Tutorial 1 Report

Machine Learning I

Group 97

Full Name	Student ID
Abdelaty Rehab	100529701
Marc Dotto	100529268
Pranav Pudu	100530377

UC3M, April 8, 2024

1 | Phase 1

This section covers the procedures followed as well as the conclusions reached while doing the first phase of the practice.

1.1 | Feature Extraction Process

The feature extraction process in the snake game involves computing various attributes that describe the current state of the game. These attributes are then used as input for the machine learning model. The process consists of the following steps:
The function `compute_game_state` computes the state of the game based on the positions of the snake's head and the food, the current direction of the snake, and the distances between the snake's head and the food in the x and y directions. Additionally, it calculates the available space in each direction from the snake's head to the boundaries of the game window.
The extracted features include:

- **Snake Head Position**: $(snake\_head\_x, snake\_head\_y)$
- **Food Position**: $(food\_x, food\_y)$
- **Distances to Food**: $x\_distance = food\_x - snake\_head\_x$ and $y\_distance = food\_y - snake\_head\_y$
- **Available Space**: $(up\_space, down\_space, left\_space, right\_space)$ represents the space available in the four cardinal directions till the snake head hits the frame boundaries and they are computed as follows.

$$\begin{aligned} up\_space &= snake\_head\_y \\ down\_space &= FRAME\_SIZE\_Y - snake\_head\_y - 10 \\ left\_space &= snake\_head\_x \\ right\_space &= FRAME\_SIZE\_X - snake\_head\_x - 10 \end{aligned}$$

  Since each snake body part occupies 10 frames, this explains how the $down_space$ and $right_space$ are computed.

- **Can Move**: $(can\_up, can\_down, can\_left, can\_right)$ a set of boolean variables that represent whether a snake can move in a certain direction or not.
- **Snake Body Grid**: represents as a grid of binary attributes (currently $6x6$)The game window is divided into a grid, and the function checks whether any part of the snake's body lies within each cell of the grid. This information is represented as binary attributes in the `snake_body_grid`. For instance, $snake\_body\_grid[1][1] = Y$(as in "YES") indicates that there is a body part of the snake that lies in $cell(2, 2)$ of the grid. This method was used as a simple way to encode the snake body parts without explicitly printing them with the feature set. Since the snake body parts are an array of arrays of length 2 in the form of $[x, y]$, it was extremely hard to include them with the features extracted since Weka does not support that data type. It was needed to print $FRAME\_SIZE\_X * FRAME\_SIZE\_Y$ attributes where each one is a nominal of value either $True$ or $False$ representing whether that location on the game frame is a snake body part or not. This was extremely inefficient to do since it would take a very long time to extract the features as well as to train the model on the dataset.

These features provide a comprehensive representation of the current state of the game, which can be used for training the machine learning model.

1.2 | Future Data Incorporation

Since the model is required to predict the next move to be taken based on the current game state, the feature set provided to the model should contain relevant attributes that represent the **current** game state (as in Figure 1.1. On the other hand, the dataset on which the model will be trained should contain instances of the feature set in addition to the next move the snake will take optimally.

To include the next move in the extracted features, it was necessary to delay the update of the game state until the next move was available (either from keyboard input or from the automatic agent programmed in Tutorial 1). This was done as shown in the following code snippet.

Listing 1: Future attributes incorporation

```
# Store next direction either from kb or from the agent
next_dir =  move_tutorial_1(game)

# Printing ARFF line (same game state, next move)
with open(arff_filename, mode='a', encoding='utf-8') as arff_file:
    print_line_data_arff(game, arff_file, new_score)

# Moving the snake
game.direction = next_dir
if game.direction == 'UP':
    game.snake_pos[1] -= 10
if game.direction == 'DOWN':
    game.snake_pos[1] += 10
if game.direction == 'LEFT':
    game.snake_pos[0] -= 10
if game.direction == 'RIGHT':
    game.snake_pos[0] += 10

# Updating the score
game.score = new_score
```

So, using this, we are sure that the game state is not updated until we print the extracted features in the output file which implies that the dataset has the form $(current\_game\_state\_attr, next\_move)$.
The same methodology was used for the next score as well. But since this is a future attribute that is not necessary for Phase 2, it was not deployed until Phase 4 of the model where we predict the score in the next tick using regression models.

1.3 | Final Words

Using the above described methodology, we were able to obtain two large datasets, one is obtained from playing manually using the keyboard and the other one is obtained from the automatic agent programmed in previous tutorials. Each data set was splitted to a training and a test data set with a ration of 4 : 1 respectively.
The Python code for the `compute_game_state` function is listed below:

Listing 2: Feature extraction process

```
# Python code for computing game state
def compute_game_state(game):
    # Compute game state attributes
    ...
    return up_space, down_space, left_space, right_space,
    can_up, can_down, can_left, can_right, x_distance, y_distance, snake_body_grid
```

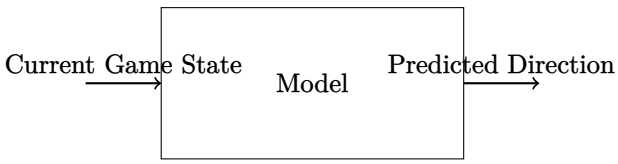

Figure 1.1: Model Architecture

2 | Phase 2

This section covers the procedures followed as well as the conclusions reached while doing the second phase of the practice.
After obtaining the datasets from the previous phase, it was necessary to try to classify them using different classification models. The following subsections describe the methodology followed.

2.1 | Data Preprocessing

First, we had to make sure that the dataset was good enough to be used for model training. The following represent the evolution of the dataset throughout the phase.

- **First Version:** This version of the dataset contained the following attributes.

```
@attribute snakeHeadX numeric
@attribute snakeHeadY numeric
@attribute snakeTailX numeric
@attribute snakeTailY numeric
@attribute foodX numeric
@attribute foodY numeric
@attribute availableUp {0,1}
@attribute availableDown {0,1}
@attribute availableLeft {0,1}
@attribute availableRight {0,1}
@attribute xDistance numeric
@attribute yDistance numeric
@attribute currentScore numeric
@attribute currentDirection {UP, DOWN, LEFT, RIGHT}
```

  This was the very first dataset we obtained for this phase. We noticed the following.

  - The dataset was balanced where all the classes inside the *currentDirection* had roughly the same number of training instances.
  - The information gain of the attributes $availableUp$, $availableDown$, $availableLeft$, $availableRight$ was too high (above 1.9) and it led to the model trained to achieve an accuracy of over 99%. This was somehow suspicious.
  - The information gain of current score attribute was too low (below 0.02) which does not make sense since the model is trying to move towards the food to maximize the score so the score attribute should have higher information gain.

- **Second Version:** This version of the dataset contained the following attributes.

```
@attribute snakeHeadX numeric
@attribute snakeHeadY numeric
@attribute snakeTailX numeric
@attribute snakeTailY numeric
@attribute foodX numeric
@attribute foodY numeric
@attribute availableUp {0,1}
@attribute availableDown {0,1}
@attribute availableLeft {0,1}
@attribute availableRight {0,1}
@attribute distance_to_food real
@attribute currentScore numeric
@attribute currentDirection {UP, DOWN, LEFT, RIGHT}
```

  This was an update on the previous dataset. We noticed the following.

  - The dataset was balanced where all the classes inside the *currentDirection* had roughly the same number of training instances.

- The information gain of the attributes $availableUp, availableDown, availableLeft, availableRight$ was too high (above 1.9) and it led to the model trained to achieve an accuracy of over 99% again. This was somehow suspicious.
- The information gain of current score attribute was too low (below 0.09) which was still not enough.
- The information gain of the Euclidean distance attribute was close to that of the distance attributes from the previous phase.

- **Final Version:** This version of the dataset contained the following attributes.

```
@attribute snake_head_x numeric
@attribute snake_head_y numeric
@attribute food_x numeric
@attribute food_y numeric
@attribute up_space numeric
@attribute down_space numeric
@attribute left_space numeric
@attribute right_space numeric
@attribute can_up {True,False}
@attribute can_down {True,False}
@attribute can_left {True,False}
@attribute can_right {True,False}
@attribute x_distance numeric
@attribute y_distance numeric
@attribute current_score numeric
@attribute row1_col1 {Y,N}
@attribute row1_col2 {Y,N}
..
@attribute row2_col1 {Y,N}
..
@attribute row6_col6 {Y,N}
@attribute current_direction {UP,DOWN,LEFT,RIGHT}
```

This was an update on the previous dataset. We noticed that the available attributes for the four cardinal directions were not computed correctly and that was why they were causing the model to overfit the data. For the sake of solving this problem, we splitted those attributes to two different sets; $(up\_space, ..)$ which represent the available space in each cardinal direction and $(can\_up, ...)$ which represent whether we can move the snake in each cardinal direction or not. In addition to this, we incorporated the snake body parts in the $@attributerowN\_colNY, N$ as explained in Phase 1.

- The dataset was balanced where all the classes inside the $currentDirection$ had roughly the same number of training instances.
- The information gain of the attributes $can\_up, can\_down, can\_left, can\_right$ was balanced (around 0.3).
- The information gain of current score attribute became the highest (around 1.8) which makes the model more efficient.
- The information gain of the Euclidean distance attribute was high as well.
- The information gain of the snake body parts attributes is too low. This is explainable because the dataset does not include enough instances that make this attribute relevant. Almost all the dataset instances are trying to maximize the score as much as they can. There are no enough instances that make the model capture details that make those attributes relevant.

Below is the top 11 attributes ranked by **InfoGainAttrEval** using the **Ranker** search method.

```
Ranked attributes:
1.884807    15 current_score
1.24609     14 y_distance
0.858049    13 x_distance
0.60601      3 food_x
```

```
0.489845    4 food_y
0.329866    7 left_space
0.329866    1 snake_head_x
0.329866    8 right_space
0.275617    2 snake_head_y
0.275617    5 up_space
0.275617    6 down_space
```

To be fair enough across the evolution of the data, we fixed the evaluation criteria of the quality of a dataset. The criteria includes accuracy of the J48 model, the size of the model tree, Information gain of the attributes and the size of the datasets. This way we make sure that we do not modify a dataset resulting in a worse representation of the game instances. Any modification was ensured that it was in the positive direction of the overall gain in the process.

2.2 | Data Classification

Now, we try different classification algorithms to try to classify the data we have in order to generate a model that is intelligent enough to make reliable decisions for the snake game. The following sections include different models that were experimented.

2.2.1 | J48 Rule Based Model

The J48 model was the first model we tested an ultimately preformed very well from the start of our testing phases. It was a model that was easy to understand and interpret due to the in class familiarity, but, J48 models also builds decision trees quickly, especially for datasets with a moderate number of features and instances like the training snake data used. This efficiency and ease of understanding is crucial for near-real-time applications to ensure smooth game play.
The confusion matrix and ROC curve showcased below:

UP	DOWN	LEFT	RIGHT
3964	14	56	39
16	4169	35	61
31	24	3911	1
30	41	1	3978

Table 2.1: Confusion Matrix J48

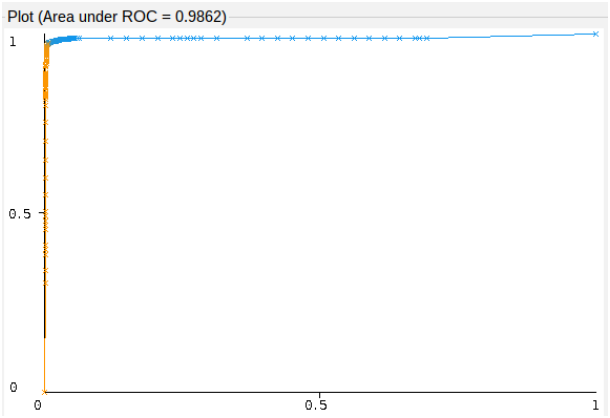

Figure 2.1: ROC Curve J48

From the figures above it is clear to see that the J48 model performed exceptionally well. With over 16000 correctly classified instances and 97.82 percent precision across all instances, it was a very strong candidate upon initial testing however a more precise method may have yet to be tested.

2.2.2 | PART Rule Based Model

The next model chosen to test was the PART model due to its robustness and ability to handle incremental learning with ease. Similar to the J48 model, PART had very promising results in many key attributes when evaluating the model while also being easy to understand and dynamic in its decision rules. After initial testing however, the PART method began as an initial worse performing method however after 4 iterations due to the benefits of the incremental learning process it became significantly more accurate. Metrics for PART model shown below:

UP	DOWN	LEFT	RIGHT
3982	15	40	36
15	4169	39	58
42	28	3903	2
29	28	1	3982

Table 2.2: Confusion Matrix PART

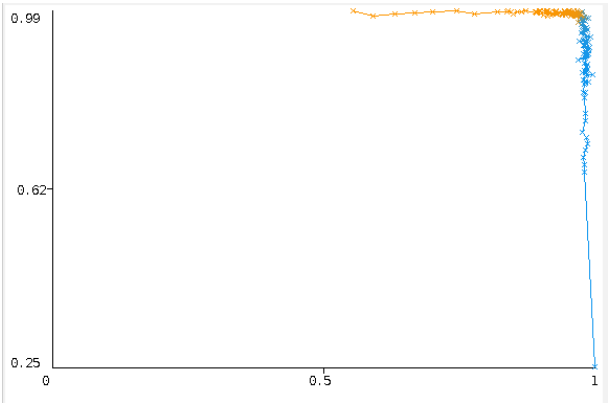

Figure 2.2: Percision Recall Curve PART

After the 4th iteration of testing, the part method was slightly weaker in terms of key metrics compared to the J48 tree with a slightly lower precision, recall and f-1 score across all classifiers (¡0.5 percent on average). However, this was after multiple iterations thus the model could be more intelligent and ultimately produce better results. Thus we decided our model implementation would not be built off of the PART model.

2.2.3 | JRip Rule Based Model

The final model we tested was one not commonly seen in class. We decided to test the JRip method as we found that it is a good model in scenarios where a balance between predictive accuracy and ease of use are needed, and where the data may contain noise or require efficient handling due to its size. This seemed like an appropriate fit to model the snake game off of thus we evaluated the model on the metrics shown below:

UP	DOWN	LEFT	RIGHT
3948	36	50	39
26	4164	38	53
43	57	3875	0
34	90	0	3926

Table 2.3: Confusion Matrix JRip

Overall, the model performed significantly worse than both others. Boasting a lower accuracy, precision and f-1 score by approximately 0.6 percent on average.

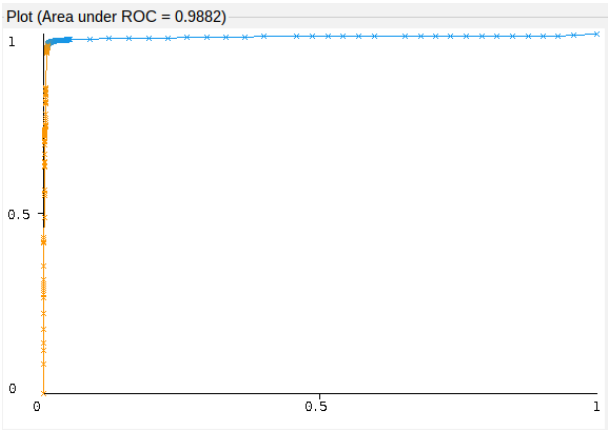

Figure 2.3: ROC Curve JRIP

2.3 | Comparisons Between Models

For more of an in depth comparison between all models, an accuracy comparison plot as well as model comparison table incorporating algorithm complexity are below:

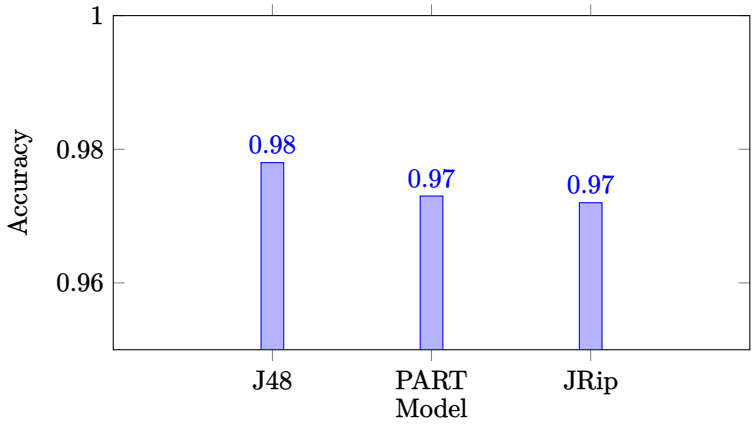

Figure 2.4: Accuracy Comparison of Classifiers

Based on the comparison above, based solely on the precision comparison, there is a clear winner to base the model off of.

Table 2.4: Comparison of Key Statistics for Classifiers

Model	Accuracy	Precision	F1 Score	# of Rules/Tree Size	Compute Time(s)
J48	0.978	0.978	0.978	493	0.2
PART	0.973	0.973	0.973	198	0.99
JRip	0.972	0.972	0.972	79	7.44

When taking a look at in depth model comparisons as well it is easy to see that J48's metrics were similar to all others that we tested however had key benefits across the board. In terms of pure evaluation statistics the J48 model performed the best but additionally had the lowest complexity of the model. The time to run the model was one fifth of the time it took for the second best PART model to complete and the JRip model did not even come close. Therefore, after taking all these facets into consideration, the J48 model was the most appropriate to base our model off to ensure the highest score in the snake game.

3 | Phase 3

This section covers the experimentation and implementation of the intelligent model. The features were tested and compared thoroughly within phase 2 thus this phase focused heavily on integrating and improving upon the game utilizing the automatic agent.

3.1 | Explanation of Model Selection

The J48 decision tree model was selected for Phase 3 due to its superior performance. It demonstrates high accuracy and efficiency with low complexity, making it ideal for real-time applications like the snake game. Its ease of interpretation and robustness across various tests also contributed to its selection, ensuring a smooth integration into the game environment with reliable and intelligent gameplay.

3.2 | How the model works

The J48 model works as the controlling module in our snake game by reading in data at each tick and utilizing the features extracted as training data where the model classifies what move to be taken. As stated previously, due to the low compute times of the J48 model it has proven to be a very efficient and robust solution to the problem and can dynamically classify instances to control the snake and achieve higher scores than we have whilst playing manually with the keyboard. The main logic is stored in the J48.model file that is called through the WEKA framework integrated within our snake game file in python. As data is fed in, the classified instance will be the input for the games next move and ultimately guide the snake throughout the subsections of the game grid. Ultimately, the model allows for an intelligent playing method that only improved as more instances were collected and it was tested on more data to recognise new patterns within the data and significantly improve on the game score as a result.

4 | Phase 4

In this phase, we try different regression models to predict the score in the next tick based on the current game state in the present tick. To achieve this, we had to prepare training instances for the model to be trained on. For simplicity, we removed irrelevant attributes from the previous phases. Each instance in the dataset had the following attributes.

```
@attribute snake_head_x numeric
@attribute snake_head_y numeric
@attribute food_x numeric
@attribute food_y numeric
@attribute x_distance numeric
@attribute y_distance numeric
@attribute current_score numeric
@attribute current_direction {UP,DOWN,LEFT,RIGHT}
@attribute next_score numeric
```

These attributes are what were found to be relevant to predict the score in the next tick. Since the score depends on the snake movements, and assuming that the movements are roughly in the direction of score optimization, then the attributes for doing optimal movements will also contribute in maximizing the new score. As explained in Phase 2, the mechanism for extracting the next score attribute was shown above. Now, we compare two models that were used for this prediction task

4.1 | Linear Regression

Linear regression by definition assumes that the output is a linear combination of the input features. More precisely, using this model to predict the next score will implicitly mean that the next score is represented as a linear combination of the input features to the model. After training the linear regression model on a dataset with more than 10000 instances, the model representation was as follows.

$$\begin{aligned} next\_score = &- 0.0025 * snake\_head\_y + 0.0035 * y\_distance \\ &+ 1.0001 * current_score + -6.8248 * current\_direction = DOWN, UP, LEFT \\ &+ 0.6971 * current\_direction = UP, LEFT + 6.2275 * current\_direction = LEFT + 6.003 \end{aligned}$$

Observing the weights of each attribute, we can see that the most influential attributes are the snake direction and current game score. The following data were obtained after the model training. Looking at

Correlation coefficient	0.9999
Mean absolute error	6.3315
Root mean squared error	17.7461
Relative absolute error	0.4355%
Root relative squared error	1.028%

Table 4.1: LR Model Summary

the summary above, we can see that the model has a high correlation coefficient (almost 1) indicating that it was able to linearly correlate the data points to each other. The mean absolute error is also low (6.33) which is relatively small compared to the range of scores in the game.
One feature that was incorporated in the game is counting how many correct score predictions have been made. If the model predicts the score within a range of error (defined by the mean absolute error), the prediction is classified as being valid. This is shown by the following code snippet.

```
def update_error(game, new_score, a):
ERROR_RANGE = 7
game.ticks += 1
# print("Predicted "f"{a} expected "f"{new_score}\n")
if (abs(a - new_score) <= ERROR_RANGE):
    game.correct_pred += 1

# And include those attributes in the game __init()__
self.ticks = 0
self.correct_pred = 0
```

4.2 | Multilayer Perceptron

The Multilayer Perceptron (MLP) is a type of artificial neural network (ANN) that consists of multiple layers of nodes, or neurons, arranged in a feedforward manner. Each neuron in one layer connects to every neuron in the subsequent layer. Each connection between neurons in adjacent layers is associated with a weight parameter. These weights determine the strength of the connection and are learned during the training process. Additionally, each neuron has an associated bias term, which allows the network to learn even when all inputs are zero. fter computing the weighted sum of inputs and biases for each neuron, an activation function is applied to introduce non-linearity into the network. Common activation functions include the sigmoid, tanh, ReLU (Rectified Linear Unit), and softmax functions. The choice of activation function depends on the specific requirements of the problem and the characteristics of the data. For our training purposes, the model used Sigmoid as an activation function. After training the model on the same dataset, a NN of 8 layers was obtained as shown below.

```
Linear Node 0
    Inputs      Weights
    Threshold     0.24358381427502884
    Node 1      -0.8060508988025199
    Node 2      -0.8549518381259075
    Node 3      -0.6021522021879491
    Node 4      -0.7347390619621538
    Node 5      1.6230408548607844
    Node 6      -0.16870101512639385
Sigmoid Node 1
    Inputs      Weights
    Threshold     -0.38543104116627225
    Attrib snake_head_x      -0.009292840449721418
    Attrib snake_head_y      0.010536120969823535
    Attrib food_x      -0.006669386631927266
```

```
        Attrib food_y       -0.014493353468317672
        Attrib x_distance       -0.01660209594253879
        Attrib y_distance       -0.0026503567764756615
        Attrib current_score       -1.886325488963521
        Attrib current_direction=UP       0.22341134777897148
        Attrib current_direction=DOWN       0.23822691941935278
        Attrib current_direction=LEFT       0.21504802171816872
        Attrib current_direction=RIGHT       0.18254793544745854
...
Sigmoid Node 6
        Inputs       Weights
        Threshold       -0.8643635968924944
        Attrib snake_head_x       -0.029047114116068067
        Attrib snake_head_y       0.03187396720349668
        Attrib food_x       0.018029366257536 23
        Attrib food_y       0.008332158832377961
        Attrib x_distance       -0.06364202978826532
        Attrib y_distance       0.016190926338594543
        Attrib current_score       -0.20933385155683432
        Attrib current_direction=UP       0.4575059207488654
        Attrib current_direction=DOWN       0.3590682338186071
        Attrib current_direction=LEFT       0.4748568827985994
        Attrib current_direction=RIGHT       0.4015040153724703
Class
        Input
        Node 0
```

As we can see above, the attributes that have the highest weights in each later are the snake direction and the current game score which coincides with the results obtained from the previous model. The following data were obtained after the model training. Similarly, the model has a high correlation coefficient

Correlation coefficient	0.9999
Mean absolute error	8.8988
Root mean squared error	17.9746
Relative absolute error	0.612%
Root relative squared error	1.0412%

Table 4.2: MLP Model Summary

indicating that it was able to correlate the data linearly. The mean absolute error is low relative to the game score but it is higher than that of the J48 model.

5 | Questions

5.1 | Difference Between Learning Models with Human-Controlled vs. Automatic Agents

Learning with instances from a human-controlled agent would lead to a higher level of variability and unpredictability since humans may not use strategies as consistently or make errors different from what an algorithm might produce. Therefore, this can provide a diverse training set that helps the model learn various scenarios.
Conversely, an automatic agent usually follows predefined rules or learned behaviors, leading to more consistent and predictable instance generation. This consistency can make the training set less diverse but might lead to a more stable learning process. However, it might not deal with unexpected scenarios as well as a human-controlled agent.

5.2 | Transforming the Regression Task into Classification and Practical Applications of Predicting the Score

To transform the regression task (predicting the score) into classification, you would need to define categories or ranges of scores and then classify each instance into these categories instead of predicting a numerical score. For example, score ranges could be categorized as low, medium, and high. Predicting the score could have practical applications like evaluating the effectiveness of different strategies, understanding game dynamics at different stages, or for creating adaptive difficulty levels where the game adjusts its complexity based on predicted scores.

5.3 | Advantages of Predicting the Score Over Classifying the Action

Predicting the score provides an overall assessment of the game strategy and effectiveness. It allows you to measure the long-term impact of actions taken by the snake. Score prediction can help in understanding the outcome of certain sequences of actions, leading to better strategic planning. In contrast, classifying the action is more about immediate decision-making. While crucial, it doesn't directly inform you about the effectiveness of those decisions in terms of game objectives (like maximizing the score).

5.4 | Impact of Incorporating an Attribute Indicating Score Drop on Ranking Improvement

Including an attribute that indicates whether the score has dropped could potentially improve the model. It might help in identifying scenarios or actions that lead to a decrease in score, thus providing valuable insights for avoiding such outcomes.
This attribute could be particularly useful in regression models as it might correlate strongly with certain actions or states that negatively impact the score.
However, the effectiveness of this attribute would depend on how well it correlates with other features and the overall predictability of score drops in the game context.

6 | Conclusion

6.1 | Technical Insights

6.1.1 | Feature Extraction Complexity

The complex process of extracting game attributes like snake position and food location was central to our model's performance, emphasizing the significance of careful feature selection in machine learning.

6.1.2 | Integration Challenges with Weka and Python

Integrating Weka with Python presented complexities, highlighting the trade-offs between specialized and versatile tools. This underscored the benefits of using flexible libraries like sklearn over more specialized ones.

6.1.3 | Dependency Management

Our experience with inaccessible dependencies without a virtual machine pointed out the need for considering tool accessibility in collaborative projects.
Model Generalization and Improvement: The models showed good performance but were limited by the specificity of their training data. This emphasized the importance of diverse datasets and the potential of ongoing model retraining for enhanced adaptability and accuracy.

6.2 | Broader Implications and Personal Reflections

6.2.1 | Practical Applications Beyond Gaming

The project broadened our understanding of machine learning's applicability in various domains, underscoring its versatility beyond just gaming scenarios.

6.2.2 | Inter-Tool Integration Challenges

Working with Weka, Python, and the gaming environment highlighted the practical challenges in machine learning, particularly in tool integration.

6.2.3 | Critique and Future Directions

A more seamless integration of machine learning algorithms within Python would be beneficial. Future improvements could include exploring Python-based machine learning libraries such as sk-learn for more streamlined project development.

6.2.4 | Value of Continual Learning

This experience reinforced the importance of continual learning and adaptation in AI, both technically and collaboratively.
This project, while challenging, provided valuable insights into machine learning's practical applications and the importance of thoughtful tool selection and feature extraction in developing effective models.