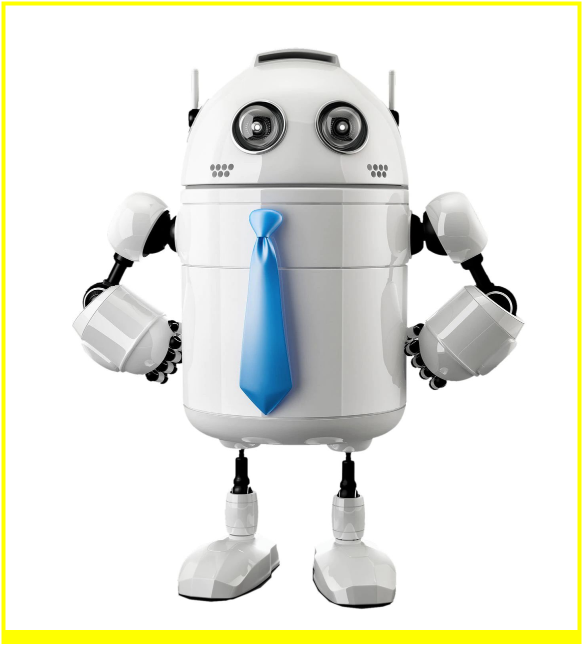
<u>**Group10: Four Musketeers**</u>

Reinforcement Learning (RL) In Games (Tic Tac Toe) & Stock Market



Bob LeBow

Qingyuan Jiang

Charles Huggins

Selva Arunachalam

Table of Contents

Reinforcement Learning Components

Reinforcement Learning models consist of the following six components:

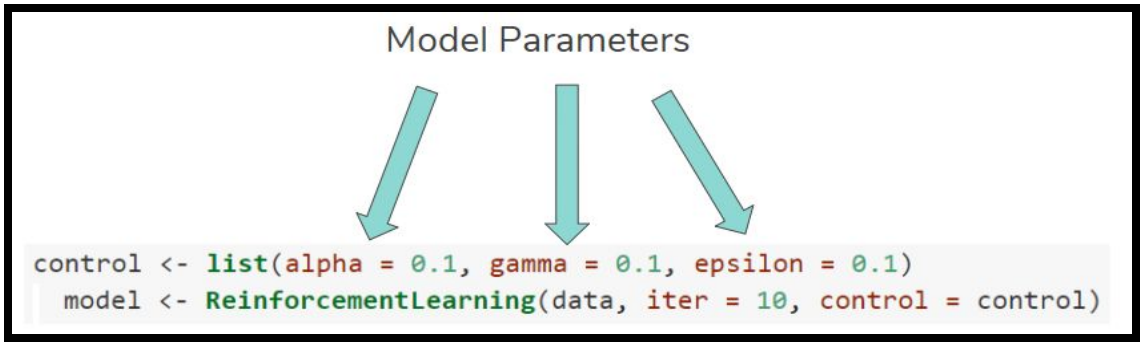
- Actions
- Agents
- States
- Rewards (or Penalties)
- Environment
- Policy

Given a particular state, agents take actions, then receive a reward or penalty based on the result of that action. Rewards (or penalties) are the feedback by which success or failure is measured. For example, while the reinforcement learning model used to train a driverless car is undergoing training: if the agent decides to take the action of "accelerate" when it is presented with the state of "red light," the agent will receive a penalty, which would be the subtraction of reward points from its net total. Given the feedback loop that is employed through reinforcement learning, the model will eventually be able to identify the optimal policy for each state.

Reinforcement learning models are created through an optimization function, where the total net reward is what the agent is trying to optimize: Given a state, the action with the highest net reward will be chosen by the agent. Given enough iterations, the agent will ideally know the optimal action for any possible set of circumstances (*state*) it finds itself in.

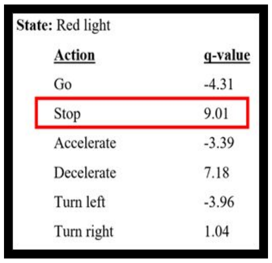
Parameters

The three parameters involved in reinforcement learning are epsilon, alpha (otherwise known as the "learning rate"), and gamma (otherwise known as the "discount factor"). These parameters control how the agent makes decisions, as well as how past information is stored.



```
control <- list(alpha = 0.1, gamma = 0.1, epsilon = 0.1)
  model <- ReinforcementLearning(data, iter = 10, control = control)
```

The chosen value for Epsilon represents the probability that your model will choose actions at random. If the chosen value for epsilon is relatively high, your model will usually "roll the dice," so to speak, by choosing an action at random. The benefit of choosing actions at random is that your model will be more likely to find never-before-seen state/action combinations that are more beneficial (i.e., higher reward) than what was previously regarded as the "optimal" action for the given state (i.e., the policy). The value 1 – epsilon will thus give you the probability that your model will select an action based on what it has learned from previous iterations. In other words, 1 - epsilon tells you the probability that your model will refer to its q-values in order to make a decision. Continuing the example of using a reinforcement learning model for training a driverless car, the state "red light" might have the following q-values:



State: Red light

Action	q-value
Go	-4.31
Stop	9.01
Accelerate	-3.39
Decelerate	7.18
Turn left	-3.96
Turn right	1.04

Based on the above q-values for the state "red light," our model will choose the action Stop, as that action is associated with the highest q-value. If a relatively high value were chosen for epsilon, your model would be more likely to choose one of the other states, regardless of how low (and unappealing) their q-values are. Because the q-values are affected by the total reward received in the past, alpha will affect the magnitude of the q-values. For this reason, the alpha will affect how quickly the agent "learns" from previous iterations (thus the reason alpha is also known as the "learning rate").

Gamma (also known as the "discount factor") tells your model how important future rewards are in comparison to immediate rewards. The higher the discount factor, the more important future rewards are. A lower discount factor makes immediate rewards more important than future rewards. How your reinforcement learning model makes decisions on which actions to take depends on the present value of the expected reward for each potential course of action. The present value is calculated as follows:

$$\text{Present\_Value} = \gamma\text{\^{}(time\_steps)} * \text{Expected\_Reward}$$

WHERE:

- "time_steps" represents how far into the future a reward is expected to be received (i.e., the number of *time steps* into the future the reward is expected).
- “Gamma” is the discount factor
- “Expected_Rewar” is the rewards your agent expects to receive

For example, imagine a situation where a driverless car has two options: Option 1) It could choose the action change lanes, which would result in an immediate reward of 5 reward points; Option 2) it could choose the action accelerate, which would result in an expected reward of 15 points three time steps into the future. The present value of Option 1 is 5 reward points, as the 5 reward points would be received immediately (i.e., 0 time steps into the future):

$$\begin{aligned}\text{Present\_Value} &= 0.4\text{^}0 * 5 \\ &= 5\end{aligned}$$

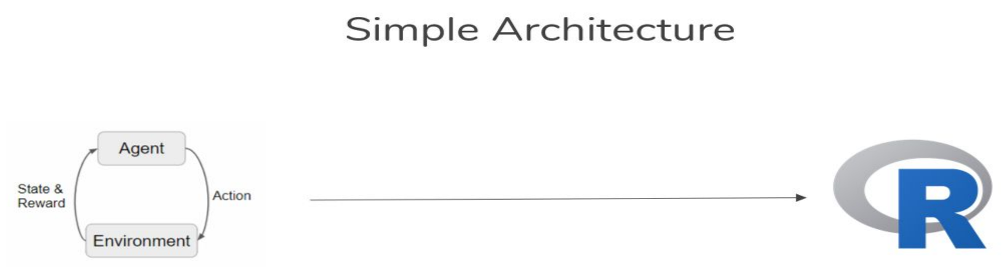
The present value of Option 2 depends on what gamma value is chosen. If the gamma is set to 0.4, then the present value of Option 2 would be a mere 0.96 reward points:

$$\begin{aligned}\text{Present\_Value} &= 0.4\text{^}3 * 15 \\ &= 0.96\end{aligned}$$

In this case, the agent would choose Option 1, as it has the higher present value. On the other hand, if the gamma were set to 1.0, then the present value of Option 2 would be 15 reward points (1.0^3 * 15), in which case Option 2 would be the chosen action. So what value should be chosen for gamma? If your agent is being trained for a task that would require it to see several steps into the future with high accuracy, then a high discount factor would be necessary, as it would force the reinforcement learning algorithm to choose options that would benefit it several steps into the future.

The question then becomes, Which values should we choose for each of these parameters? Considering how experimentation is the only effective way to determine optimal parameter values, answering this question is a difficult task due to how time-consuming training a reinforcement learning model is. Training a reinforcement learning model with a million iterations can take an entire week (or longer) on a standard laptop. This has been a significant limitation for us.

Architecture

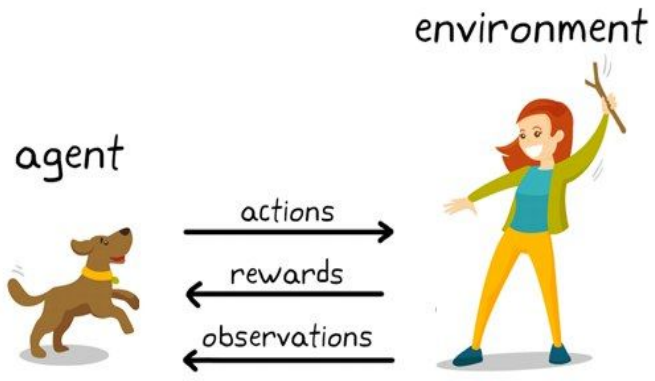


Our Implementation of RL architecture is straight forward, we use RL in R.

There are two main principles in the Reinforcement Learning Architecture

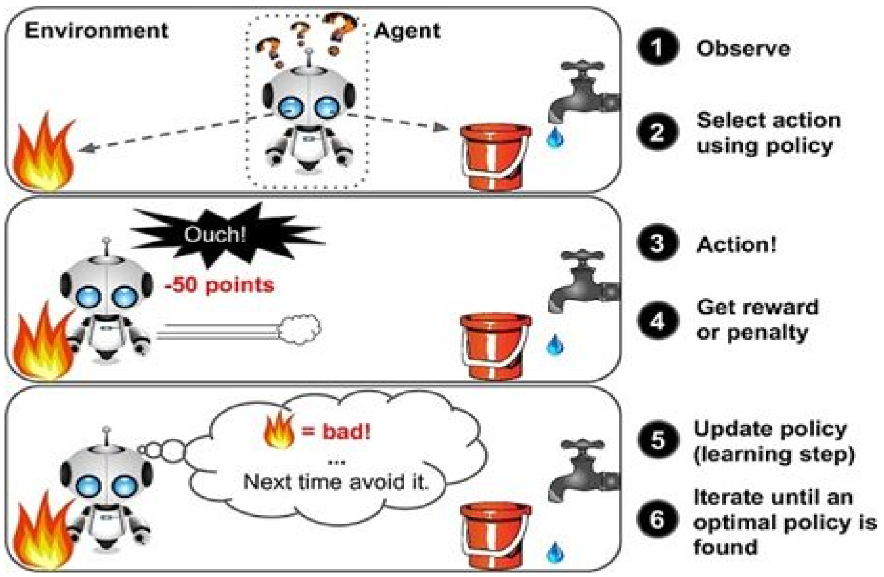
1. Principle One:

    a. When an Agent takes an action after observing the environment and in-return we reward the agent with a reward if the action taken has a positive influence towards our goal. Example depicted in the picture below, if our agent, lets say, our dog is our agent and we play fetch game and we throw a ball and our agent(dog) observes environment / our commands and makes a decision to run to go fetch and fetches it and brings it to us, then when it completes we reward with a biscuit some sort.



2. Principle Two:

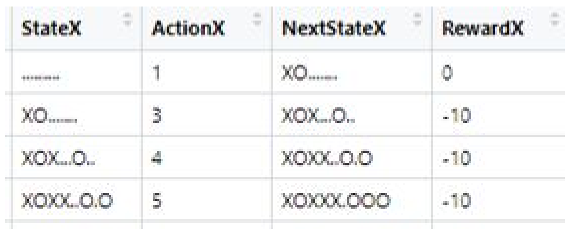
   a. When an agent takes an action after observing the environment and if the action leads to unfavourable consequences, then we penalize or give penalty to the agent (opposite of rewarding, ie if we reward with positive points, here we give negative points) thus our agent will learn not to repeat this step or will try to avoid making such decisions. This 2nd principle is depicted in the picture below


Environment
Agent
1 Observe
2 Select action using policy
Ouch!
-50 points
3 Action!
4 Get reward or penalty
= bad!
...
Next time avoid it.
5 Update policy (learning step)
6 Iterate until an optimal policy is found

Example

Tic-Tac-Toe

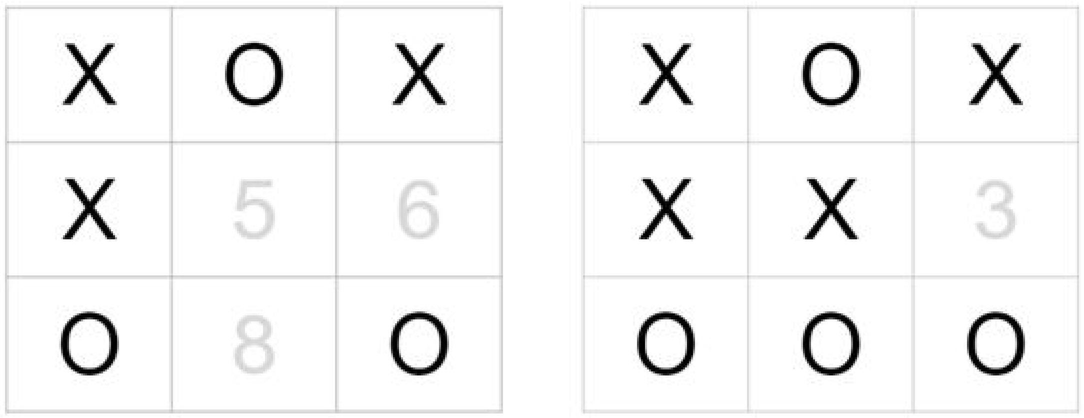
Here we have an example of RL in tic-tac-toe. As shown in picture 1. There are four columns. First and the third are the "States".



StateX	ActionX	NextStateX	RewardX
....	1	XO.......	0
XO.......	3	XOX...O..	-10
XOX...O..	4	XOXX..O.O	-10
XOXX..O.O	5	XOXXX.OOO	-10

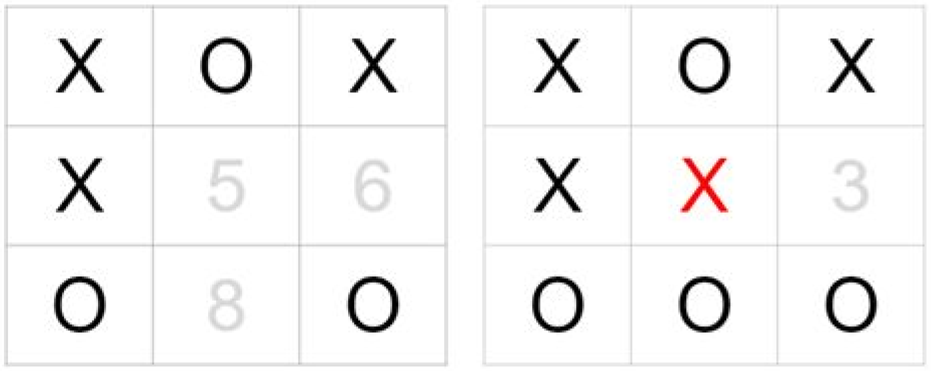
Picture 1

First column is the state before action. To distinguish it from another state, we call it "Initial State". The third column in the state after action, we call it the "End State". In Picture 2, the left one is the state before action, the right one is the state after.



Picture 2

The second column in Picture 1 is “Action” column. Action is the move after Initial State, it is the cause of the End State. As shown in picture 3, the “X” in red is the action by agent 1. The End State is a result caused by the Action.



Picture 3

Column four is the reward, it could be a positive reward, which means we want to reward for the Action. It also could be negative, which means we don’t want this kind of Action to shows up again.

Autonomous Vehicle

Games are not the only place that we can apply Reinforcement Learning.

For example, Reinforcement Learning could apply to auto driving cars. The State in auto driving cars are pictures captured by sensors. Actions are the different movements for cars like accelerate, decelerate and make turns etc.

Whenever the sensor captures an image. Reinforcement Learning will analyze the possible Actions, then give rewards and penalties based on the End State condition. If the End State image shows the car is off the road, it will give the action penalty. If the End State is driving safely, then it will be rewarded.

That's how an auto driving car knows to avoid dangerous actions.

State	Action	NextState	Reward
	Go!	MATURE 17+ ™ M Blood Strong Language Violence ESRB CONTENT RATING www.esrb.org	-999
	STOP!		+1
	TURN LEFT!		+1

Picture. Auto Driving Car Example

Business Model

Coming from a business background I was tasked with explaining our business model for the purposes of this project. Our team initially wanted to create a simulated autonomous vehicle, powered by reinforced learning. The business application of this idea was very simple. Once our lack of time and processing power became evident, we decided to change the scope of our project to simply: understanding how to train a reinforcement model technically, while also figuring out what resources were needed to do this as efficiently as possible. All of this while still keeping in mind the end goal of using reinforcement learning to power an autonomous vehicle.

Analysts at Allied Market Research estimated the autonomous vehicle market at $54 billion.[1] Our team figured this was a very large pie that we, or a company using our model, would only need a small sliver of to be considered successful. We found two companies and a paper discussing the use of reinforcement learning principles in one way or another for autonomous vehicle applications and stuck with that (Since our presentation we have geared our thinking to recommendation applications based on the professor's advice).[2] We thought the understanding we had gained from our tic-tac-toe experiment would give us the ability to make other models for other applications.

This is when we decided to also apply our skills to the card game blackjack as well. We figured we would learn even more about reinforcement learning by making a model for another application and we would learn a lot from comparing and contrasting two use cases. This would improve our ability of being able to make new models for different applications and not just be pidgeon holded with a plethera of useless tic-tac-toe-specific knowledge.

Project 1: Tic Tac Toe

Generating Data

For our attributes of data, we need a start State, Action for the State, the End State, and the reward. Because of these specific attributes of data we need, it is impossible to get data from websites. So we need to find a way to generate data by ourselves.

We developed three ways to generate the dataset.

1. We input all the rules for Tic-Tac-Toe in R, then let the computer generate all the possible combinations. But we give it up. There are two reasons why we didn’t choose this plan: 1. It have 9*8*7*6*5*4*2 ~ 360,000 different combinations, take too much time to generate all possible combinations in R. 2. In reality, we will not have a “perfect” dataset which contains all the possible solutions, that's the reason why we need AI to learn by itself.

2. Doing it randomly. Let the computer decide where to place the chess randomly.

3. use RL to analyze and generate data at the same time.

[1] Jadhav, A. (2018, January 14). Autonomous Vehicle Market Size, Share and Analysis: Forecast 2026. Retrieved from https://www.alliedmarketresearch.com/autonomous-vehicle-market

[2] **F. Pusse and M. Klusch, "Hybrid Online POMDP Planning and Deep Reinforcement Learning for Safer Self-Driving Cars," 2019 IEEE Intelligent Vehicles Symposium (IV), Paris, France, 2019, pp. 1013-1020.**

Data Storage

In this project, we have 4 models and all models generate their own dataset while playing against another player and it’s stored locally within the R file and local system. Our RL player uses these self generated dataset based on his all previous moves that lead him to win or lose.

Algorithm

- **Q learning (Tabular Q)**

  Q stands for Quality and Q function assigns quality score to a state, action pair. In formula, for any given Q(S,A), we will have a score i.e. quality values to actions that can be chosen in a given state.

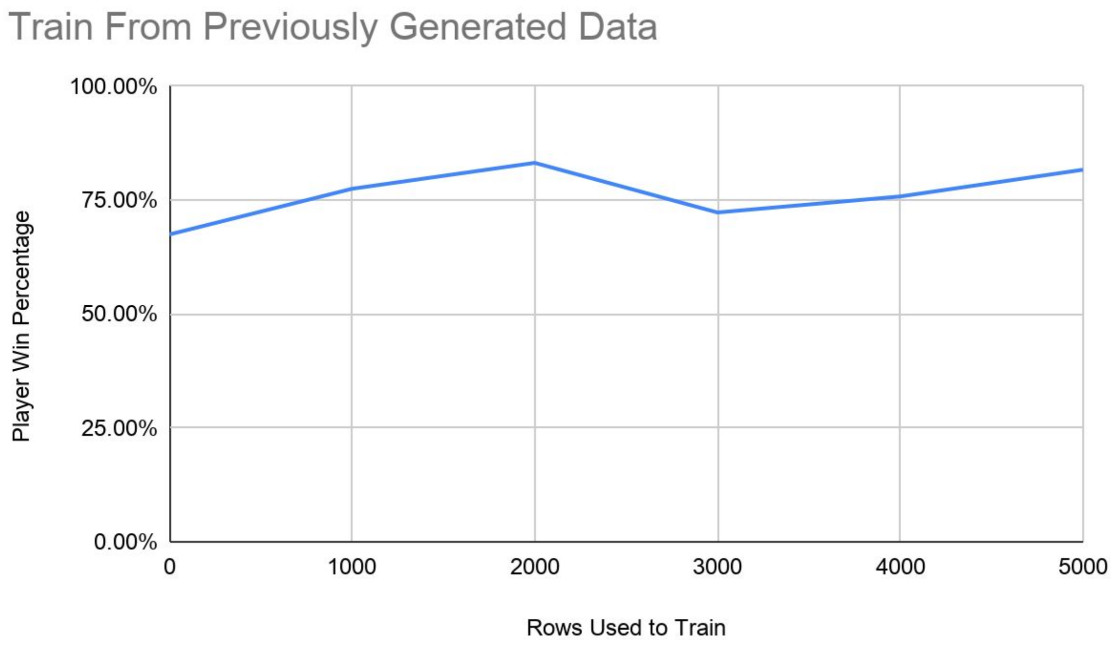
  The key is: The value of a particular given action in a particular given state is related closely to the value of the all possible following state in the Q table. Basically by looking up the Q value in the table for all possible moves and choosing the one that has the highest value, by doing this, we choose the best possible move in this given situation.
  $Q(S,A)= Q(S,A)+\alpha*(\gamma*\text{maxa}Q(S',a)- Q(S,A))$

Models:

Model 1: Random vs. Random (Bob’s)

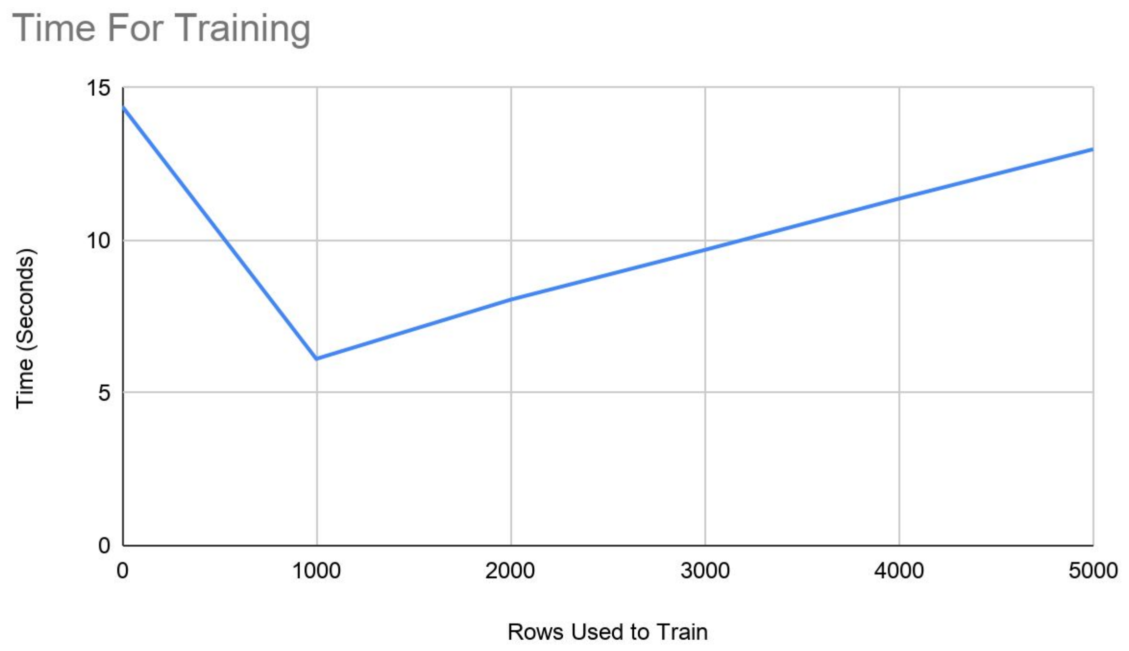
Each member of our team took one iteration of our model in order to split up the time and processing power required to produce useful results. The model I took up was our least power intensive. This was due to fact my model would only be trained on data I had already generated, more on this later. My model was given the moniker “Random vs. Random”, because I would create a table with the specific attributes required for reinforced learning (StartState, NextState, Reward, Etc) and fill it with two random players making moves, while recording the winner. For our presentation and for consistency's sake, we each used five thousand rows of games played as our maximum training set. After our models were trained we would play a game and record how many our players won.

As stated previously my model would learn after the games had been played and the winner had been recorded. This would mean my model could not learn from its mistakes, it had a fixed ability to play tic-tac-toe. As you can see in the graph below, this did not work to great success. Both the random players made nonsensical moves which would sometimes lead to a win or a loss essentially at random (with a preference to our player because it places its “X” first). It became clear we were not reinforcing moves that would add up to a good strategy but actually a random strategy.



According to the trend line it would take nearly a quarter million rows of data to obtain a player that could win 100% of the time. To test this i trained my model on a random vs. random set i created with a million rows just to see what would happen. My win rate actually decreased to 68%; a 50% chance of winning plus 18%, being about the average advantage given to the player that gets to place their marker first. This confirms that over a large enough data set the “Random vs. Random” would eventually even out to no discernable strategy and be useless.

The only advantage being the short time and processing power required to test it, as evidenced



below

Model 2: RL vs Random (Selva's)

In Selva's model, our first player is "Agent-X", who is learning by doing using Q Algorithm with Reinforcement learning technique for every step (after the initial first step, which was a random move, since there was no previous dataset for our Agent-X, thus first move was random even for our Agent-X Reinforcement Learning Player). And "Agent-Y" is the regular player with random moves using guess work, the 2nd player does not use reinforcement learning, he just basically guesses every move with no intention to learn from all his previous moves. Thus, this model was named as RL vs Random.

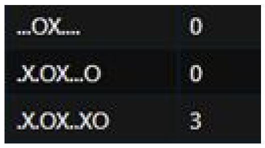
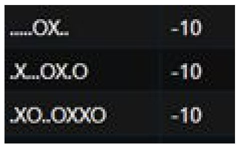
· Player-1, Agent-X, creates his own dataset from step-2 onwards (using all his previous moves he has made thus far). For every move, he uses all his previous steps to learn from it and uses the Q learning algorithm and makes the best possible move to win the game. Example, if he is at the nth move, he will learn from 1st step till the (n-1)th step and makes a decision with high Q value and decides to move to that spot in the table, with high probability of winning (at that given point of time)

- Player-2, Agent-Y, creates no dataset to learn from his previous steps
- But we capture both players' moves in our t1 table.
- RL parameters (values used ---> alpha = 0.2, gamma = 0.4, epsilon = 0.1)

<u>Reward and penalty</u>

Both the players will be rewarded if their moves attribute to their game winning, ie if Player1,Agent-X wins he gets 3 points as reward and on the other hand, if he loses a game, he gets a penalty of -10 points (not only to the final step that cost him the game but all the steps that lead him to lose the game).

Example:

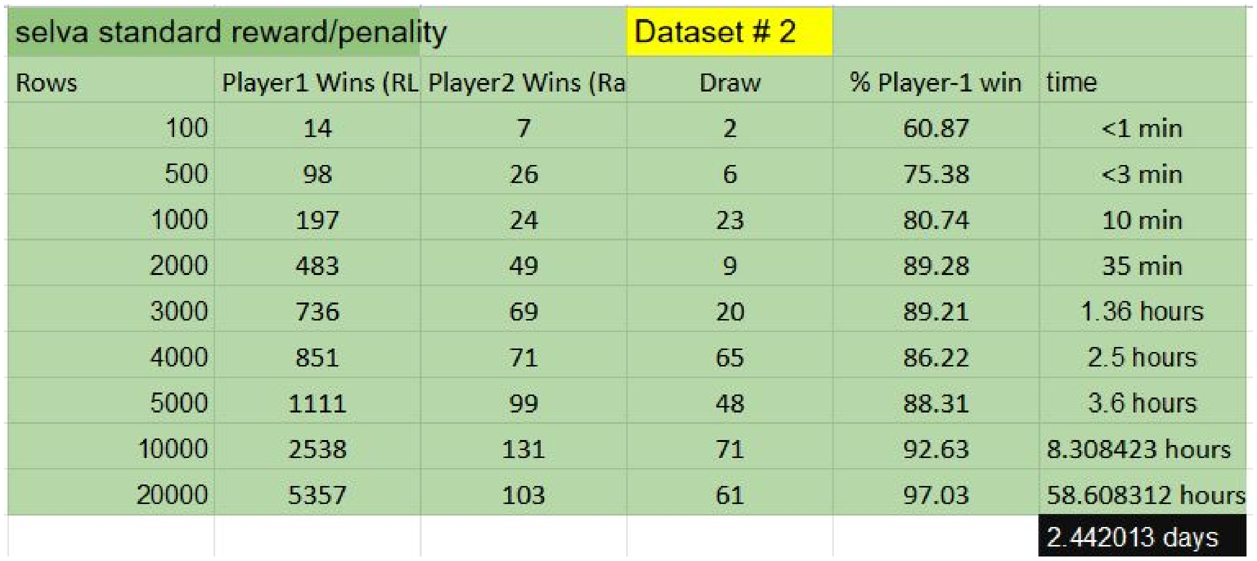
Agent-X gets rewarded for winning a game	Agent-X gets penalty for losing a game
 ...OX.... 0 .X.OX..O 0 .X.OX..XO 3 ● Reward = +3 (for winning, for only the winning final step of a winning game)	 .....OX.. -10 .X...OX.O -10 .XO..OXXO -10 ● Penalty = -10 (for losing, for all the steps taken to lose a game)

<u>Stages of models trained</u>

Primary Dataset :

We used increment approach to train our models, i.e. first we started off with just n rows (ie 100 rows) to let players play against each other and captured their winning rate for Player-1,AgentX and Agent Y. Likewise 500, 1000, 2000, 3000, 4000, 5000, 10000 and 20000 rows were trained.

Tables to show the list of rows, wins by Player1(Agent-X), Player2(Agent-Y) and Agent-X's winning percentage including the draw. Draw match for Agent-X is also considered to get a penalty since it takes away his winning percentage.



selva standard reward/penality			Dataset # 2		
Rows	Player1 Wins (RL	Player2 Wins (Ra	Draw	% Player-1 win	time
100	14	7	2	60.87	<1 min
500	98	26	6	75.38	<3 min
1000	197	24	23	80.74	10 min
2000	483	49	9	89.28	35 min
3000	736	69	20	89.21	1.36 hours
4000	851	71	65	86.22	2.5 hours
5000	1111	99	48	88.31	3.6 hours
10000	2538	131	71	92.63	8.308423 hours
20000	5357	103	61	97.03	58.608312 hours
					2.442013 days

Charts

Chart-1: shows winning percent for Agent-X for rows from (1000, with increment of 1000 till 5000 rows)

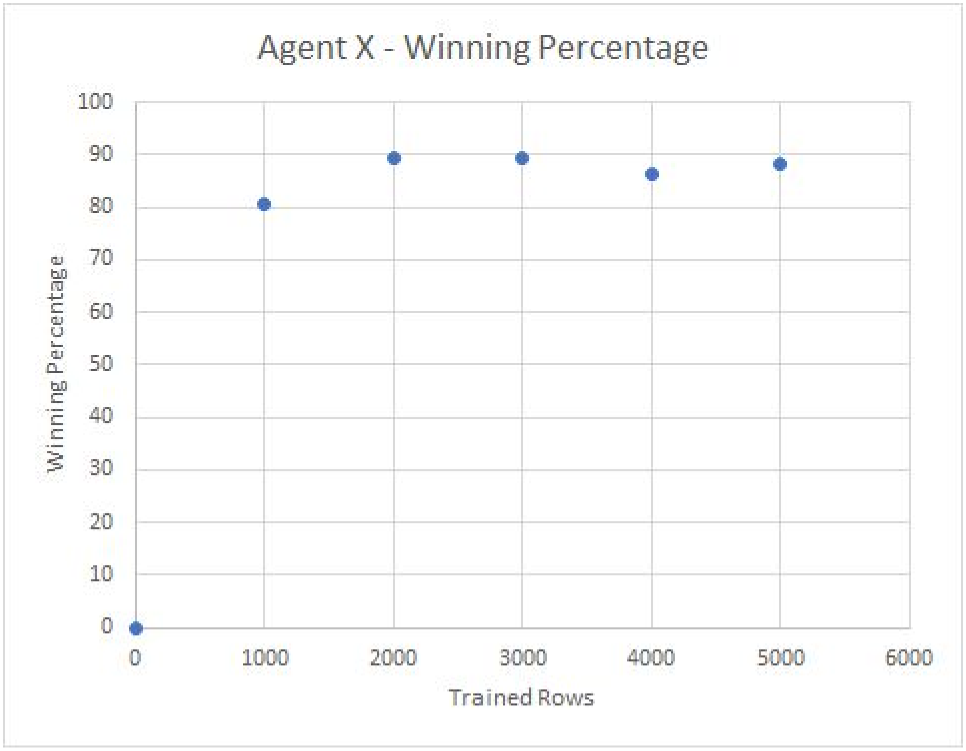


Chart-2: This Charts shows the time taken for each of those number of rows played.

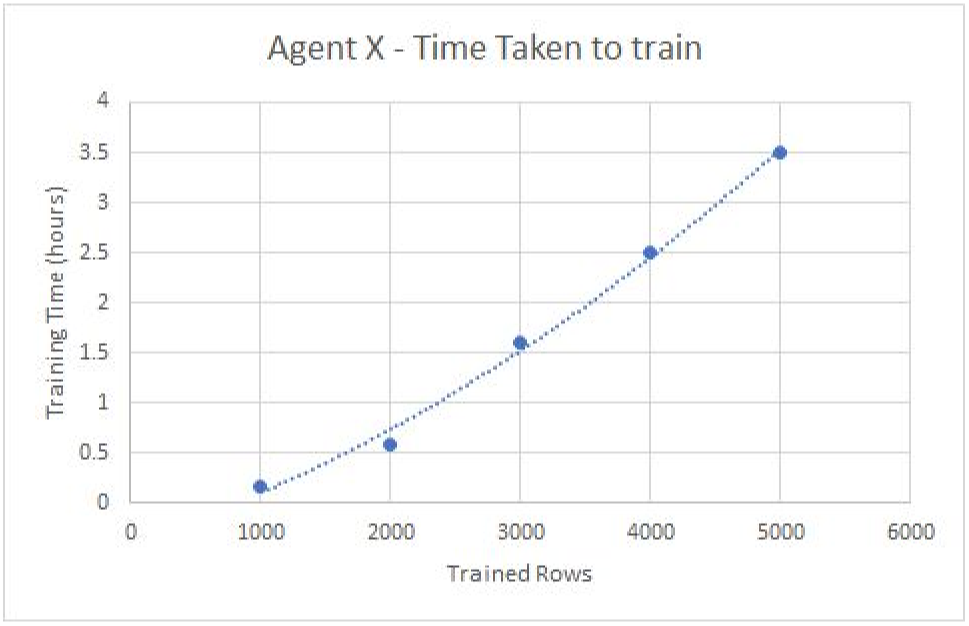


Chart-3: We pushed the limits to try 10000 and 20000 rows trained. This (20k) run took literally 2.5 days to run (ie 58.60 hours) compared to 5000 rows took 3.6 hours. Exponential increase in time.

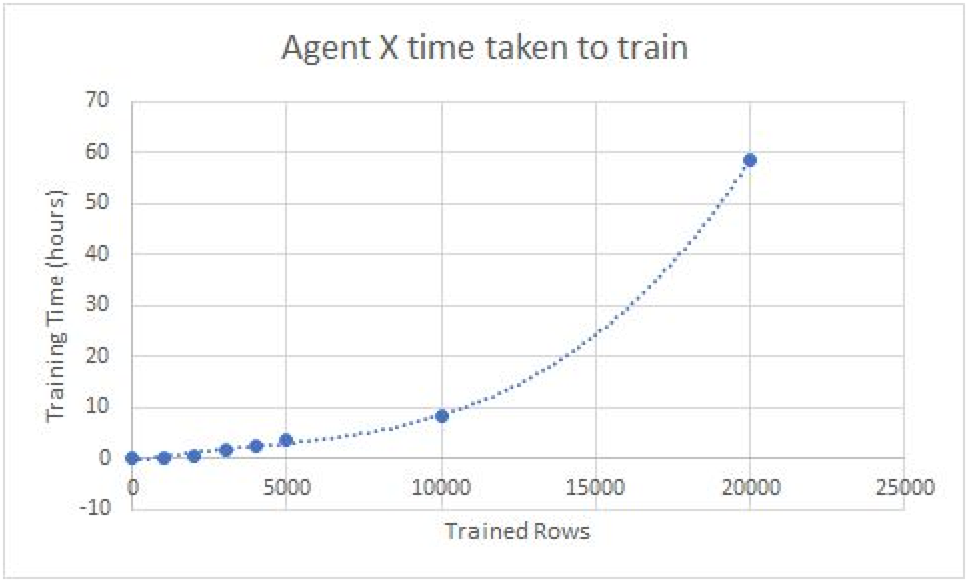
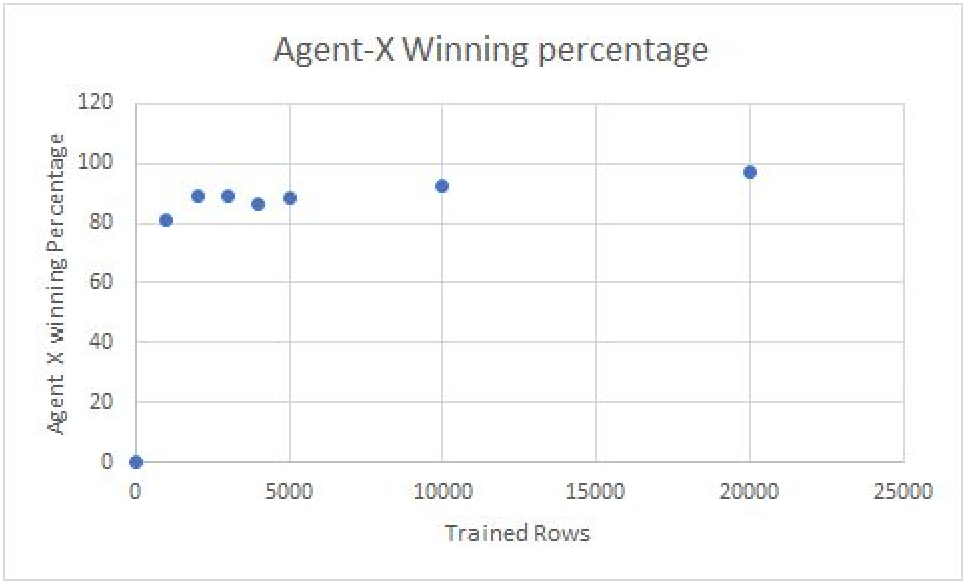
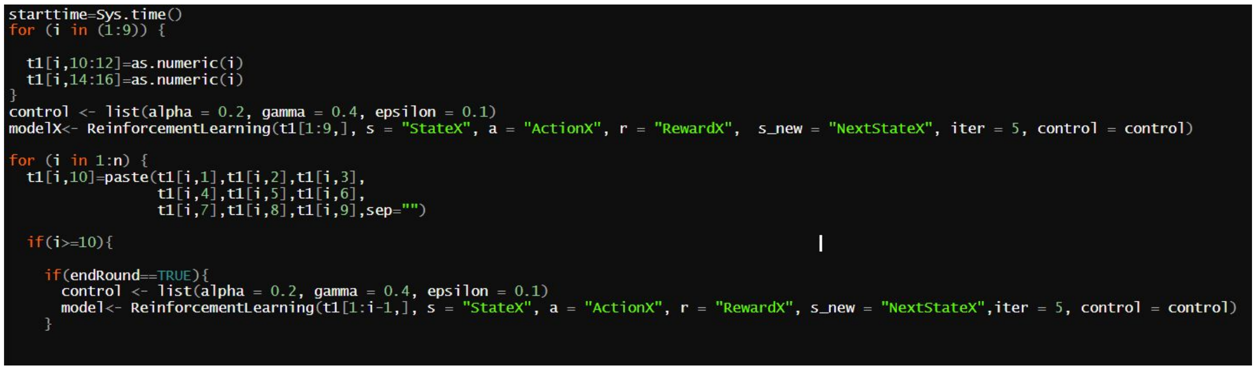


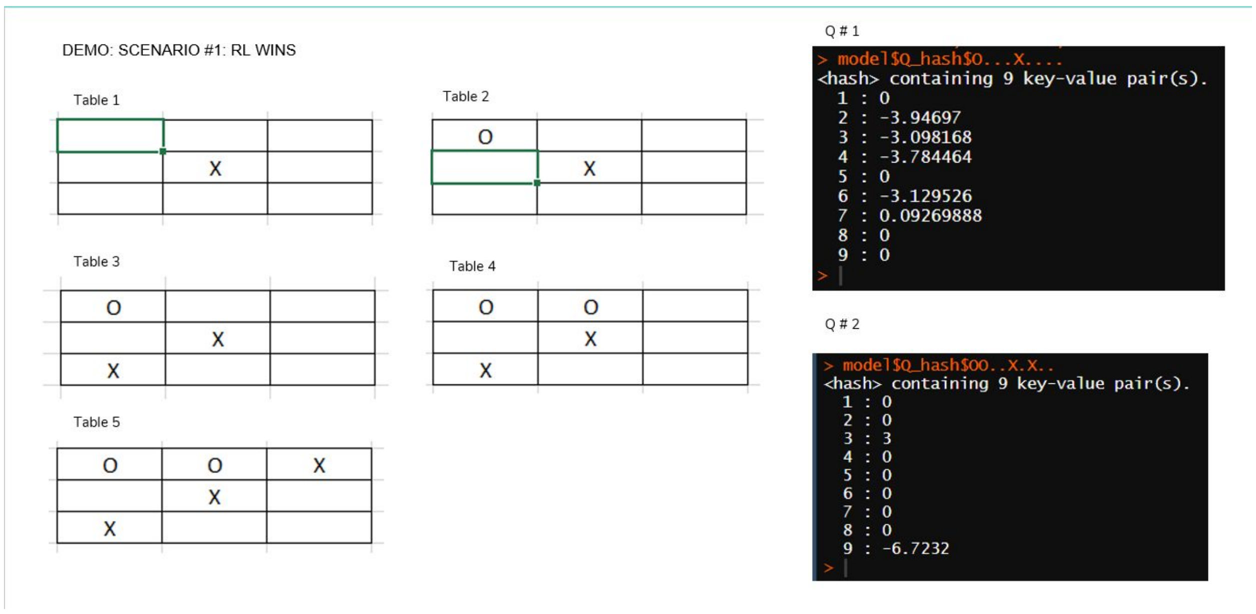
Chart-4: Winning percentage have increased to a maximum of 97% with 20k rows



Listed below is the crux of the code Used to train Agent X (RL) in Model 2 with Primary dataset.



```
starttime=Sys.time()
for (i in (1:9)) {

  t1[i,10:12]=as.numeric(i)
  t1[i,14:16]=as.numeric(i)
}
control <- list(alpha = 0.2, gamma = 0.4, epsilon = 0.1)
modelX<- ReinforcementLearning(t1[1:9,], s = "StateX", a = "ActionX", r = "RewardX",  s_new = "NextStateX", iter = 5, control = control)

for (i in 1:n) {
  t1[i,10]=paste(t1[i,1],t1[i,2],t1[i,3],
                 t1[i,4],t1[i,5],t1[i,6],
                 t1[i,7],t1[i,8],t1[i,9],sep="")

  if(i>=10){

    if(endRound==TRUE){
      control <- list(alpha = 0.2, gamma = 0.4, epsilon = 0.1)
      model<- ReinforcementLearning(t1[1:i-1,], s = "StateX", a = "ActionX", r = "RewardX", s_new = "NextStateX",iter = 5, control = control)
    }
```



DEMO: SCENARIO #1: RL WINS

Table 1

	X	

Table 2

O		
	X	

Table 3

O		
	X	
X		

Table 4

O	O	
	X	
X		

Table 5

O	O	X
	X	
X		

Q # 1

```
> model$Q_hash$O...X....
<hash> containing 9 key-value pair(s).
  1 : 0
  2 : -3.94697
  3 : -3.098168
  4 : -3.784464
  5 : 0
  6 : -3.129526
  7 : 0.09269888
  8 : 0
  9 : 0
> 
```

Q # 2

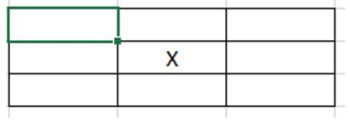
```
> model$Q_hash$OO..X.X..
<hash> containing 9 key-value pair(s).
  1 : 0
  2 : 0
  3 : 3
  4 : 0
  5 : 0
  6 : 0
  7 : 0
  8 : 0
  9 : -6.7232
> 
```

In Demo scenario #1,

- Step 1: RL (Agent X) plays first and takes 5th position and marks X (Table 1)
- Step 2: Random player picks 1st position, placing O (Table 2)
- Step 3: Before making the next move by our Agent X, he looks at the Q table (Q#1) and decides to make a move to position 7 because of its positive value (based on previous experience), that's the best spot to win so he puts X in position 7 (Table 3)
- Step 4: Random player pics position 2 (Table 4)
- Step 5: RL player looks at the Q table again (Q # 2) before his move, he finds position 3 with positive value of 3 (higher chance of winning) and he decides to put X in 3rd spot and wins the game (Table 5)

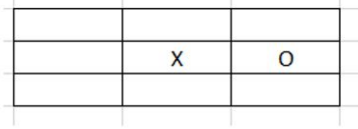
DEMO: SCENARIO #2: RANDOM PLAYER WINS

Table 1



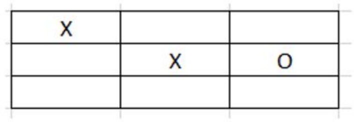
	X	

Table 2



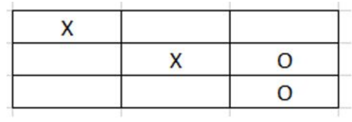
	X	O

Table 3



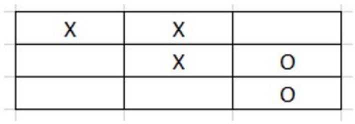
X		
	X	O

Table 4



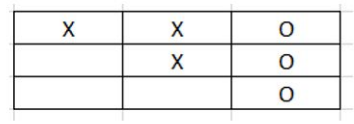
X		
	X	O
		O

Table 5



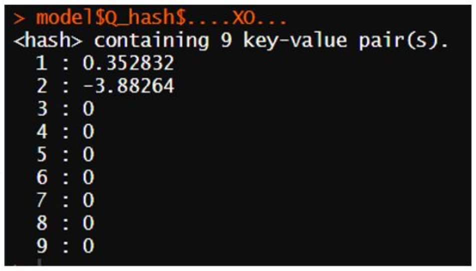
X	X	
	X	O
		O

Table 6



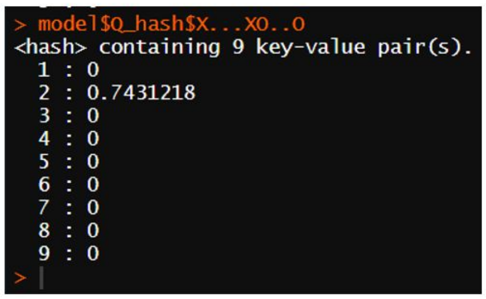
X	X	O
	X	O
		O

Q # 1



```
> model$Q_hash$....XO...
<hash> containing 9 key-value pair(s).
  1 : 0.352832
  2 : -3.88264
  3 : 0
  4 : 0
  5 : 0
  6 : 0
  7 : 0
  8 : 0
  9 : 0
```

Q # 2

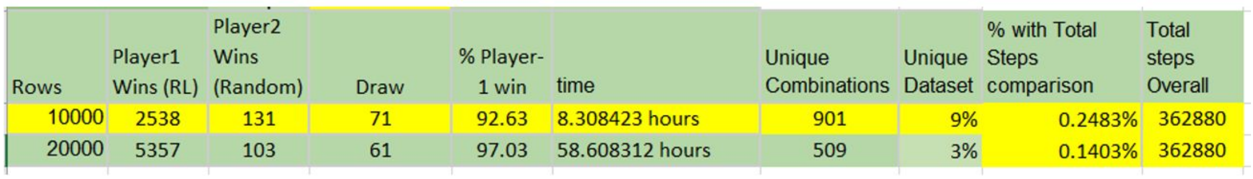


```
> model$Q_hash$X...XO..O
<hash> containing 9 key-value pair(s).
  1 : 0
  2 : 0.7431218
  3 : 0
  4 : 0
  5 : 0
  6 : 0
  7 : 0
  8 : 0
  9 : 0
> |
```

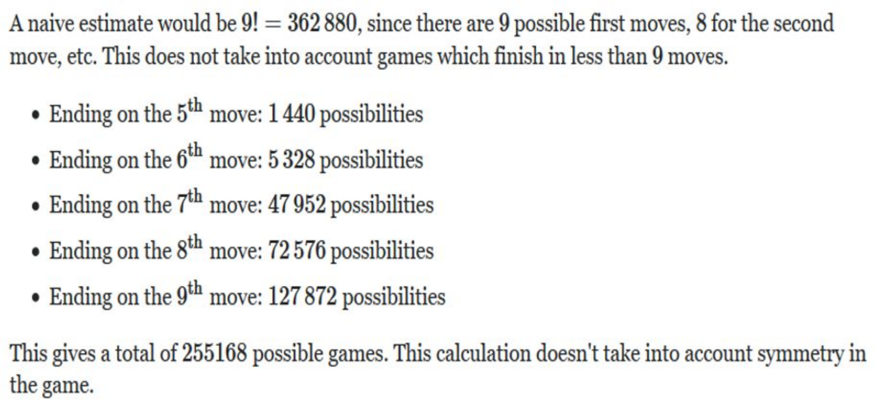
In Demo scenario #2,

- Step 1: RL (Agent X) plays first and takes 5th position and marks X (Table 1)
- Step 2: Random player picks 6th position, placing O (Table 2)
- Step 3: Before making the next move by our Agent X, he looks at the Q table (Q # 1) and decides to make a move to position 1st position because of its positive value (based on previous experience), that's the best spot to win so he puts X (Table 3)
- Step 4: Random player pics position 9 (Table 4)
- Step 5: RL player looks at the Q table (Q#2) again before his move, he finds position 2 with positive value (others positions are zeroes) and he decides to put X in 2nd spot (Table 5)
- Step 6: Random player places O in the position 3 and wins the game. (Table 6)
- In this scenario, we discovered that an RL (Agent X) player could not win because he has no values for all other positions in the Q table to choose from because he didn't have enough dataset that could give him the Q value for making a decision.

There are 362880 possible moves can be made in Tic Tac Toe, and our 20k trained rows only covered 901 possible unique combinations (there are duplicates, but it proves a point that our RL is getting smarter by playing the same winning strategy/sequence every time). Also 0.14% of the unique data set combinations were covered with 20k trained rows. It will take months upto a year if we set out to cover all possible combinations if RL is used to learn from each of its previous steps/games. Listed below is the table with the information about all possible moves



Rows	Player1 Wins (RL)	Player2 Wins (Random)	Draw	% Player-1 win	time	Unique Combinations	Unique Dataset	% with Total Steps comparison	Total steps Overall
10000	2538	131	71	92.63	8.308423 hours	901	9%	0.2483%	362880
20000	5357	103	61	97.03	58.608312 hours	509	3%	0.1403%	362880



A naive estimate would be $9! = 362\,880$, since there are 9 possible first moves, 8 for the second move, etc. This does not take into account games which finish in less than 9 moves.

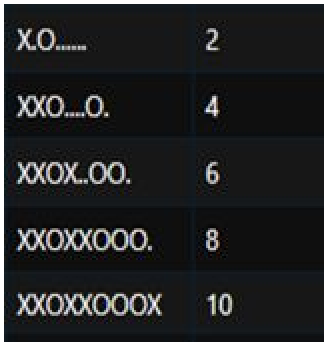
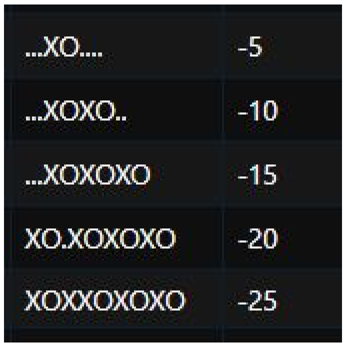
- Ending on the 5th move: 1 440 possibilities
- Ending on the 6th move: 5 328 possibilities
- Ending on the 7th move: 47 952 possibilities
- Ending on the 8th move: 72 576 possibilities
- Ending on the 9th move: 127 872 possibilities

This gives a total of 255168 possible games. This calculation doesn't take into account symmetry in the game.

<u>Another Dataset with various combinations</u>:

In this dataset, I tried to change a few of items to try out if I can improve the model from previous dataset by setting:

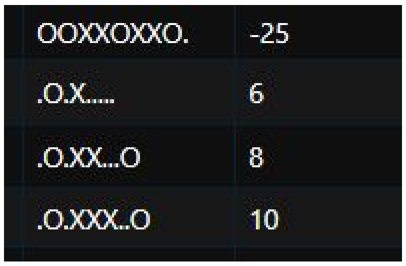
- Rewards to +10
- Penalty to -25
- Alpha = 0.5, Gamma = 0.5, Epsilon = 1.0
- Applying gradient descent approach to reward/penalty (gradual reward/penalize method)
- And repeated the training of 1000 rows, increment by 1000 to a maximum of 5000 rows

Agent-X gets rewarded for winning a game	Agent-X gets penalty for losing a game
Fig 1:  X.O...... 2 XXO....O. 4 XXOX..OO. 6 XXOXXOOO. 8 XXOXXOOOX 10 In Fig 1: If the game ends (wins) in 5 steps,	Fig 3:  ...XO.... -5 ...XOXO.. -10 ...XOXOXO -15 XO.XOXOXO -20 XOXXOXOXO -25 In Fig 3: If the game ends (loss) in 5 steps,

rewards are calculated (increment at each step) accordingly.

Step 1 will get +2
Step 2 will get +4
Step 3 will get +6
Step 4 will get +8
Step 5 will get +10 as rewards respectively.

Fig 2:



OOXXOXXO.	-25
.O.X.....	6
.O.XX...O	8
.O.XXX..O	10

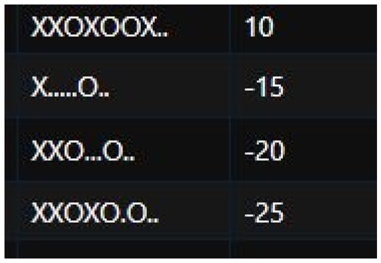
In Fig 2: If the game ends (wins) in 3 steps, rewards are calculated (increment at each step) accordingly.

Step 1 will get +6
Step 2 will get +8
Step 3 will get +10 as rewards respectively.
(I showed -25 to show previous game ending to distinguish this game's start)

penalty are calculated (severe penalty at each step leading to a game loss) accordingly

Step 1 will get -5
Step 1 will get -10
Step 1 will get -15
Step 2 will get -20
Step 3 will get -25 as penalty respectively.

Fig 4:

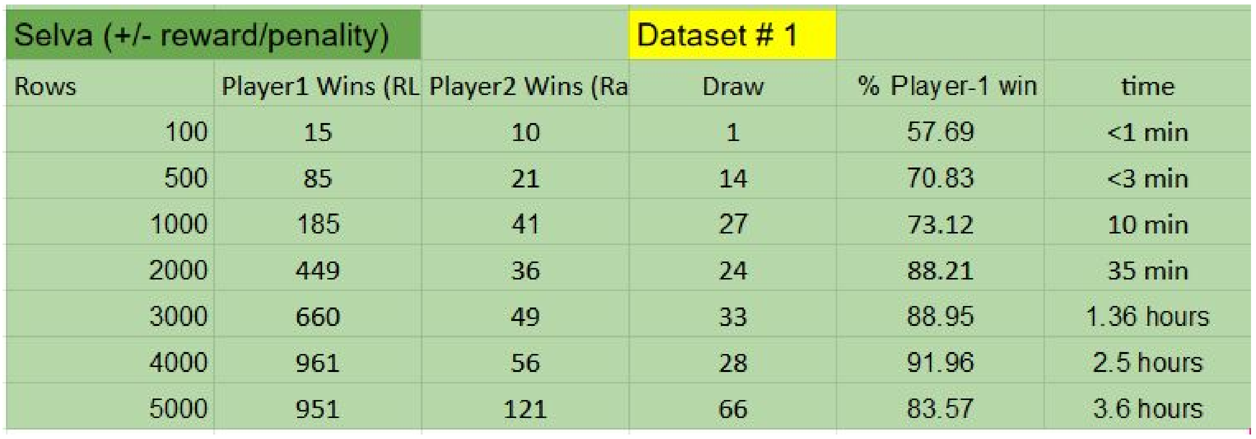


XXOXOOX..	10
X.....O..	-15
XXO...O..	-20
XXOXO.O..	-25

In Fig 4: If the game ends (loss) in 3 steps, penalty are calculated (severe penalty at each step leading to a game loss) accordingly

Step 1 will get -15
Step 2 will get -20
Step 3 will get -25 as penalty respectively.
(I showed +10 to show previous game ending to distinguish this game's start)

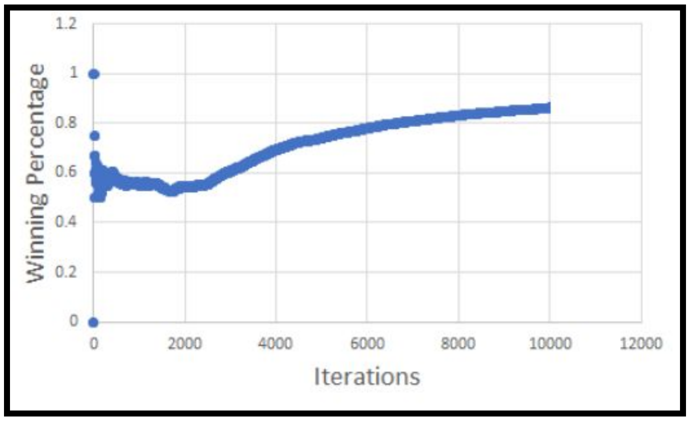
Number of rows trained, time taken and winning percent for Player 1:



Selva (+/- reward/penality)			Dataset # 1		
Rows	Player1 Wins (RL	Player2 Wins (Ra	Draw	% Player-1 win	time
100	15	10	1	57.69	<1 min
500	85	21	14	70.83	<3 min
1000	185	41	27	73.12	10 min
2000	449	36	24	88.21	35 min
3000	660	49	33	88.95	1.36 hours
4000	961	56	28	91.96	2.5 hours
5000	951	121	66	83.57	3.6 hours

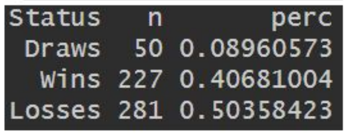
Model 3: Random vs. RL (Charles's)

Charles's model consisted of 10,000 iterations. Player Y's actions were chosen at random for all 10,000 iterations. For the first 2,500 iterations, Player X's actions were also chosen at random. For the last 7,500 iterations, Player X used reinforcement learning to decide on which action to take. The following graph displays Player X's winning percentage vs. the number of iterations:



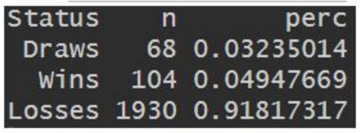
As can be seen, Player X's winning percentage steadily improved for the last 7500 iterations. Player X's winning percentage for the first 2500 iterations was a little over 50%. For the last 7500 iterations, Player X's winning percentage was almost 92% (1930 ÷ :

First 2500 iterations
(Random vs. Random)



```
Status    n       perc
 Draws   50 0.08960573
  Wins  227 0.40681004
Losses  281 0.50358423
```

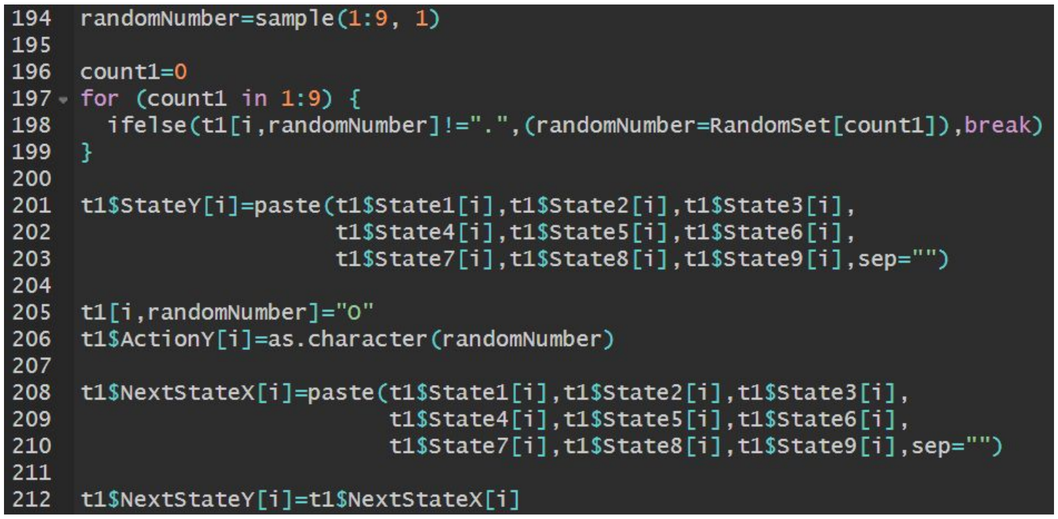
Last 7500 iterations
(RL vs. Random)



```
Status     n       perc
 Draws    68 0.03235014
  Wins   104 0.04947669
Losses  1930 0.91817317
```

Player Y

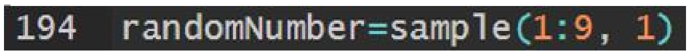
Player Y's decisions were chosen at random during all iterations. The following code chunk displays this fact.



```
194  randomNumber=sample(1:9, 1)
195
196  count1=0
197  for (count1 in 1:9) {
198    ifelse(t1[i,randomNumber]!=".",(randomNumber=RandomSet[count1]),break)
199  }
200
201  t1$StateY[i]=paste(t1$State1[i],t1$State2[i],t1$State3[i],
202                     t1$State4[i],t1$State5[i],t1$State6[i],
203                     t1$State7[i],t1$State8[i],t1$State9[i],sep="")
204
205  t1[i,randomNumber]="O"
206  t1$ActionY[i]=as.character(randomNumber)
207
208  t1$NextStateX[i]=paste(t1$State1[i],t1$State2[i],t1$State3[i],
209                         t1$State4[i],t1$State5[i],t1$State6[i],
210                         t1$State7[i],t1$State8[i],t1$State9[i],sep="")
211
212  t1$NextStateY[i]=t1$NextStateX[i]
```

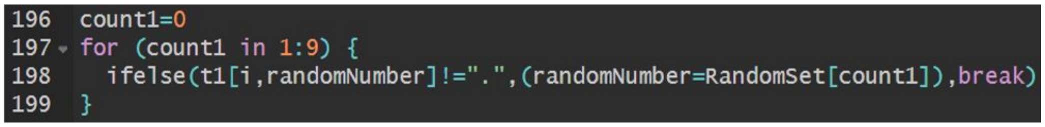
Explanation of code:

- Notice how the variable **randomNumber** represents the chosen TicTacToe cell. On line 194 of the below code chunk, you can see that **randomNumber** is a random number between 1 – 9:



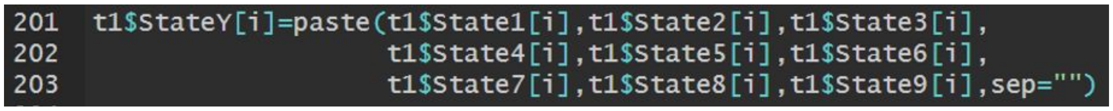
```
194  randomNumber=sample(1:9, 1)
```

- Lines 196 → 199 of the code chunk shown above ensures that Player Y's chosen TicTacToe cell doesn't already contain and X or an O. If it does, then Player Y will choose a diffeerent action at random.



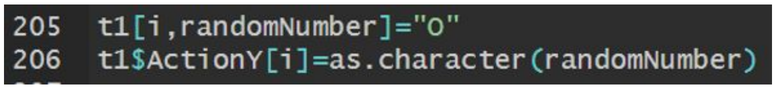
```
196  count1=0
197  for (count1 in 1:9) {
198    ifelse(t1[i,randomNumber]!=".",(randomNumber=RandomSet[count1]),break)
199  }
```

- Lines 201 - 203 then takes all of the "states" and combines them to create the overall state of the game *before* Player Y makes its move.



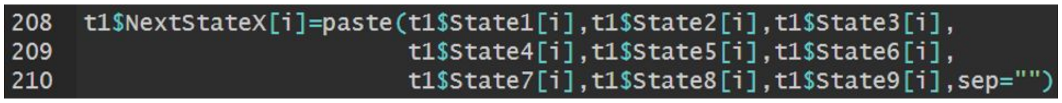
```
201  t1$StateY[i]=paste(t1$State1[i],t1$State2[i],t1$State3[i],
202                      t1$State4[i],t1$State5[i],t1$State6[i],
203                      t1$State7[i],t1$State8[i],t1$State9[i],sep="")
```

- Line 205 of the code places an "O" into the randomly chosen cell, while Line 206 records which cell the "O" was placed into.



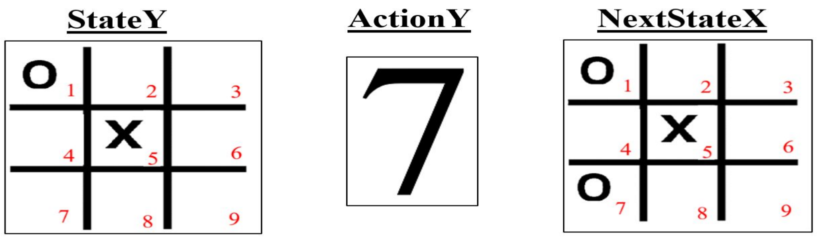
```
205  t1[i,randomNumber]="O"
206  t1$ActionY[i]=as.character(randomNumber)
```

- Now that Player Y has chosen an action, the state must be updated to reflect a new state. This is done in lines 208 – 210. This new state, *NextStateX*, represents the state *after* Player Y has made its move and, thus, also representes the *next state* that PlayerX will be confronted with.



```
208  t1$NextStateX[i]=paste(t1$State1[i],t1$State2[i],t1$State3[i],
209                         t1$State4[i],t1$State5[i],t1$State6[i],
210                         t1$State7[i],t1$State8[i],t1$State9[i],sep="")
```

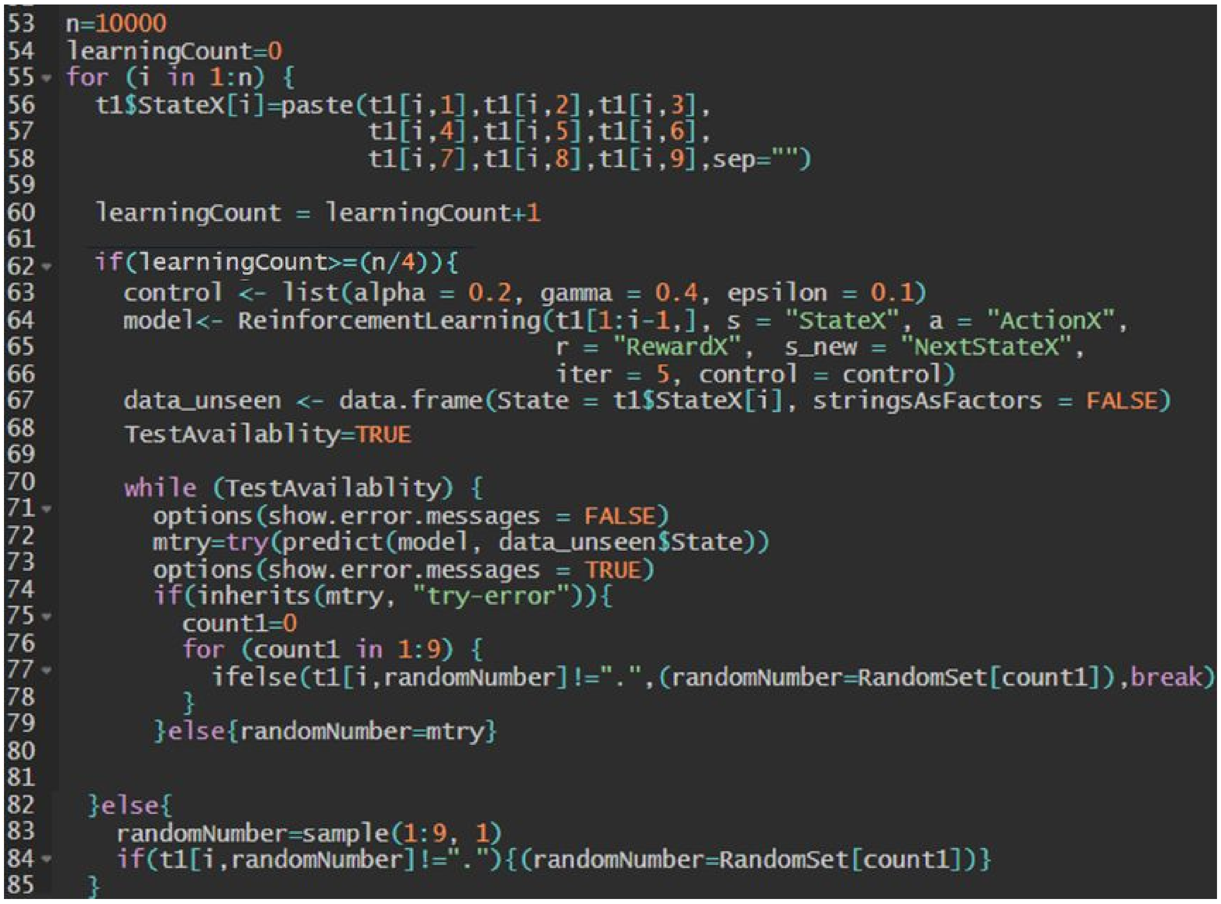
The following illustration exemplifies this process:



- If StateY is "O…X…." (i.e., *State1* is "O", *State5* "X", and the rest of the individual states are empty), and Player Y's chosen action is 7, then an "O" will be placed into the *seventh* column and the state will be updated to "O…X.O.." to reflect this change.

Player X

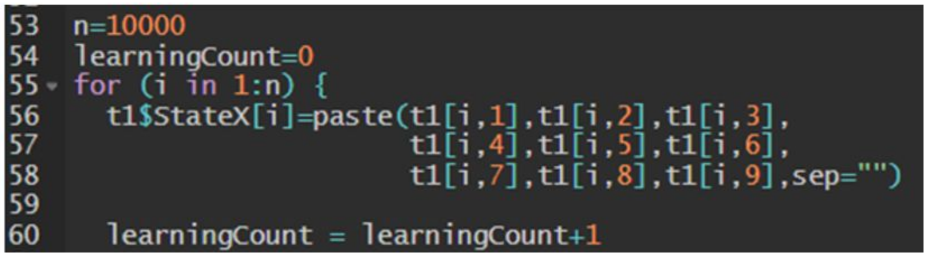
The following code chunk displays how Player X chose actions.



```r
53   n=10000
54   learningCount=0
55   for (i in 1:n) {
56     t1$StateX[i]=paste(t1[i,1],t1[i,2],t1[i,3],
57                        t1[i,4],t1[i,5],t1[i,6],
58                        t1[i,7],t1[i,8],t1[i,9],sep="")
59
60     learningCount = learningCount+1
61
62     if(learningCount>=(n/4)){
63       control <- list(alpha = 0.2, gamma = 0.4, epsilon = 0.1)
64       model<- ReinforcementLearning(t1[1:i-1,], s = "StateX", a = "ActionX",
65                                     r = "RewardX",  s_new = "NextStateX",
66                                     iter = 5, control = control)
67       data_unseen <- data.frame(State = t1$StateX[i], stringsAsFactors = FALSE)
68       TestAvailablity=TRUE
69
70       while (TestAvailablity) {
71         options(show.error.messages = FALSE)
72         mtry=try(predict(model, data_unseen$State))
73         options(show.error.messages = TRUE)
74         if(inherits(mtry, "try-error")){
75           count1=0
76           for (count1 in 1:9) {
77             ifelse(t1[i,randomNumber]!=".",(randomNumber=RandomSet[count1]),break)
78           }
79         }else{randomNumber=mtry}
80
81
82     }else{
83       randomNumber=sample(1:9, 1)
84       if(t1[i,randomNumber]!="."){(randomNumber=RandomSet[count1])}
85     }
```

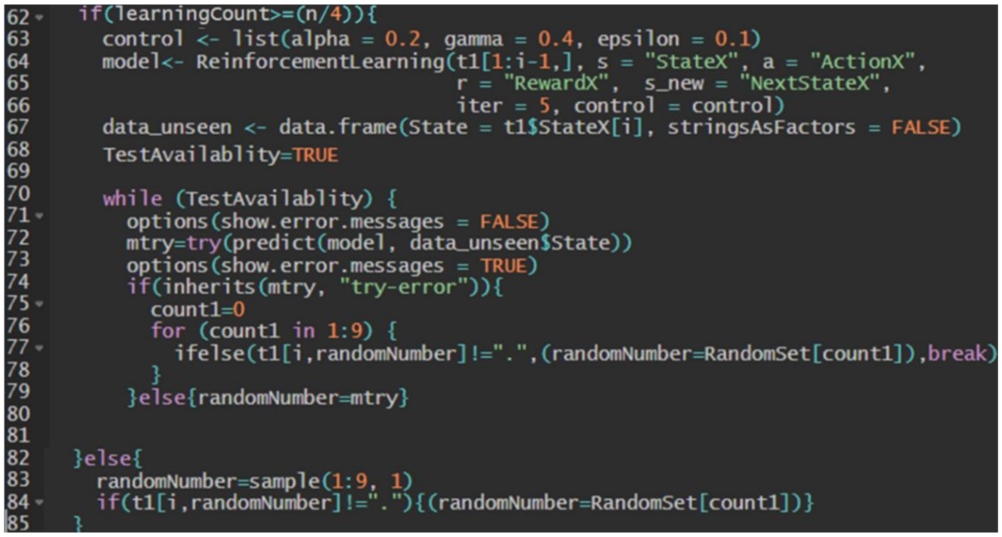
<u>Explanation of code</u>:

- As can be seen on line 54, we set the variable **learningCount** to 0 before we begin training the model. Then, in each iteration, we incremented **learningCount** by 1 (as can be seen in line 60 fo the code section copied & pasted again below).



```r
53   n=10000
54   learningCount=0
55   for (i in 1:n) {
56     t1$StateX[i]=paste(t1[i,1],t1[i,2],t1[i,3],
57                        t1[i,4],t1[i,5],t1[i,6],
58                        t1[i,7],t1[i,8],t1[i,9],sep="")
59
60     learningCount = learningCount+1
```

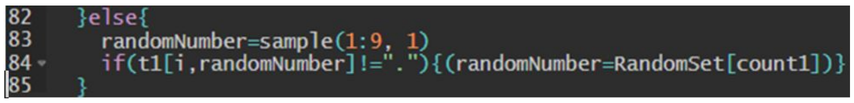
- As can be seen on line 62 of the code, we told R to use reinforcement learning to govern Player X's actions only when **learningCount** is greater than or equal to the total number of iterations divided by 4 (i.e., when **learningCount** ≥ 2500).



```
62    if(learningCount>=(n/4)){
63      control <- list(alpha = 0.2, gamma = 0.4, epsilon = 0.1)
64      model<- ReinforcementLearning(t1[1:i-1,], s = "StateX", a = "ActionX",
65                                    r = "RewardX",  s_new = "NextStateX",
66                                    iter = 5, control = control)
67      data_unseen <- data.frame(State = t1$StateX[i], stringsAsFactors = FALSE)
68      TestAvailablity=TRUE
69
70      while (TestAvailablity) {
71        options(show.error.messages = FALSE)
72        mtry=try(predict(model, data_unseen$State))
73        options(show.error.messages = TRUE)
74        if(inherits(mtry, "try-error")){
75          count1=0
76          for (count1 in 1:9) {
77            ifelse(t1[i,randomNumber]!=".",(randomNumber=RandomSet[count1]),break)
78          }
79        }else{randomNumber=mtry}
80
81
82    }else{
83      randomNumber=sample(1:9, 1)
84      if(t1[i,randomNumber]!="."){(randomNumber=RandomSet[count1])}
85    }
```

- You can see in the above code section that we turn off error messages by using the command **show.error.messages = FALSE**. We do this because there is a chance that the State used within the **predict()** function (on line 72) has not yet been seen by the model. If an when this happens, lines 74 – 79 tells R to randomly choose an action for Player X. Player X will then receive a reward or penalty for this action, based on the outcome. Therefore, Player X's *optimal policy* will be updated for the *state* that was presented to the **predict()** function in line 72. In other words, if Player X's randomly chosen action results in a reward, then it will likely choose that action when it is again presented with the same state. Likewise, if Player X's randomly chosen action results in a penalty, it will choose a *different* action when it is again presented with the same state. If the state presented to the **predict()** does *not* produce an error (i.e., if the model *has* been presented with this state in the past), then what is currently the *optimal policy* will be chosen as Player X's actions. The code for this is shown on line 79 of the code.

- When **learningCount** is *not* greater than n ÷ 4, the *else* statement in lines 82 – 85 of the below code chunk are used to govern Player X's actions (randomly chosen actions):

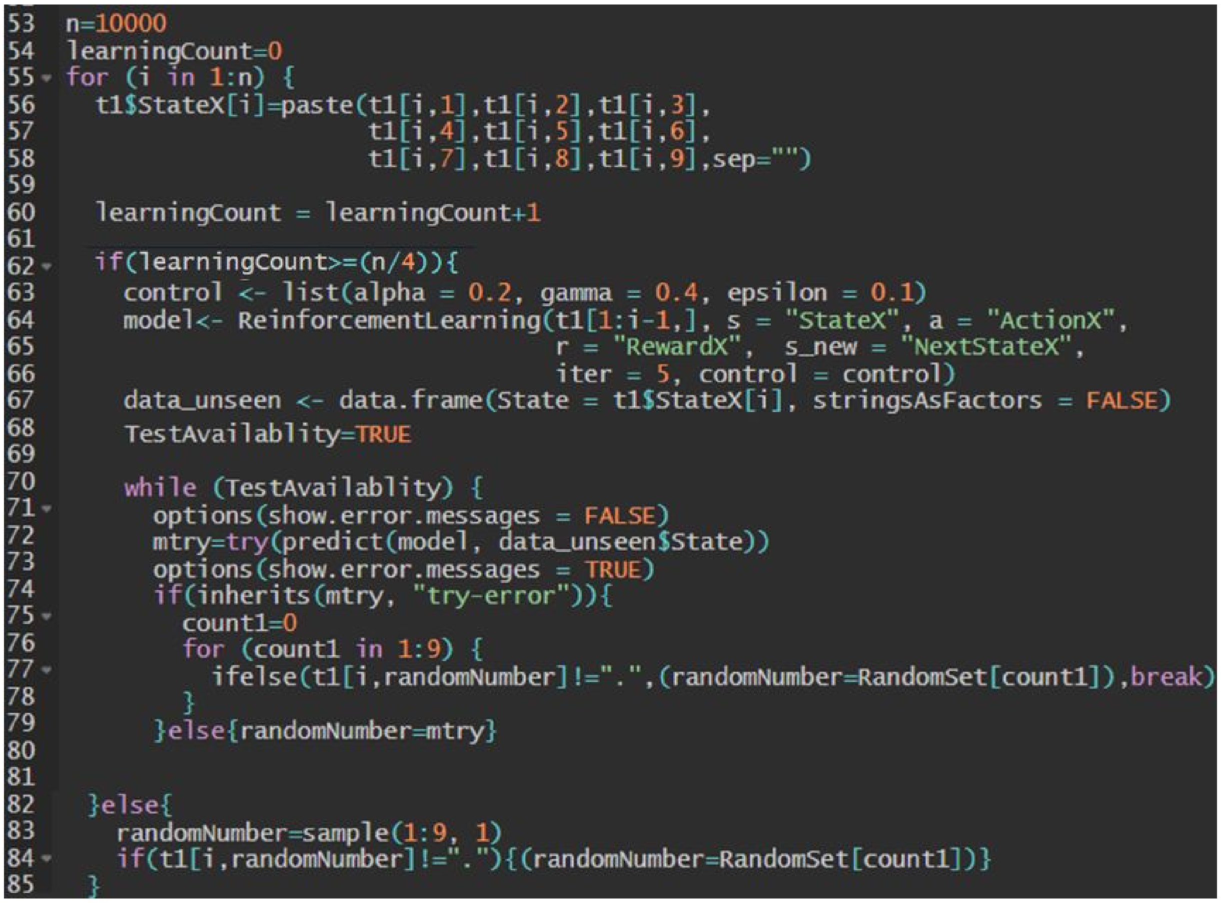


```
82    }else{
83      randomNumber=sample(1:9, 1)
84      if(t1[i,randomNumber]!="."){(randomNumber=RandomSet[count1])}
85    }
```

Model 4: RL vs RL (Qing's)

The last model we tried is to use RL to analyze and generate data at the same time for both agents.

Player X & Y

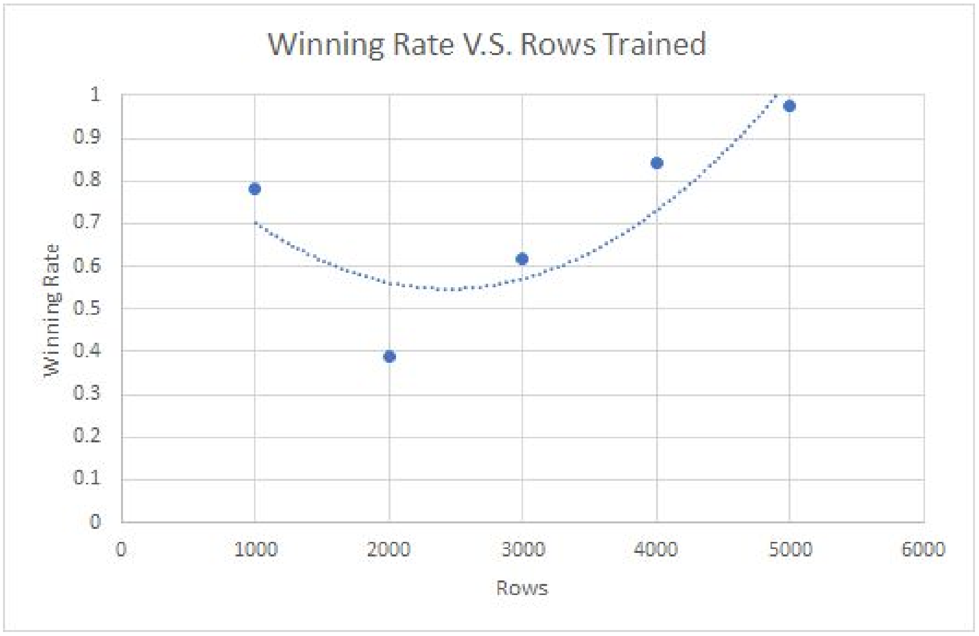


```r
n=10000
learningCount=0
for (i in 1:n) {
  t1$StateX[i]=paste(t1[i,1],t1[i,2],t1[i,3],
                     t1[i,4],t1[i,5],t1[i,6],
                     t1[i,7],t1[i,8],t1[i,9],sep="")

  learningCount = learningCount+1

  if(learningCount>=(n/4)){
    control <- list(alpha = 0.2, gamma = 0.4, epsilon = 0.1)
    model<- ReinforcementLearning(t1[1:i-1,], s = "StateX", a = "ActionX",
                                  r = "RewardX",  s_new = "NextStateX",
                                  iter = 5, control = control)
    data_unseen <- data.frame(State = t1$StateX[i], stringsAsFactors = FALSE)
    TestAvailablity=TRUE

    while (TestAvailablity) {
      options(show.error.messages = FALSE)
      mtry=try(predict(model, data_unseen$State))
      options(show.error.messages = TRUE)
      if(inherits(mtry, "try-error")){
        count1=0
        for (count1 in 1:9) {
          ifelse(t1[i,randomNumber]!=".",(randomNumber=RandomSet[count1]),break)
        }
      }else{randomNumber=mtry}


  }else{
    randomNumber=sample(1:9, 1)
    if(t1[i,randomNumber]!="."){(randomNumber=RandomSet[count1])}
  }
```

The model we used to train agent X makes no difference compared to the two models before this. But for agent Y, we used another RL model instead of generating a random dataset.

Policy change

From picture 5, we can see there is a significant increase in winning rate. It reaches a 95% winning rate at 4000 rows of data.



Picture 5

In the first 2000 rows, we could observe a decrease of the winning rate. That was caused by the policy for drawing.

Since Agent X always goes first, so X has a higher chance to win compared to Y - who always goes second. So whenever Agent Y has a draw, we reward him a little bit, because it's hard for Y to win. But when agent X has a draw, we will give him some penalty. Because its performance is not good enough.

Just like what we played at the beginning of the presentation. There is no absolute winning route for the second player, so what happened in the dataset was: Agent Y stops trying to win, it always chooses to draw. The decrease in the curve is just because of the number of draw significant increases.

After we observed the sequence, we changed the reward policy. Now it will penalize both agents if the draw happens.

From the curve after 2000, we could observe the winning rate keep increasing. Which means this modification helps with the result.

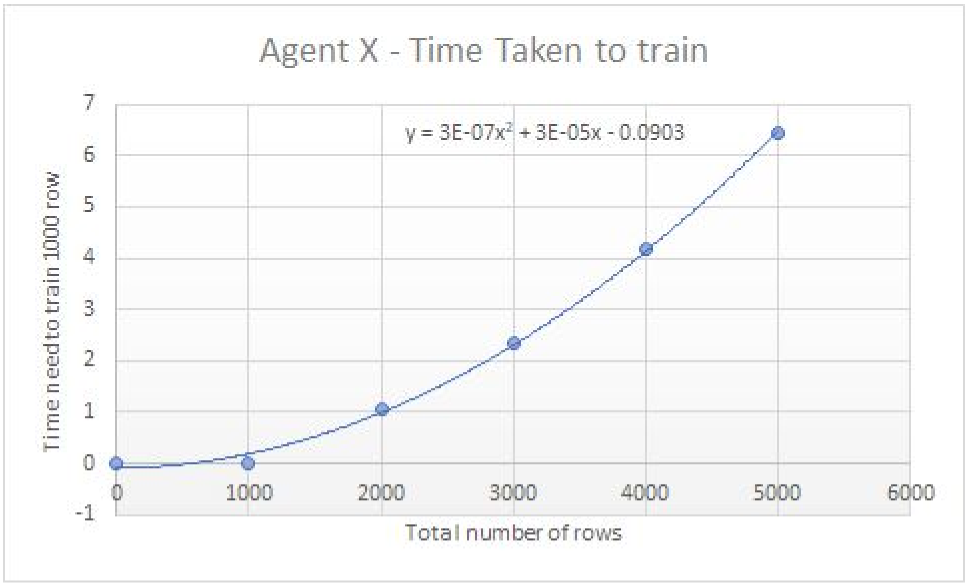
Advantages

There are two advantages for using this model:

First, since both of the agents are controlled by RL, so they both know how to dodge bad moves. Which means there will be less duplicate moves compared to random generate datasets.

Second. Since RL has a smart opponent, it will learn faster. More losses agent X have, more winning RL will perform in the future. It learns more combinations.

The major disadvantage for this model is it requires an even longer time to run compared to the other models. From picture 4, we can see the time it takes increase exponentially when the number of rows increases.



Picture 4

Conclusion

For the first model by Bob. Random generated dataset takes the shortest time, but the winning rate is under expectation. This model only works when there is insufficient time. So we will not consider this model in our next step.

The second model by Selva's takes longer, but the winning rate is much better.
Charles's third model takes less time than the second one, but the winning rate reduced a little too.
Between the second model and the third model, there is a trade off between the time and winning rate. Will consider both in our Black Jack example.

The last model by Qing takes the longest time. Although the winning rate is much better then the others. The time it takes is significantly larger than the others as well. If we have enough time to run this model, we will definitely go with it. If there is not enough, then we could run one part of this model to check our results made by the other models.

Challenges

The challenge for this part is all about the computer performance. We have a lot of thoughts which could possibly increase the model efficiency. But even a single change will take a long time to run and observe the result. And the best combination of Alpha, Gamma and Epsilon was always an experiment and we settled with alpha = 0.2, gamma = 0.4, epsilon = 0.1.

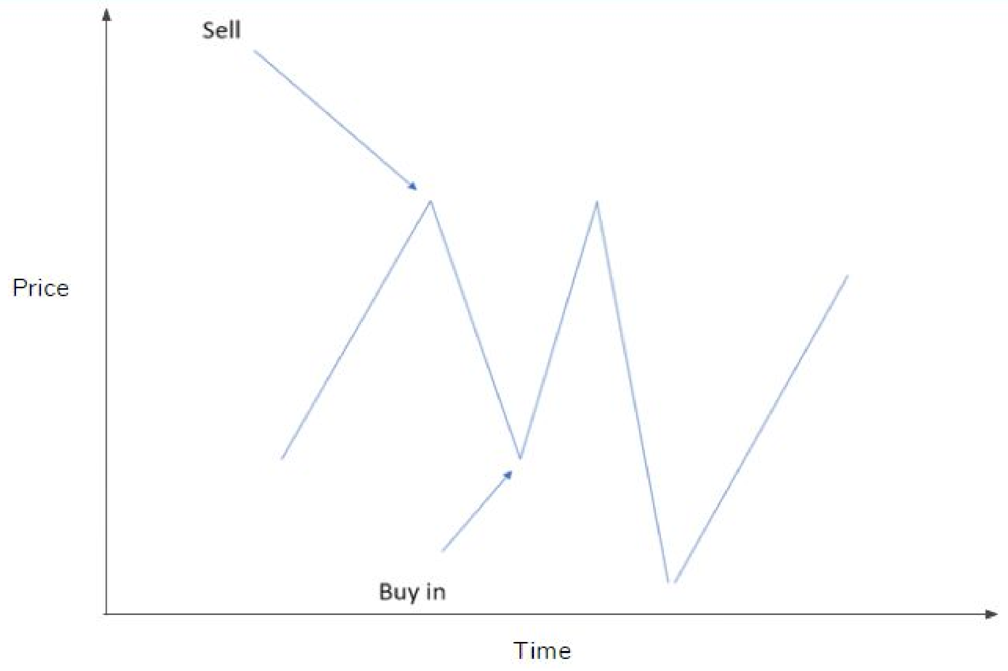
Project 2: STOCK MARKET

The logic behind

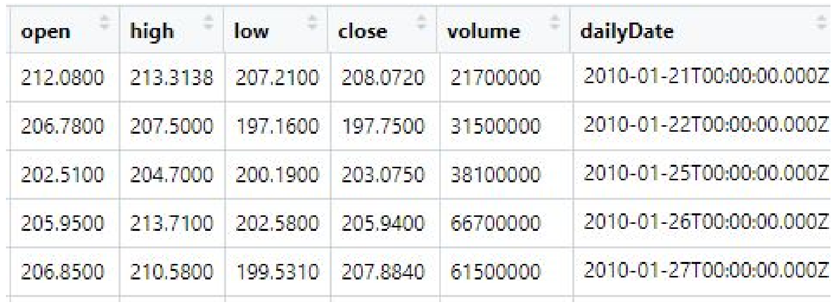
To apply RL in the stock market, we need to find out what are the “Action”, “State”, “Policy" and "Reward" in stock.

Our target is to build an agent for the stock market, the AI agent can sell and buy stock automatically. So “buy” and “sell” are the actions.

We want to sell the stock before the price decreases. We want to buy stock before the price increases. This is the policy.
If the agent buys stock just before the stock price decreases or sells stock before price increase, it will be punished. If the agent buys stock before price increases or sells before decrease, we will give it some rewards.



Everything else are the “States”:



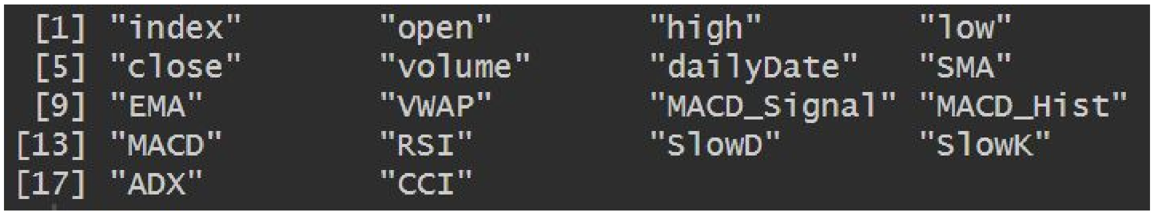
open	high	low	close	volume	dailyDate
212.0800	213.3138	207.2100	208.0720	21700000	2010-01-21T00:00:00.000Z
206.7800	207.5000	197.1600	197.7500	31500000	2010-01-22T00:00:00.000Z
202.5100	204.7000	200.1900	203.0750	38100000	2010-01-25T00:00:00.000Z
205.9500	213.7100	202.5800	205.9400	66700000	2010-01-26T00:00:00.000Z
206.8500	210.5800	199.5310	207.8840	61500000	2010-01-27T00:00:00.000Z

RL needs a sequence. It only works when the states repeat. But we only have 2000 rows of data, the numbers with decimals rarely repeat.

To solve this problem, we have two solutions.
The first way is to round the numbers. If we round the number to tenth, it will show the repeats. But we didn't choose this solution because of the limit of data and longer training time.
The second way is to transfer the numbers to binaries just like the "X" and "O" in Tic-tac-toe.

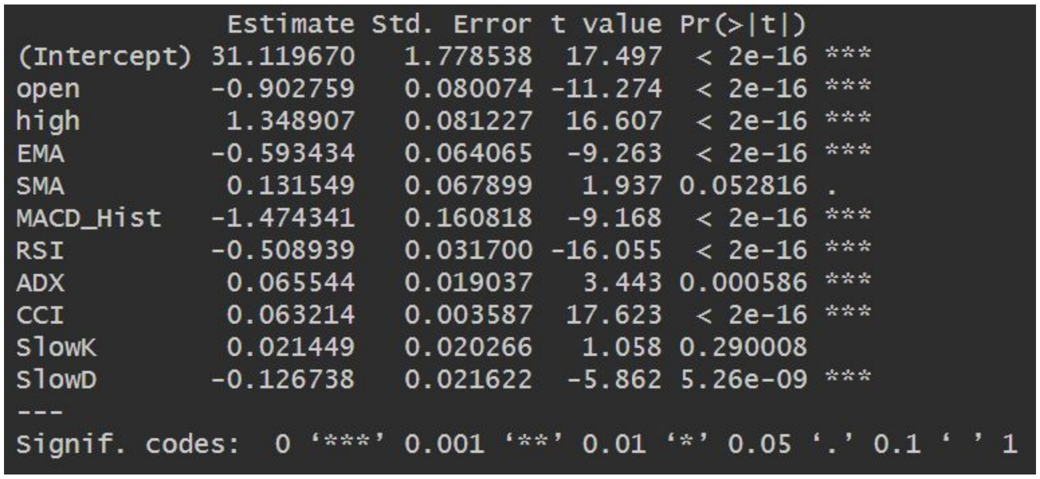
Stock Indicators

The following print out shows the features that were included within the stock data we obtained:



```
 [1] "index"       "open"        "high"        "low"
 [5] "close"       "volume"      "dailyDate"   "SMA"
 [9] "EMA"         "VWAP"        "MACD_Signal" "MACD_Hist"
[13] "MACD"        "RSI"         "SlowD"       "SlowK"
[17] "ADX"         "CCI"
```

The first question we need to answer is, *Which of these variables significantly affects a stock's closing price?* To answer this question, we fitted a series of linear regression models. The following printout shows each of these variables, along with their p-values (which represent how significantly changes in those variables affect closing price):



```
             Estimate Std. Error t value Pr(>|t|)    
(Intercept) 31.119670   1.778538  17.497  < 2e-16 ***
open        -0.902759   0.080074 -11.274  < 2e-16 ***
high         1.348907   0.081227  16.607  < 2e-16 ***
EMA         -0.593434   0.064065  -9.263  < 2e-16 ***
SMA          0.131549   0.067899   1.937 0.052816 .  
MACD_Hist   -1.474341   0.160818  -9.168  < 2e-16 ***
RSI         -0.508939   0.031700 -16.055  < 2e-16 ***
ADX          0.065544   0.019037   3.443 0.000586 ***
CCI          0.063214   0.003587  17.623  < 2e-16 ***
SlowK        0.021449   0.020266   1.058 0.290008    
SlowD       -0.126738   0.021622  -5.862 5.26e-09 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Notice how this model shows that SlowK does *not* significantly affect the closing price. The reason we decided to include it in our RL model is because the attributes SlowD (which *is* significant) is only useful when it can be compared to SlowK: A *buy signal* occurs for a stock if SlowD is less than SlowK ***and*** if SlowK is less than 20.

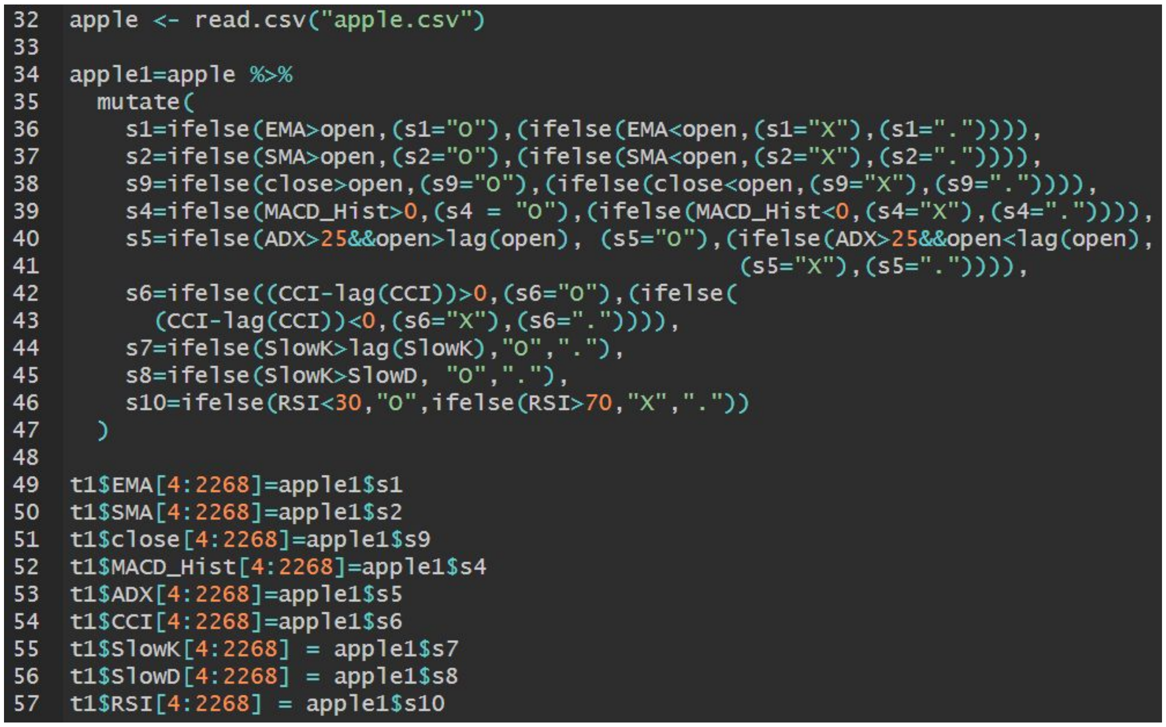
The next thing we needed to determine is how each of these attributes affects a stock's performance. We did some research to determine the following rules.

Attributes	**Rules**
MACD (Moving Average Convergence Divergence)	● If MACD_Hist is positive, buy the stock ● If MACD_Hist is negative, sell the stock ● If MACD_Hist is = 0, hold the stock
RSI (Relative Strength Index)	● If RSI ≤ 30, the stock is *under*valued ● If RSI ≥ 70, the stock is *over*valued
CCI (Commodity Channel Index)	● An *uptrend* in stock value (price) is associated with an *increasing* CCI value ● An *downtrend* in stock value (price) is associated with a *decreasing* CCI value

ADX (Average Directional Movement Index)	● An ADX value between 0 – 25 represents there is a weak trend ● An ADX value between 26 – 50 represents a strong trend ● An ADX value between 51 – 75 represents a very strong trend ● An ADX value between 76 – 100 represents an extremely strong trend
SlowD & SlowK (Slow Stochastic Operators)	● If SlowK < SlowD, and if SlowK > 80, sell the stock ● If SlowK > SlowD, and if SlowK < 20, buy the stock

Model 1: With 9 Parameters

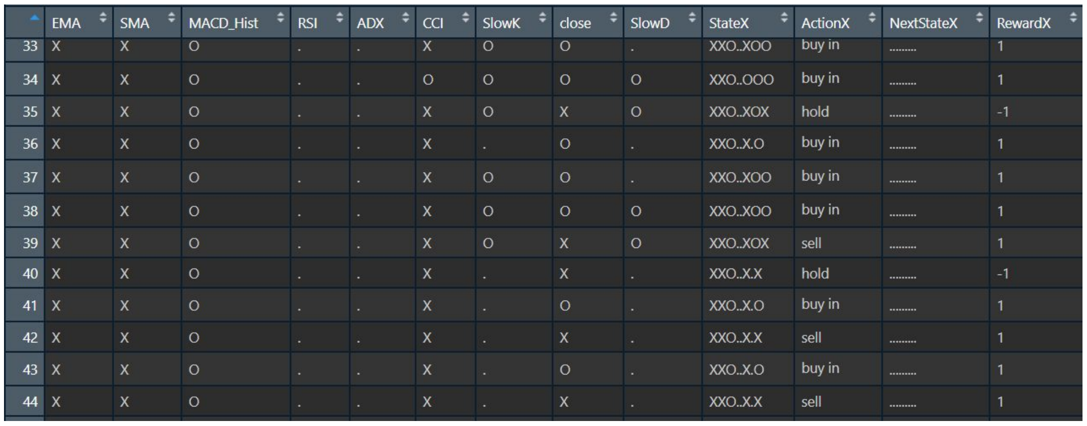
Now we'll discuss how we implemented these rules in our reinforcement learning model. We appended the Apple stocks data with columns s1, s2, …, s10. These values in these new columns correspond to how the aforementioned rules apply to each stock attribute:



```r
apple <- read.csv("apple.csv")

apple1=apple %>%
  mutate(
    s1=ifelse(EMA>open,(s1="O"),(ifelse(EMA<open,(s1="X"),(s1=".")))),
    s2=ifelse(SMA>open,(s2="O"),(ifelse(SMA<open,(s2="X"),(s2=".")))),
    s9=ifelse(close>open,(s9="O"),(ifelse(close<open,(s9="X"),(s9=".")))),
    s4=ifelse(MACD_Hist>0,(s4 = "O"),(ifelse(MACD_Hist<0,(s4="X"),(s4=".")))),
    s5=ifelse(ADX>25&&open>lag(open), (s5="O"),(ifelse(ADX>25&&open<lag(open),
                                                  (s5="X"),(s5=".")))),
    s6=ifelse((CCI-lag(CCI))>0,(s6="O"),(ifelse(
      (CCI-lag(CCI))<0,(s6="X"),(s6=".")))),
    s7=ifelse(SlowK>lag(SlowK),"O","."),
    s8=ifelse(SlowK>SlowD, "O","."),
    s10=ifelse(RSI<30,"O",ifelse(RSI>70,"X","."))
  )

t1$EMA[4:2268]=apple1$s1
t1$SMA[4:2268]=apple1$s2
t1$close[4:2268]=apple1$s9
t1$MACD_Hist[4:2268]=apple1$s4
t1$ADX[4:2268]=apple1$s5
t1$CCI[4:2268]=apple1$s6
t1$SlowK[4:2268] = apple1$s7
t1$SlowD[4:2268] = apple1$s8
t1$RSI[4:2268] = apple1$s10
```

So for example, line 39 of the displayed code shows that the character assigned to s4 for a particular stock depends on its MACD_Hist value: If MACD_Hist is positive, s4 is assigned an "O". If MACD_Hist is negative, s4 is assigned an "X". If it's 0, s4 is assigned a dot.

As shown in lines 49 → 57, these new columns are then copied over to the empty dataframe t1:

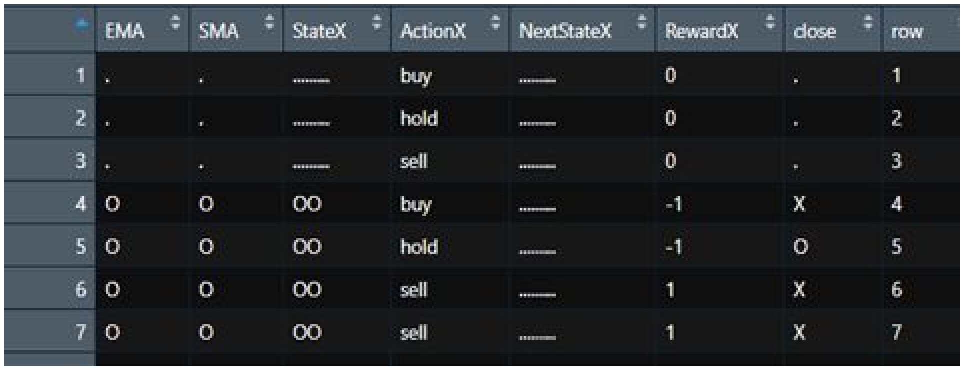


	EMA	SMA	MACD_Hist	RSI	ADX	CCI	SlowK	close	SlowD	StateX	ActionX	NextStateX	RewardX
33	X	X	O	.	.	X	O	O	.	XXO..XOO	buy in	....	1
34	X	X	O	.	.	O	O	O	O	XXO..OOO	buy in	....	1
35	X	X	O	.	.	X	O	X	O	XXO..XOX	hold	....	-1
36	X	X	O	.	.	X	.	O	.	XXO..X.O	buy in	....	1
37	X	X	O	.	.	X	O	O	.	XXO..XOO	buy in	....	1
38	X	X	O	.	.	X	O	O	O	XXO..XOO	buy in	....	1
39	X	X	O	.	.	X	O	X	O	XXO..XOX	sell	....	1
40	X	X	O	.	.	X	.	X	.	XXO..X.X	hold	....	-1
41	X	X	O	.	.	X	.	O	.	XXO..X.O	buy in	....	1
42	X	X	O	.	.	X	.	X	.	XXO..X.X	sell	....	1
43	X	X	O	.	.	X	.	O	.	XXO..X.O	buy in	....	1
44	X	X	O	.	.	X	.	X	.	XXO..X.X	sell	....	1

Model 2: With 2 Parameters

In this Model, we used only 2 parameters (vanilla version) to check if we can achieve higher accuracy in suggesting when to buy/sell/hold a stock based on Simple moving average (SMA) and exponential moving average (EMA). A column (S9) is set to calculate if that day's closing price is greater than open price, it's set to O (opportunity to buy). Likewise there are two rules set when EMA is greater than open price O is set for another column S1 and SMA is greater than open price O is set at S2. Our model will try to run RL with given two O's it will suggest either Buy/Sell/hold and if that matches the value in S9 And if it matches it will be rewarded with +1 point, if it doesn't match it penalizes -1 as shown in this table.

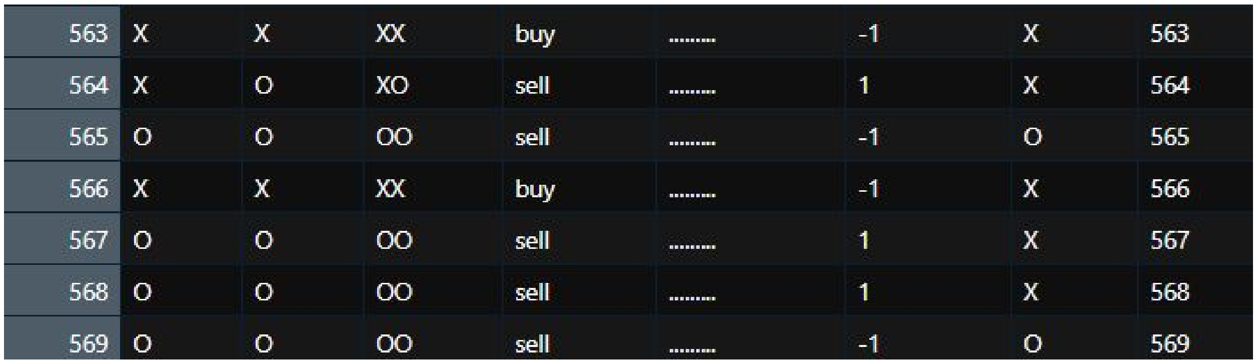
Table 1:



	EMA	SMA	StateX	ActionX	NextStateX	RewardX	close	row
1	.	.	....	buy	....	0	.	1
2	.	.	....	hold	....	0	.	2
3	.	.	....	sell	....	0	.	3
4	O	O	OO	buy	....	-1	X	4
5	O	O	OO	hold	....	-1	O	5
6	O	O	OO	sell	....	1	X	6
7	O	O	OO	sell	....	1	X	7

Table 1 shows the Buy, Sell, Hold actions suggested by RL. if that ActionX match with the close value on that day in column (Close), then RL gets a reward of +1 and on the other hand if it doesn't match he gets a penalty of -1.

Table 2:

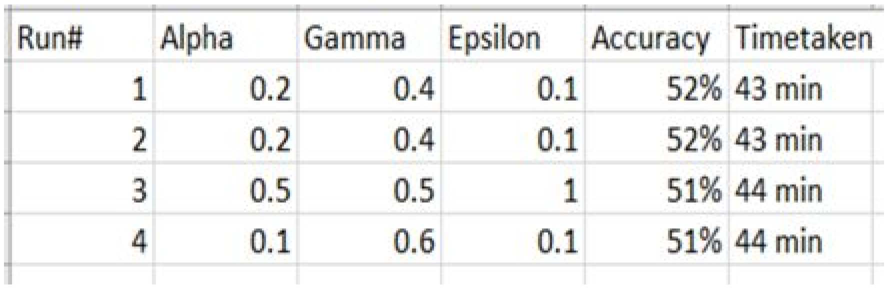


563	X	X	XX	buy	....	-1	X	563
564	X	O	XO	sell	....	1	X	564
565	O	O	OO	sell	....	-1	O	565
566	X	X	XX	buy	....	-1	X	566
567	O	O	OO	sell	....	1	X	567
568	O	O	OO	sell	....	1	X	568
569	O	O	OO	sell	....	-1	O	569

Table 2 shows another example of when RL prediction matches and he is rewarded +1 and when there is a wrong prediction, he gets a penalty of -1.

Table 3 shows, we ran 4 iterations by changing the Alpha, Gamma, Epsilon to get any accuracy improved. It remains the same at 51% to 52%, hence we need to leverage more variables to improve accuracy.

Table 3:



Run#	Alpha	Gamma	Epsilon	Accuracy	Timetaken
1	0.2	0.4	0.1	52%	43 min
2	0.2	0.4	0.1	52%	43 min
3	0.5	0.5	1	51%	44 min
4	0.1	0.6	0.1	51%	44 min

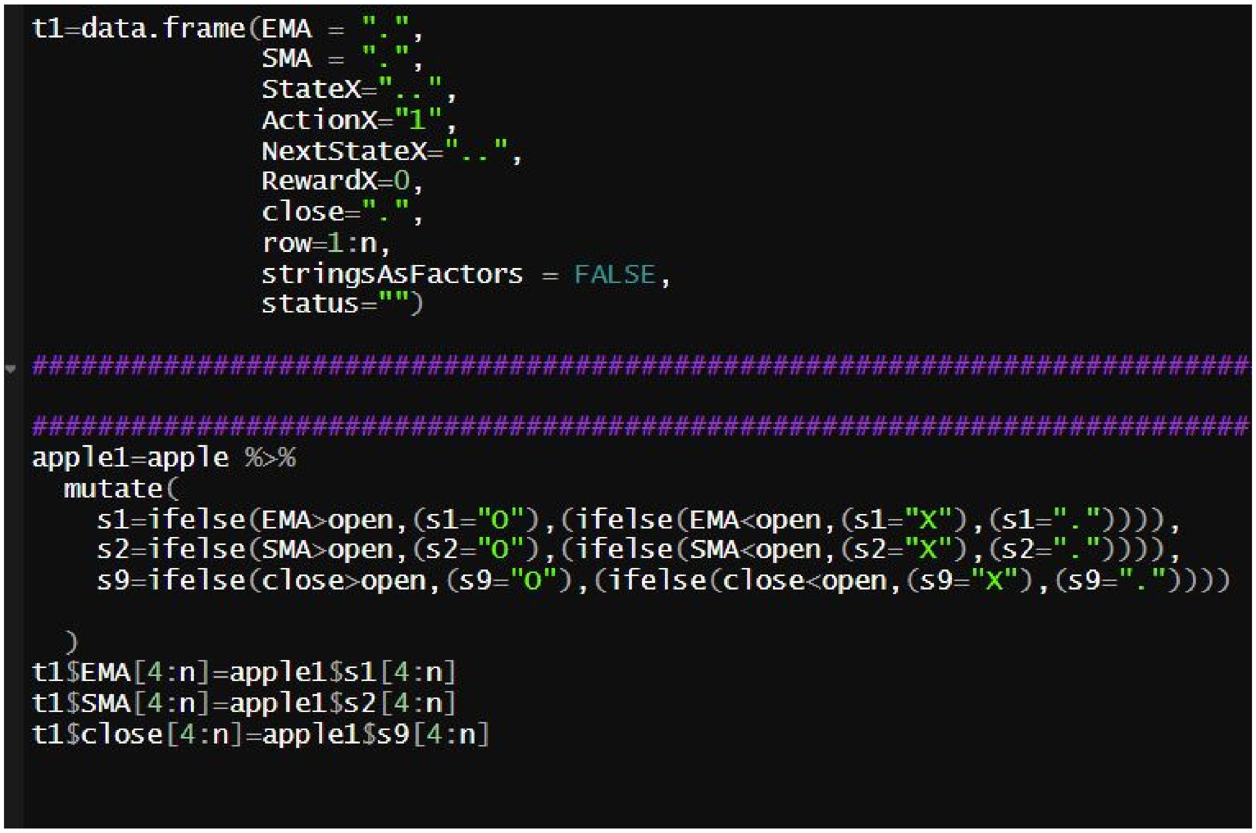
Code for Model 2:

In Image1: T1 is the data frame with EMA, SMA, StateX, Action X, RewardX, Close parameters.

Added S1,S2,S9 columns to the end of the dataframe to calculate

- If EMA > open price, then O (opportunity to buy) is updated in the S1 column
- If SMA > open price, then O (opportunity to buy) is updated in the S2 column
- If close > open price, then O (opportunity to buy) is updated in the S9 column

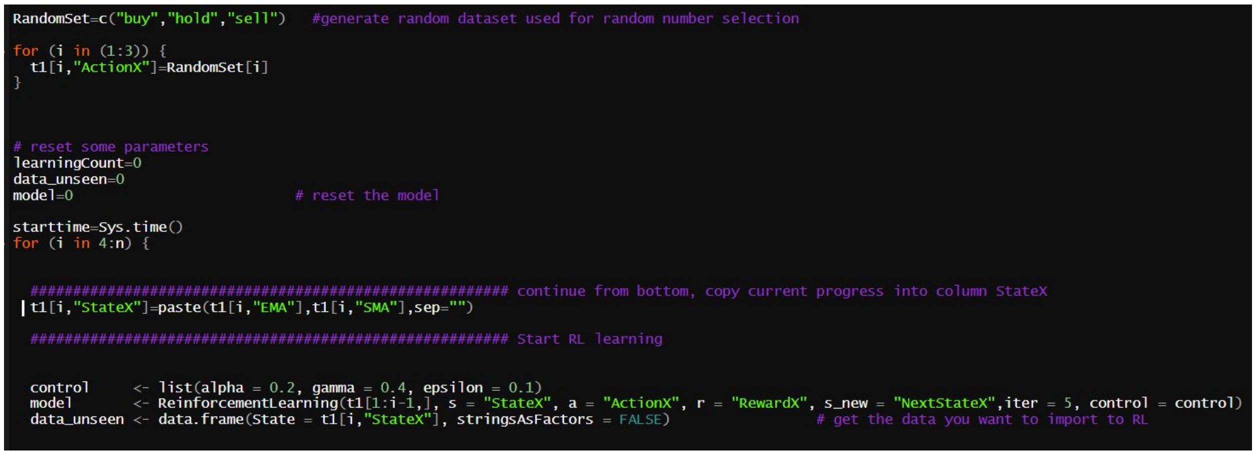
Image 1:



```
t1=data.frame(EMA = ".",
              SMA = ".",
              StateX="..",
              ActionX="1",
              NextStateX="..",
              RewardX=0,
              close=".",
              row=1:n,
              stringsAsFactors = FALSE,
              status="")

###########################################################################

###########################################################################
apple1=apple %>%
  mutate(
    s1=ifelse(EMA>open,(s1="O"),(ifelse(EMA<open,(s1="X"),(s1=".")))),
    s2=ifelse(SMA>open,(s2="O"),(ifelse(SMA<open,(s2="X"),(s2=".")))),
    s9=ifelse(close>open,(s9="O"),(ifelse(close<open,(s9="X"),(s9=".")))

  )
t1$EMA[4:n]=apple1$s1[4:n]
t1$SMA[4:n]=apple1$s2[4:n]
t1$close[4:n]=apple1$s9[4:n]
```

In Image 2, the first 3 lines is to populate Buy, Sell, Hold place holders to prevent errors.
After that, RL predicts from 4th row using Alpha, Gamma, Epsilon. And if it matches its reward +1, if prediction fails RL gets a penalty of -1.

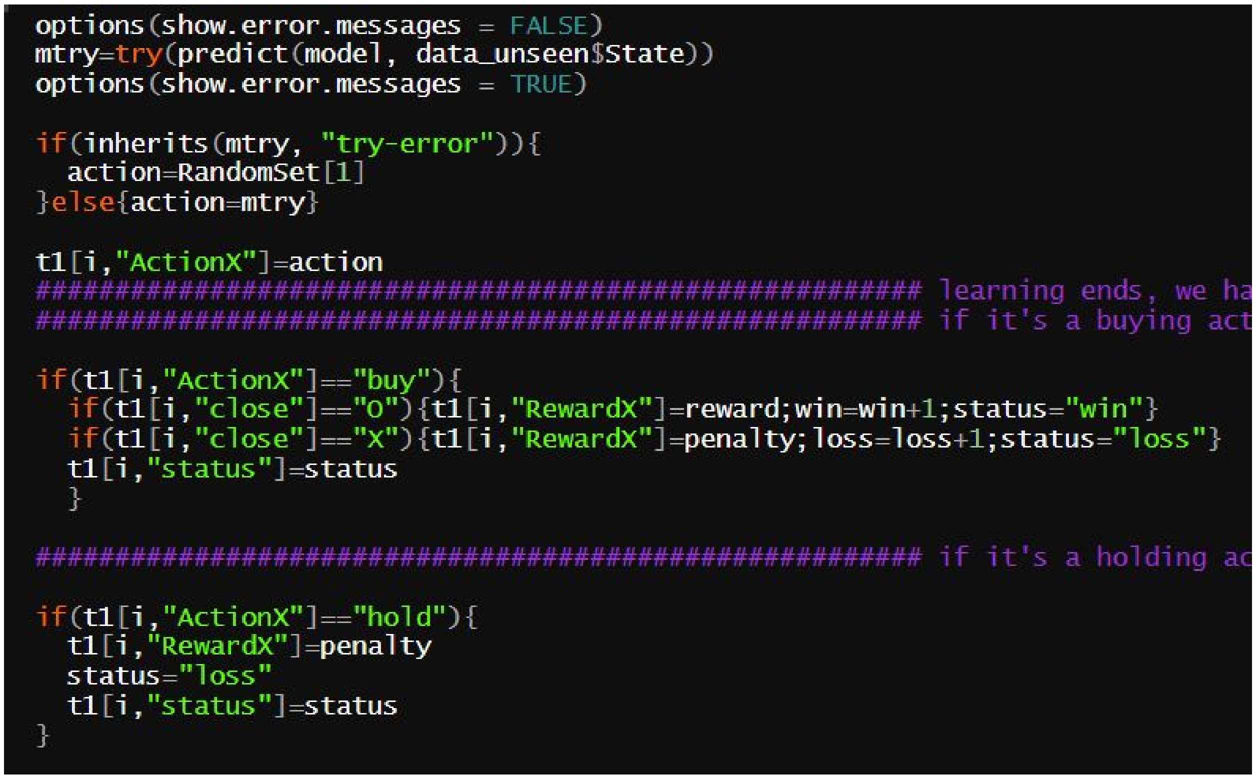
Image 2:



```r
RandomSet=c("buy","hold","sell")   #generate random dataset used for random number selection

for (i in (1:3)) {
  t1[i,"ActionX"]=RandomSet[i]
}


# reset some parameters
learningCount=0
data_unseen=0
model=0                          # reset the model

starttime=Sys.time()
for (i in 4:n) {


  ####################################################### continue from bottom, copy current progress into column StateX
  t1[i,"StateX"]=paste(t1[i,"EMA"],t1[i,"SMA"],sep="")

  ####################################################### Start RL learning


  control     <- list(alpha = 0.2, gamma = 0.4, epsilon = 0.1)
  model       <- ReinforcementLearning(t1[1:i-1,], s = "StateX", a = "ActionX", r = "RewardX", s_new = "NextStateX",iter = 5, control = control)
  data_unseen <- data.frame(State = t1[i,"StateX"], stringsAsFactors = FALSE)                 # get the data you want to import to RL
```

In Image 3, we update the column for reward , penalty as discussed in image 2. Also status column is updated as either “win” or “loss”.

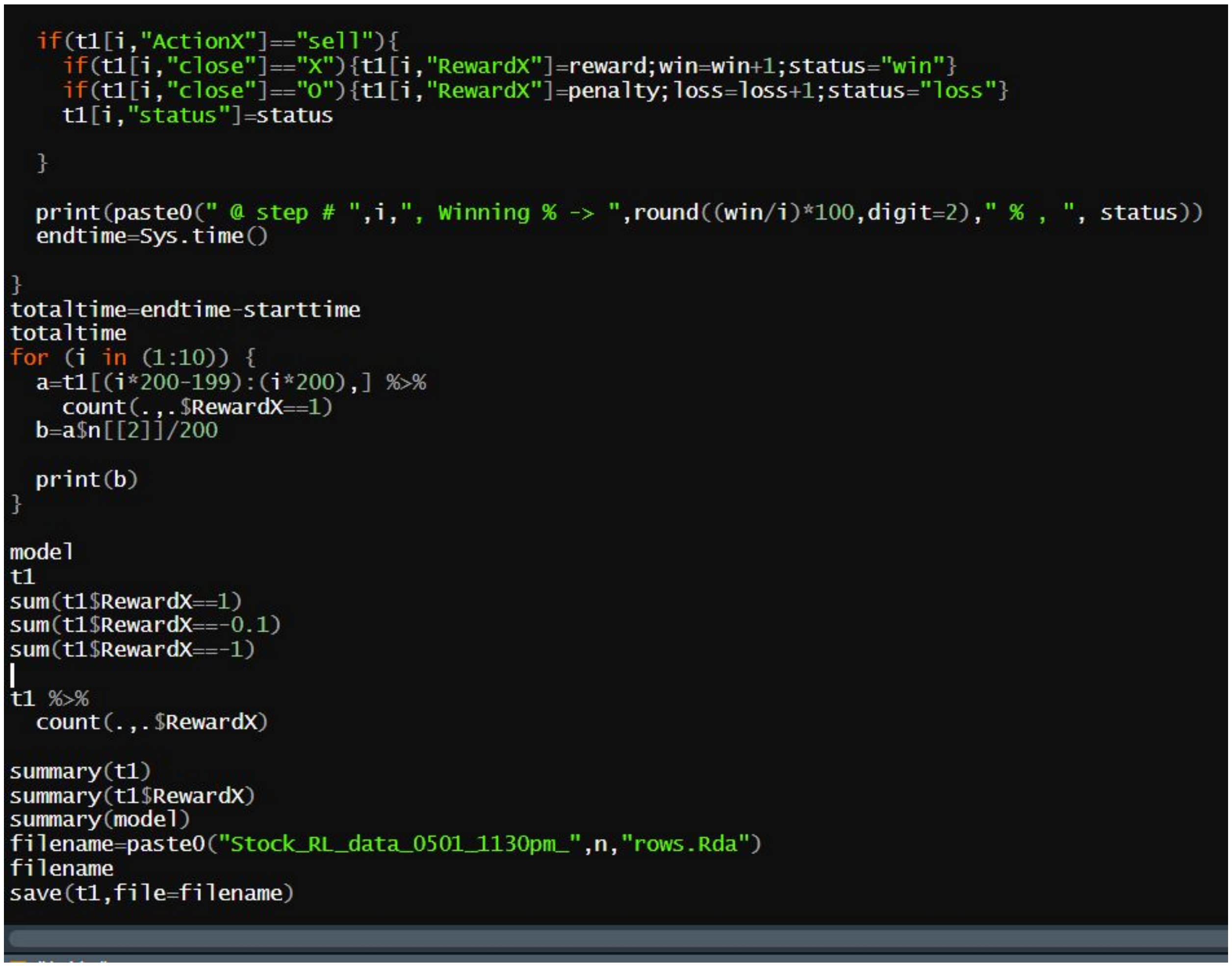
Image 3:



```r
options(show.error.messages = FALSE)
mtry=try(predict(model, data_unseen$State))
options(show.error.messages = TRUE)

if(inherits(mtry, "try-error")){
  action=RandomSet[1]
}else{action=mtry}

t1[i,"ActionX"]=action
####################################################### learning ends, we ha
####################################################### if it's a buying act

if(t1[i,"ActionX"]=="buy"){
  if(t1[i,"close"]=="O"){t1[i,"RewardX"]=reward;win=win+1;status="win"}
  if(t1[i,"close"]=="X"){t1[i,"RewardX"]=penalty;loss=loss+1;status="loss"}
  t1[i,"status"]=status
  }

####################################################### if it's a holding ac

if(t1[i,"ActionX"]=="hold"){
  t1[i,"RewardX"]=penalty
  status="loss"
  t1[i,"status"]=status
}
```

In Image 4, this model is saved to the local drive as .Rda file for any future reference.
Also summarises the winning count, time taken from start till end and winning %.

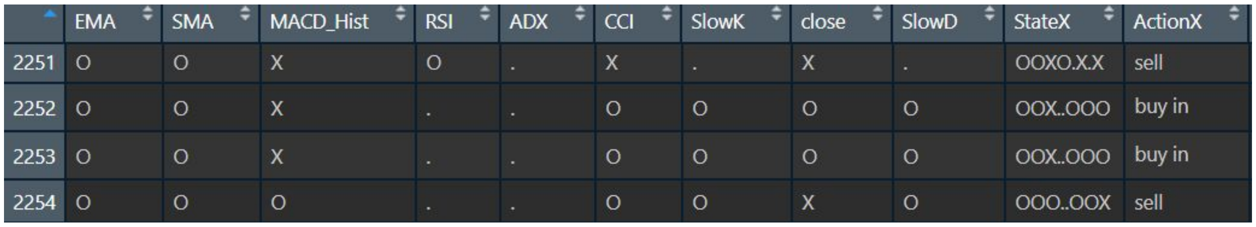
Image 4:



```
  if(t1[i,"ActionX"]=="sell"){
    if(t1[i,"close"]=="X"){t1[i,"RewardX"]=reward;win=win+1;status="win"}
    if(t1[i,"close"]=="O"){t1[i,"RewardX"]=penalty;loss=loss+1;status="loss"}
    t1[i,"status"]=status

  }

  print(paste0(" @ step # ",i,", Winning % -> ",round((win/i)*100,digit=2)," % , ", status))
  endtime=Sys.time()

}
totaltime=endtime-starttime
totaltime
for (i in (1:10)) {
  a=t1[(i*200-199):(i*200),] %>%
    count(.,.$RewardX==1)
  b=a$n[[2]]/200

  print(b)
}

model
t1
sum(t1$RewardX==1)
sum(t1$RewardX==-0.1)
sum(t1$RewardX==-1)

t1 %>%
  count(.,.$RewardX)

summary(t1)
summary(t1$RewardX)
summary(model)
filename=paste0("Stock_RL_data_0501_1130pm_",n,"rows.Rda")
filename
save(t1,file=filename)
```

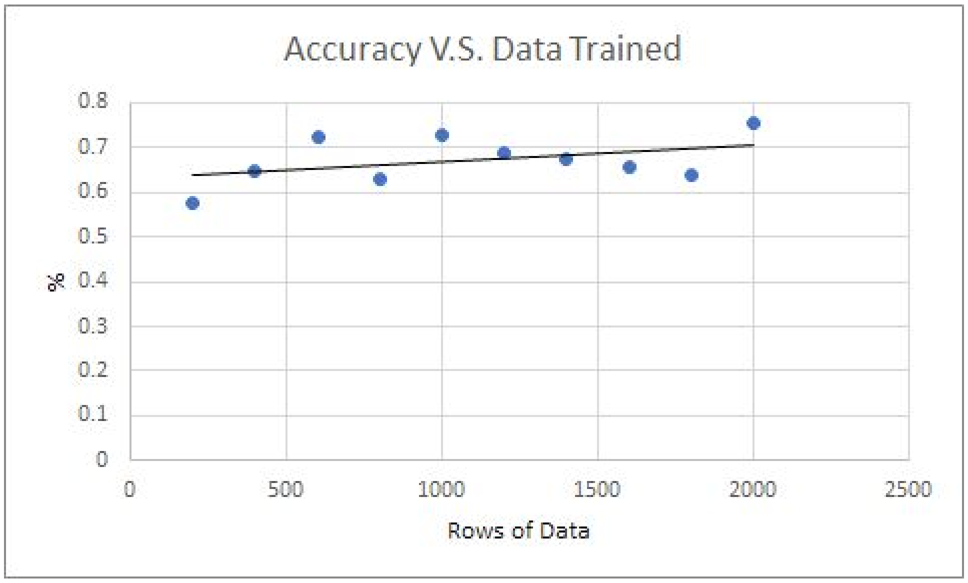
Model 3: With 9 Parameters

Next we tried to apply all the attributes to the model.

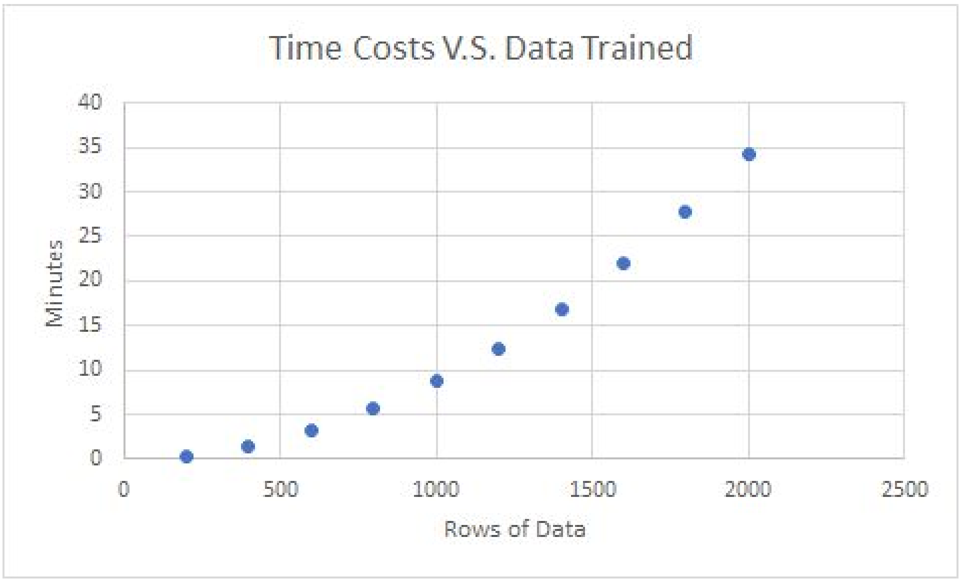


	EMA	SMA	MACD_Hist	RSI	ADX	CCI	SlowK	close	SlowD	StateX	ActionX
2251	O	O	X	O	.	X	.	X	.	OOXO.X.X	sell
2252	O	O	X	.	.	O	O	O	O	OOX..OOO	buy in
2253	O	O	X	.	.	O	O	O	O	OOX..OOO	buy in
2254	O	O	O	.	.	O	O	X	O	OOO..OOX	sell

Then we observed an increase in accuracy from 55% to 72% percent:

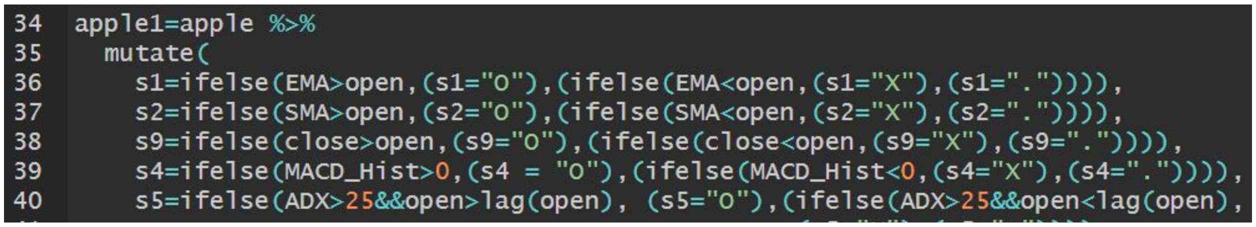


Although time costs increase exponentially, it takes only 35 mins to finish the model:



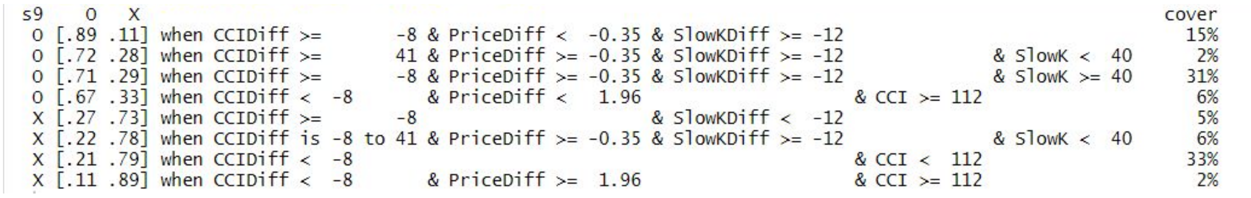
Validation

The conditions used in RL are similar to the conditions in tree models:



```
apple1=apple %>%
  mutate(
    s1=ifelse(EMA>open,(s1="O"),(ifelse(EMA<open,(s1="X"),(s1=".")))),
    s2=ifelse(SMA>open,(s2="O"),(ifelse(SMA<open,(s2="X"),(s2=".")))),
    s9=ifelse(close>open,(s9="O"),(ifelse(close<open,(s9="X"),(s9=".")))),
    s4=ifelse(MACD_Hist>0,(s4 = "O"),(ifelse(MACD_Hist<0,(s4="X"),(s4=".")))),
    s5=ifelse(ADX>25&&open>lag(open), (s5="O"),(ifelse(ADX>25&&open<lag(open),
```

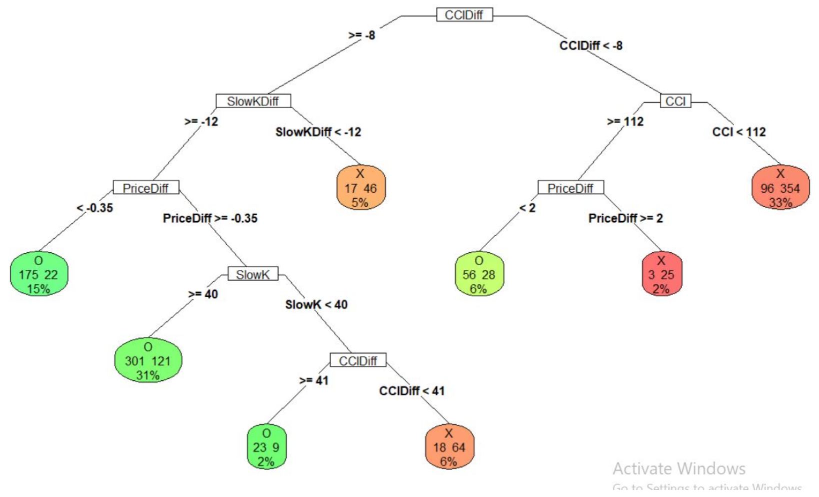
Conditions for RL



```
 s9     O   X                                                                                  cover
  O [.89 .11] when CCIDiff >=       -8 & PriceDiff <  -0.35 & SlowKDiff >= -12                     15%
  O [.72 .28] when CCIDiff >=       41 & PriceDiff >= -0.35 & SlowKDiff >= -12           & SlowK <  40   2%
  O [.71 .29] when CCIDiff >=       -8 & PriceDiff >= -0.35 & SlowKDiff >= -12           & SlowK >= 40  31%
  O [.67 .33] when CCIDiff <  -8       & PriceDiff <   1.96                  & CCI >= 112              6%
  X [.27 .73] when CCIDiff >=       -8                      & SlowKDiff <  -12                      5%
  X [.22 .78] when CCIDiff is -8 to 41 & PriceDiff >= -0.35 & SlowKDiff >= -12           & SlowK <  40   6%
  X [.21 .79] when CCIDiff <  -8                                             & CCI <  112             33%
  X [.11 .89] when CCIDiff <  -8       & PriceDiff >=  1.96                  & CCI >= 112              2%
```

Conditions for Tree Model

To validate the results, we performed a tree model to compare the accuracy:



We achieved 76% from the Tree Model. For our RL model, it was 72%.
Although it is not great compared to the other ML models made by the group Smart Profolio, we successfully proved that RL could be used in business fields like the stock market.

Challenges

The major challenge for this part is to develop a way to apply the model, because the data we have for Stock Market and Tic-Tac-Toe are totally different. From research, there are multiple ways to apply different RL models on stock.
Although the other model might have a better accuracy, since we are more familiar with the model we used in Tic-Tac-Toe, we decide to move forward with the same model.
Conclusion

Through this project, we learned how to train a Reinforcement Learning Model, and how to apply this model in different fields.

For tic-tac-toe, we successfully trained a high intelligence AI, it rarely loses to humans when testing. For stock, RL could achieve some results, but the performance is not as good as the game field.
To further increase the accuracy for RL, either train the model with more data or replace binary attributes to rounded numbers in order to create more State combinations.

From testing RL in different fields, we figured that compared to the other machine learning models, the RL learning model is lacking performance on a fixed dataset. Accuracy is close to the Tree model.
But it can do well if we know the rules to generate a dataset and let RL learn by itself- just like the Tic-Tac-Toe model.

We believe if we know the official algorithm of stock trending. We believe we could have RL achieve 100% accuracy.

References

https://www.calcalistech.com/ctech/articles/0,7340,L-3730858,00.html
https://www.google.com/imgres?imgurl=https%3A%2F%2Fimages1.calcalist.co.il%2FPicServer3%2F2018%2F01%2F30%2F791326%2F1NL.jpg&imgrefurl=https%3A%2F%2Fwww.calcalistech.com%2Fctech%2Farticles%2F0%2C7340%2CL-3730858%2C00.html&tbnid=lOmglPXxEDHQvM&vet=12ahUKEwjM59GMnvDoAhVXCVMKHQAdA2oQMygJegUIARCbAg..i&docid=yNWIiNzm3YRAPM&w=590&h=320&q=traffic%20accident&ved=2ahUKEwjM59GMnvDoAhVXCVMKHQAdA2oQMygJegUIARCbAg
https://www.google.com/imgres?imgurl=https%3A%2F%2Fak5.picdn.net%2Fshutterstock%2Fvideos%2F24247085%2Fthumb%2F11.jpg&imgrefurl=https%3A%2F%2Fwww.shutterstock.com%2Fvideo%2Fclip-24247085-robot-fight-on-white-ring-toy-robots&tbnid=0_rmjfjYdzC7nM&vet=12ahUKEwiY2sCqoPDoAhWi8FMKHb15CjcQMygeegQIARBp..i&docid=B0YBxEqBP-py6M&w=852&h=480&q=robot%20fighht&ved=2ahUKEwiY2sCqoPDoAhWi8FMKHb15CjcQMygeegQIARBp
http://joyreactor.com/post/663997
https://www.google.com/imgres?imgurl=https%3A%2F%2Fnewtheory.com%2Fwp-content%2Fuploads%2F2018%2F07%2F173-04062014032141.jpg&imgrefurl=https%3A%2F%2Fnewtheory.com%2Fthe-rise-and-fall-of-blackjack%2F&tbnid=FQZKrmKkbcBRzM&vet=12ahUKEwj_y4Gm4vDoAhVNHVMKHa-bDxIQMygNegUIARCyAg..i&docid=fmtZ5qsr6gxnqM&w=1870&h=1870&q=black%20jack&ved=2ahUKEwj_y4Gm4vDoAhVNHVMKHa-bDxIQMygNegUIARCyAg
https://pathmind.com/wiki/deep-reinforcement-learning
https://cran.r-project.org/web/packages/ReinforcementLearning/vignettes/ReinforcementLearning.html
https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56
https://www.r-project.org/logo/
https://marutitech.com/businesses-reinforcement-learning/
https://istem.ai/zh/courses/reinforcement-learning/
https://www.atlassian.com/blog/technology/artificial-intelligence-myths-john-maeda
https://towardsdatascience.com/tic-tac-toe-creating-unbeatable-ai-with-minimax-algorithm-8af9e52c1e7d
https://towardsdatascience.com/reinforcement-learning-implement-tictactoe-189582bea542
(https://www.quora.com/What-is-the-number-of-possible-ways-to-win-a-3*3-game-of-tic-tac-toe)