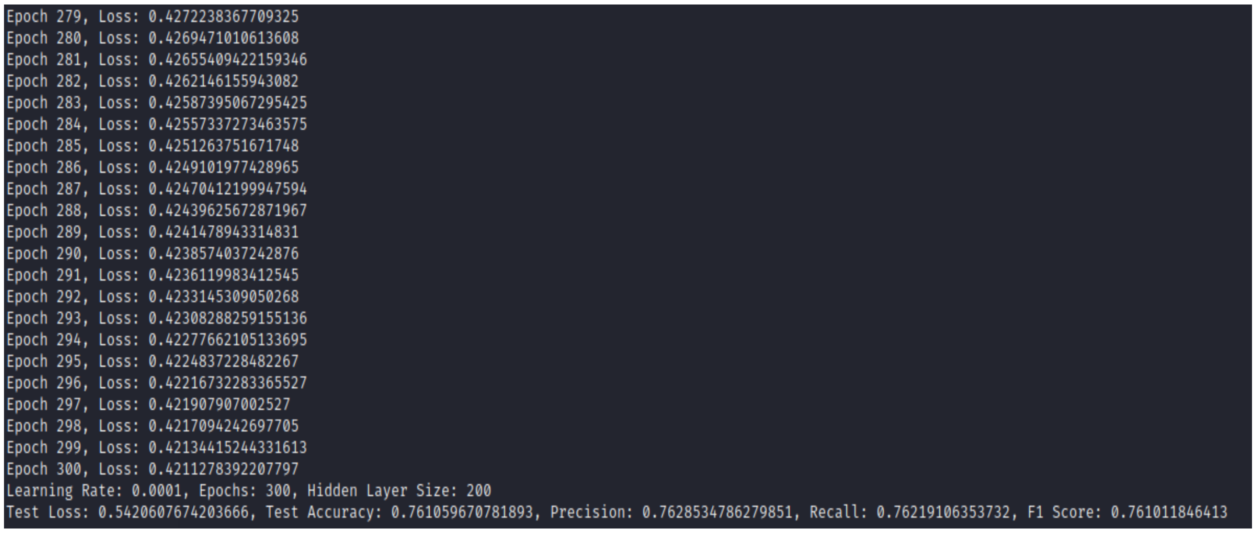
Loss Function:
Unlike model 1 where we were calculating losses in a multiclass scenario, in model 2, we only have 2 output classes. Thus, we have shifted to binary cross entropy as we predict labels “RESCUED” and “CAUGHT” which will have 1 and 0 values respectively. We also visualize a confusion matrix for the performance of model 2 on test data below. The values 0 and 1 represent “caught” and “rescued” respectively.



Training Results:
Again we conduct a grid search with:

learning_rates = [0.001, 0.0001, 0.0005]
epoch_numbers = [200, 100, 300]
hidden_layer_sizes = [200, 100]

Model 2 was trained on a learning rate of 0.0001 over 300 epochs and it yields an accuracy of 76% on the test data. Again, since the testing and training loss values are almost comparable, we do not have any reason to suspect overfitting.



```
Epoch 279, Loss: 0.4272238367709325
Epoch 280, Loss: 0.4269471010613608
Epoch 281, Loss: 0.42655409422159346
Epoch 282, Loss: 0.4262146155943082
Epoch 283, Loss: 0.42587395067295425
Epoch 284, Loss: 0.42557337273463575
Epoch 285, Loss: 0.4251263751671748
Epoch 286, Loss: 0.4249101977428965
Epoch 287, Loss: 0.42470412199947594
Epoch 288, Loss: 0.42439625672871967
Epoch 289, Loss: 0.4241478943314831
Epoch 290, Loss: 0.4238574037242876
Epoch 291, Loss: 0.4236119983412545
Epoch 292, Loss: 0.4233145309050268
Epoch 293, Loss: 0.42308288259155136
Epoch 294, Loss: 0.42277662105133695
Epoch 295, Loss: 0.4224837228482267
Epoch 296, Loss: 0.42216732283365527
Epoch 297, Loss: 0.421907907002527
Epoch 298, Loss: 0.4217094242697705
Epoch 299, Loss: 0.42134415244331613
Epoch 300, Loss: 0.4211278392207797
Learning Rate: 0.0001, Epochs: 300, Hidden Layer Size: 200
Test Loss: 0.5420607674203666, Test Accuracy: 0.761059670781893, Precision: 0.7628534786279851, Recall: 0.76219106353732, F1 Score: 0.761011846413
```

<u>**Model 3 Description**</u>

Input Space:
The input space of the model is initially the same as models 1 and 2. However, the predictions from these models form a feedback loop where the output of the actor loop is processed and fed to the critic and the out from the critic in turn is fed to the actor.

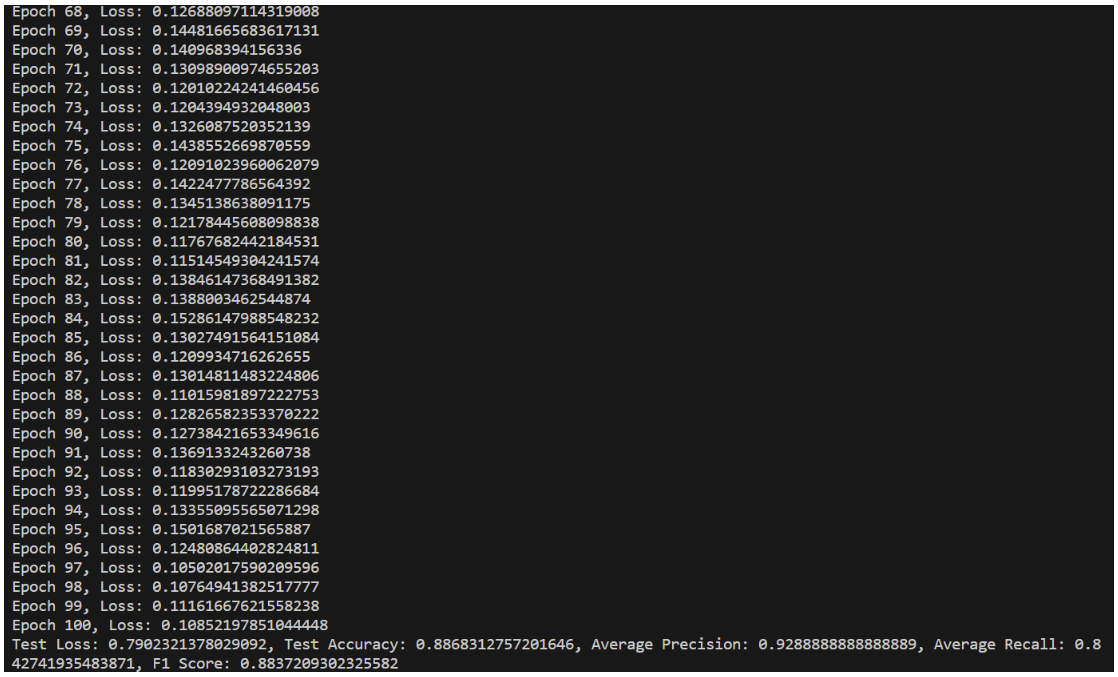
Model Space:
The model space is similar to the original structure for model 1 and 2. The main difference is that the data being fed for training is coming from different sources now. The ACTOR model for model 3 uses the data from the initial data set of 89194 rows obtained by running bot 1 on 1000 simulations. We are training the actor bot to predict the action taken by bot1 at each time step based on the features given. After training for the ACTOR model is completed, it now mimics bot 1's movements reasonably accurately. We generate new data for the CRITIC model by using action predictions from the ACTOR model until we get a balanced ratio for all outcomes, culminating in a data set of 40,000 rows (each simulation was limited to 150 time steps to restrict the data to 40000 rows). This data (which is now appended with the actions predicted by the actor model as a feature as well as the RESCUED/CAUGHT field as added to model 2 data) is then fed to the CRITIC loop for training with a training test split of 60:40. The critic is then trained to predict the percentage change of the final result being either Rescued or Caught based on the feature set. After testing is complete 4 rows are added to the test data for every row with all features remaining the same except the action taken changes in each new row. This data set is then fed to the critic giving us a data set at 16,000 possible states with the percentage rescued afor each possible

action at that state. This data is then used to create a new target column for the actor model of the action at each state with the highest percentage chance of rescuing the crewmate. Thus, instead of mimicking bot 1, the ACTOR loop will now output the action that is most likely to result in a successful rescue as predicted by the CRITIC model.

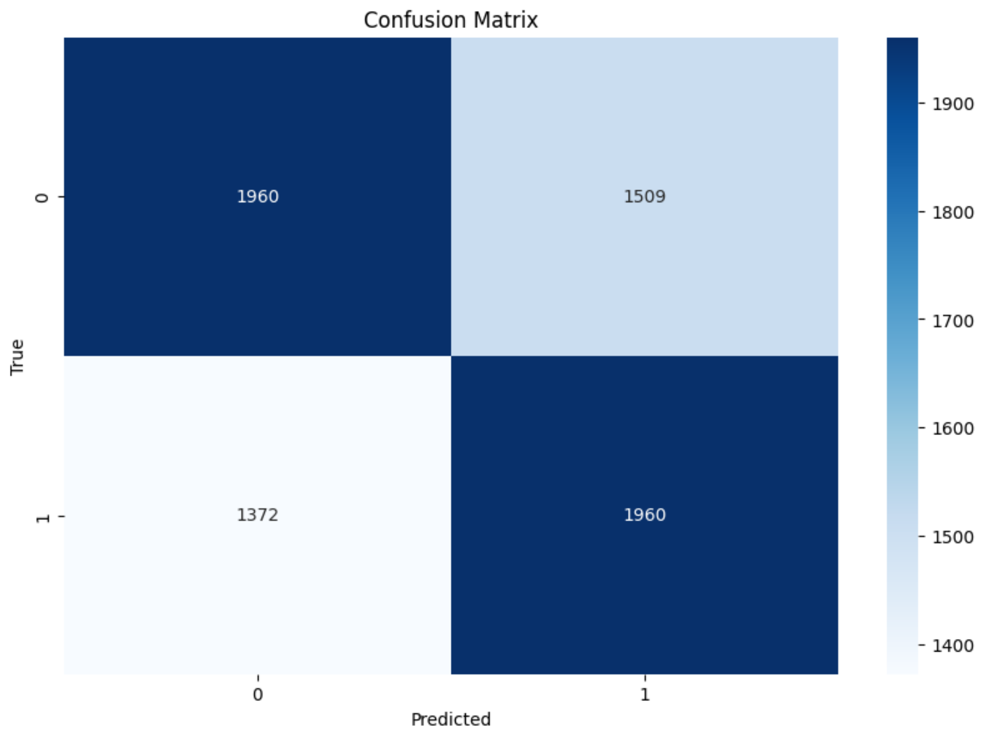
Once we had our ACTOR model ready to output the best move at any time step (the move most likely to result in a successful rescue of the crewmate), we simulated bot 1 based on this data to see if it actually resulted in a success. Unfortunately, the model always suggests that the bot move in a single direction until it runs into a wall.
Given the unsuccessful results, we did not perform a feedback loop over the ACTOR and CRITIC models.

Loss Function:
The loss functions for model 1 and 2 are used for the actor and critic evaluation respectively.

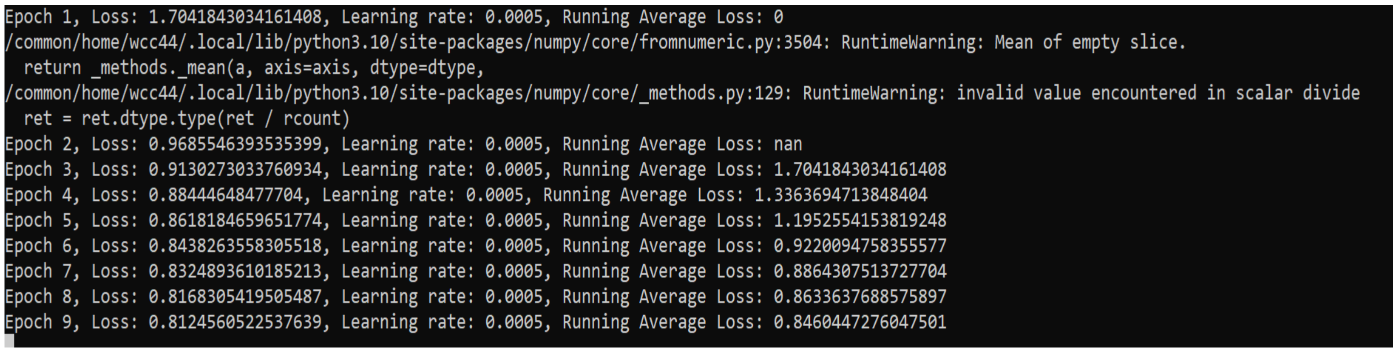
Insights:
We see that the CRITIC network is consistently learning from the performance of the actor network as the loss in training of the CRITIC network continuously drops throughout the training as can be seen below:



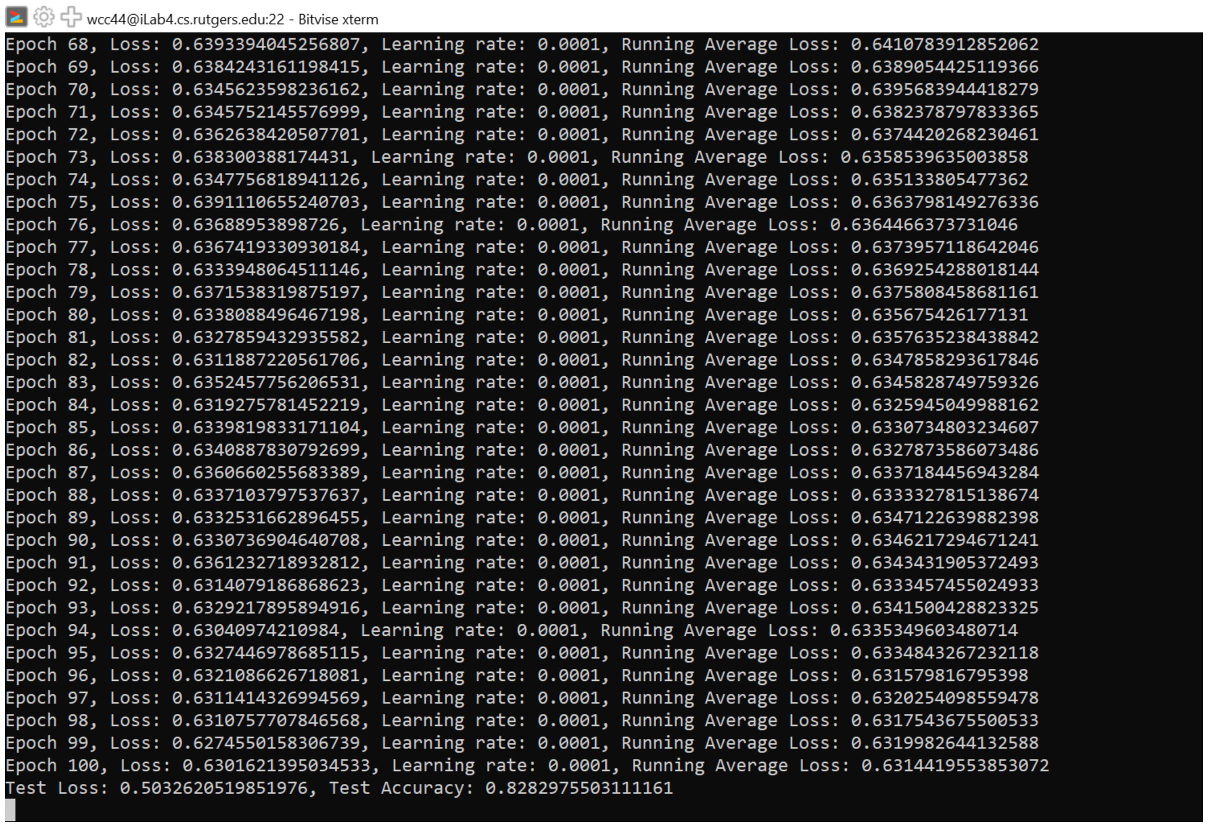
```
Epoch 68, Loss: 0.12688097114319008
Epoch 69, Loss: 0.14481665683617131
Epoch 70, Loss: 0.140968394156336
Epoch 71, Loss: 0.13098900974655203
Epoch 72, Loss: 0.12010224241460456
Epoch 73, Loss: 0.1204394932048003
Epoch 74, Loss: 0.1326087520352139
Epoch 75, Loss: 0.1438552669870559
Epoch 76, Loss: 0.12091023960062079
Epoch 77, Loss: 0.1422477786564392
Epoch 78, Loss: 0.1345138638091175
Epoch 79, Loss: 0.12178445608098838
Epoch 80, Loss: 0.11767682442184531
Epoch 81, Loss: 0.11514549304241574
Epoch 82, Loss: 0.13846147368491382
Epoch 83, Loss: 0.1388003462544874
Epoch 84, Loss: 0.15286147988548232
Epoch 85, Loss: 0.13027491564151084
Epoch 86, Loss: 0.1209934716262655
Epoch 87, Loss: 0.13014811483224806
Epoch 88, Loss: 0.11015981897222753
Epoch 89, Loss: 0.12826582353370222
Epoch 90, Loss: 0.12738421653349616
Epoch 91, Loss: 0.1369133243260738
Epoch 92, Loss: 0.11830293103273193
Epoch 93, Loss: 0.11995178722286684
Epoch 94, Loss: 0.13355095565071298
Epoch 95, Loss: 0.1501687021565887
Epoch 96, Loss: 0.12480864402824811
Epoch 97, Loss: 0.10502017590209596
Epoch 98, Loss: 0.10764941382517777
Epoch 99, Loss: 0.11161667621558238
Epoch 100, Loss: 0.10852197851044448
Test Loss: 0.7902321378029092, Test Accuracy: 0.8868312757201646, Average Precision: 0.9288888888888889, Average Recall: 0.8
42741935483871, F1 Score: 0.8837209302325582
```



In turn, we can also show that the ACTOR network is learning from the suggestions given by the CRITIC network through the analysis of loss in training as shown:

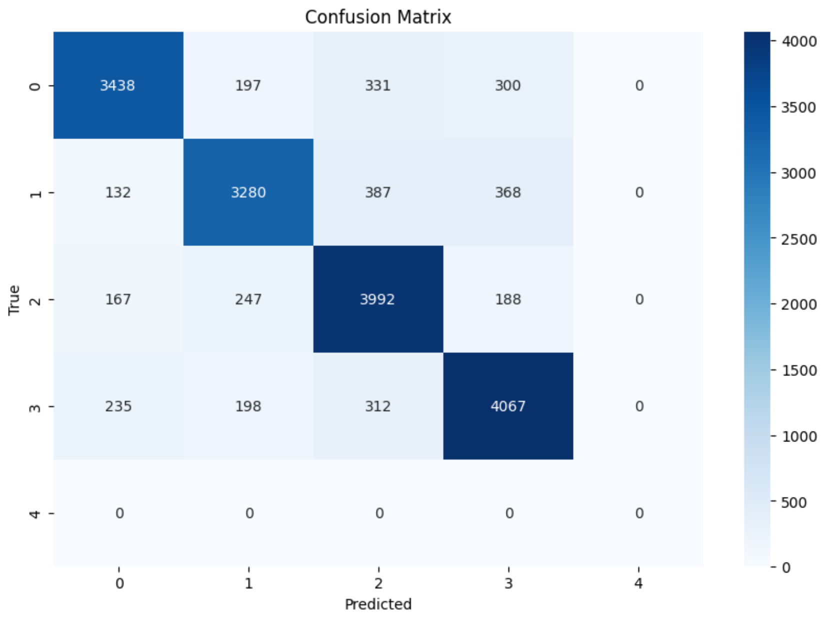


```
Epoch 1, Loss: 1.7041843034161408, Learning rate: 0.0005, Running Average Loss: 0
/common/home/wcc44/.local/lib/python3.10/site-packages/numpy/core/fromnumeric.py:3504: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/common/home/wcc44/.local/lib/python3.10/site-packages/numpy/core/_methods.py:129: RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
Epoch 2, Loss: 0.9685546393535399, Learning rate: 0.0005, Running Average Loss: nan
Epoch 3, Loss: 0.9130273033760934, Learning rate: 0.0005, Running Average Loss: 1.7041843034161408
Epoch 4, Loss: 0.88444648477704, Learning rate: 0.0005, Running Average Loss: 1.3363694713848404
Epoch 5, Loss: 0.8618184659651774, Learning rate: 0.0005, Running Average Loss: 1.1952554153819248
Epoch 6, Loss: 0.8438263558305518, Learning rate: 0.0005, Running Average Loss: 0.9220094758355577
Epoch 7, Loss: 0.8324893610185213, Learning rate: 0.0005, Running Average Loss: 0.8864307513727704
Epoch 8, Loss: 0.8168305419505487, Learning rate: 0.0005, Running Average Loss: 0.8633637688575897
Epoch 9, Loss: 0.8124560522537639, Learning rate: 0.0005, Running Average Loss: 0.8460447276047501
```



wcc44@iLab4.cs.rutgers.edu:22 - Bitvise xterm

```
Epoch 68, Loss: 0.6393394045256807, Learning rate: 0.0001, Running Average Loss: 0.6410783912852062
Epoch 69, Loss: 0.6384243161198415, Learning rate: 0.0001, Running Average Loss: 0.6389054425119366
Epoch 70, Loss: 0.6345623598236162, Learning rate: 0.0001, Running Average Loss: 0.6395683944418279
Epoch 71, Loss: 0.6345752145576999, Learning rate: 0.0001, Running Average Loss: 0.6382378797833365
Epoch 72, Loss: 0.6362638420507701, Learning rate: 0.0001, Running Average Loss: 0.6374420268230461
Epoch 73, Loss: 0.638300388174431, Learning rate: 0.0001, Running Average Loss: 0.6358539635003858
Epoch 74, Loss: 0.6347756818941126, Learning rate: 0.0001, Running Average Loss: 0.635133805477362
Epoch 75, Loss: 0.6391110655240703, Learning rate: 0.0001, Running Average Loss: 0.6363798149276336
Epoch 76, Loss: 0.63688953898726, Learning rate: 0.0001, Running Average Loss: 0.6364466373731046
Epoch 77, Loss: 0.6367419330930184, Learning rate: 0.0001, Running Average Loss: 0.6373957118642046
Epoch 78, Loss: 0.6333948064511146, Learning rate: 0.0001, Running Average Loss: 0.6369254288018144
Epoch 79, Loss: 0.6371538319875197, Learning rate: 0.0001, Running Average Loss: 0.6375808458681161
Epoch 80, Loss: 0.6338088496467198, Learning rate: 0.0001, Running Average Loss: 0.635675426177131
Epoch 81, Loss: 0.6327859432935582, Learning rate: 0.0001, Running Average Loss: 0.6357635238438842
Epoch 82, Loss: 0.6311887220561706, Learning rate: 0.0001, Running Average Loss: 0.6347858293617846
Epoch 83, Loss: 0.6352457756206531, Learning rate: 0.0001, Running Average Loss: 0.6345828749759326
Epoch 84, Loss: 0.6319275781452219, Learning rate: 0.0001, Running Average Loss: 0.6325945049988162
Epoch 85, Loss: 0.6339819833171104, Learning rate: 0.0001, Running Average Loss: 0.6330734803234607
Epoch 86, Loss: 0.6340887830792699, Learning rate: 0.0001, Running Average Loss: 0.6327873586073486
Epoch 87, Loss: 0.6360660255683389, Learning rate: 0.0001, Running Average Loss: 0.6337184456943284
Epoch 88, Loss: 0.6337103797537637, Learning rate: 0.0001, Running Average Loss: 0.6333327815138674
Epoch 89, Loss: 0.6332531662896455, Learning rate: 0.0001, Running Average Loss: 0.6347122639882398
Epoch 90, Loss: 0.6330736904640708, Learning rate: 0.0001, Running Average Loss: 0.6346217294671241
Epoch 91, Loss: 0.6361232718932812, Learning rate: 0.0001, Running Average Loss: 0.6343431905372493
Epoch 92, Loss: 0.6314079186868623, Learning rate: 0.0001, Running Average Loss: 0.6333457455024933
Epoch 93, Loss: 0.6329217895894916, Learning rate: 0.0001, Running Average Loss: 0.6341500428823325
Epoch 94, Loss: 0.63040974210984, Learning rate: 0.0001, Running Average Loss: 0.6335349603480714
Epoch 95, Loss: 0.6327446978685115, Learning rate: 0.0001, Running Average Loss: 0.6334843267232118
Epoch 96, Loss: 0.6321086626718081, Learning rate: 0.0001, Running Average Loss: 0.631579816795398
Epoch 97, Loss: 0.6311414326994569, Learning rate: 0.0001, Running Average Loss: 0.6320254098559478
Epoch 98, Loss: 0.6310757707846568, Learning rate: 0.0001, Running Average Loss: 0.6317543675500533
Epoch 99, Loss: 0.6274550158306739, Learning rate: 0.0001, Running Average Loss: 0.6319982644132588
Epoch 100, Loss: 0.6301621395034533, Learning rate: 0.0001, Running Average Loss: 0.6314419553853072
Test Loss: 0.5032620519851976, Test Accuracy: 0.8282975503111161
```



Conclusion

Model 1 and Model 2 utilize machine learning to draw fairly precise inferences regarding the actions the bot takes and the impact they have on its success. Model 3 does not improve the performance of the bot in our case, but the actor and critic network do seem to be learning from the feedback they get in each iteration. Thus, applying machine learning to the problem does seem to help replicate the behavior of the bot accurately. It also potentially improves the results, although our implementation could not completely reflect this in practice.