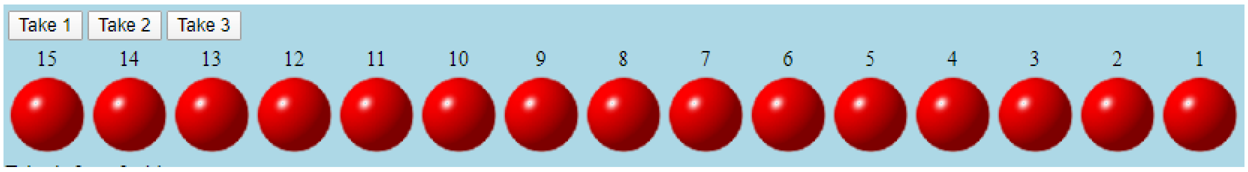
Decision Management Community Challenge Jan 2020 NIM Rules: A Rule Based and a 'Roll-Your-Own' Machine Learning Solution

(Bob Moore, JETset Business Consulting, 20 May 2020)

1 Problem Statement (from the web site)

Variants of Nim Game have been played since ancient times. This challenge uses the following version of the game:

There is a number of red balls in the row below (it could be 15, 16, or 17 balls). Two players take turns removing balls from the row, but only 1, 2 or 3 balls at a time. The player who removes the last ball loses.



You may play this game with your kids using pencils or matches. Let's assume you play against a computer. You need to make decisions who starts the game and how many balls to take after the computer takes its balls. Try to define the rules that provide a winning strategy.

2 Not very challenging?

In most problems of this kind there is a 'trick'. It probably helps to be good at number theory and it may take a bit of time to spot it, but when you do the answer is often rather simple. In this case it is defined by a single rule:

> *When it is your turn, take the number of balls such that the number which remain is one more than a multiple of four.*

So, for example if there are 12 balls left, take 3, leaving 9 (which is 4 x 2 + 1). In the event you can't do this – which for example would be if it is your turn and there are 13 balls left, then if your opponent knows this rule you will lose however many balls you take, so it doesn't really matter what you do.

More explicitly, if it is your go and there are N balls left, find the remainder when you divide N-1 by 4. This is the number of balls you should take. If the remainder is zero you are in trouble because your opponent can force a win whatever number of balls you take.

A couple of solutions have been provided to the challenge, by Jacob Feldman[1] and Alex Fleischer[2]. Jacob's hints there is a simple rule, but uses a decision table approach, Alex uses a constraint engine, which I found interesting, but both are bit 'sledge hammer to crack a nut' compared to the simple rule above.[3]

However, it struck me there was another way of addressing the challenge, avoiding the analytic approach entirely. One which Jacob alluded too at the end of his solution. Namely could one write a program which could 'learn' the best strategy?

3 A deeper analysis of the problem

Before we start, it is worth thinking a little bit more about the game mechanics and in particular why the above rule works.

There are several ways of figuring out the rule, but the way I did it was by working backwards from what happens at the end of the game.

- Firstly, one can see that if, after your last move, your opponent is left with one ball, they lose. So 'one ball left' is a losing scenario.
- Next you observe that if you have two, three or four balls you can engineer this situation because you can take one, two or three balls respectively. But you can't do it if there are five balls left.
- Worse still, if you have five balls left, after your move, your opponent will be left with either two, three or four balls, so they only have to make the appropriate move to leave you with only one ball left. So 'five balls left' is also a losing scenario, and if you can engineer matters such that there are five balls left when it is your opponent's turn you will win.
- Repeating this argument, you find if you can leave your opponent with nine balls left, thirteen balls left, or seventeen balls left you can always win if you play accurately.

Somewhere along the line the penny drops that the 'losing scenarios' are four balls apart, and that the significance of it being four rather than some other number, is that four is one more than the maximum number of balls you can take at any point. You can then come up with a couple of quick generalisations:

- If instead of saying each player can take between 1 and 3 balls each time, you say they can take between 1 and M balls, where M is arbitrary (but fixed for the duration of the game) the rule becomes:
  - When there are N balls left, take (N-1) modulo (M+1) balls (unless this is number is zero when you are in a losing position)

- If we change the rules so the person who takes the last ball is the winner rather than the losing the rule is:
  - When there are N balls left take N modulo (M+1) balls (unless this is number is zero when you are in a losing position)

[1] https://openrules.wordpress.com/2020/02/13/nim-game/
[2] https://dmcommunity.files.wordpress.com/2020/02/challenge2020jan.alexfleischer.pdf
[3] Both Alex and Jacob seem to suggest if the starting number of balls leads to a forced loss you should ask your opponent to go first, suggesting there is 'another' rule to decide whether to start or go second. But it's just a different way of looking at the situation where the first player doesn't have a forced win. If I were playing a game like this and my opponent 'suggested' I go first, my inclination would be to say 'no'.

There are some other observations which come in useful when considering machine learning.

Firstly, the game is memoryless in the sense that when it is your turn, none of the previous moves matter. If it is your go and there are say 8 balls left, it makes no difference if you started with 15 balls, 17 balls or indeed 1000 balls. All that matters is you have 8 left.

Secondly, if you record a game, by listing how many balls were taken at each move, you can determine from this record, not only how many balls there were to start with (by adding up the number of balls taken at each move), but also who won (by counting the number of moves: if it is even the first player won; otherwise it the second won).

Finally, if you have a record of a game, by ignoring the first entry in the list, you essentially have the record of a different, shorter game, with a different starting number of balls and the opposite result. For example, if you have the game:

Player 1 takes 3, Player 2 takes 2, Player 1 takes 3, Player 2 takes 1

We see we have a game starting with 9 balls and Player 1 making the first move and finally winning after 4 moves. Knocking off the first move, we get a shorter game involving 6 balls and Player 2 starting, finally losing after 3 moves:

Player 2 takes 2, Player 1 takes 3, Player 2 takes 1

So, every game of more than one move, essentially contains a number of 'sub games' one for each move of the top level game.

4 How to go about it?

While I believed I should be able to build an algorithm which learns to play this version of NIM expertly, it wasn't at all obvious how to do it. I've read a moderate amount about machine learning, but never actual tried it for real. And the immediate problem is that most of the examples rely on a kind of single shot type of input. So, for example if you want to build a system to recognise cats you can show it a lot of images of cats. But during the process you have a fixed set of inputs you can provide.

In trying to learn a two-player game, things are a bit different. You must pick a move. Your opponent responds and then you must pick another. So instead of the system having to come up with a 'single' output, it has to come up with a stream of outputs each depending partly on what it did last time and partly on how its opponent responded, You want the a system to figure out best move to make at each turn. But how will it 'know' if it was 'good' until the end of the game, because it does not know what moves its opponent will make[4]?

[4] When looking at machine learning for playing games like chess, there are means of figuring out if a position is likely to be 'good' or 'bad' after each move, so you can (in principle) do learning by evaluating the current state, rather only the final state. In the NIM game only the given rule does this, and it would make no sense to build this into the system since the objective is for the system to work it out for itself!

I did a few internet searches to see if I could see any useful examples of how one might handle the challenge without finding anything useful, but then decided that the overall problem is so simple that I really should be able to build a solution from scratch.

Taking inspiration from the tale of AlphaGo learning how to play Go by playing itself, my basic idea was to build a program which could play NIM, initially making random moves, and then getting it to play itself and to start favouring moves which in the past had led to it winning.

It turned out to be a bit more time consuming than I expected, because I made quite a quite a few mistakes on the way, but in the end the solution turned out to be fairly simple.

5 <u>Implementing the Machine Learning</u>

To get started, if I were to build a solution which learnt by playing against itself, I would need some framework to play the game. I decided to code everything up in Python. Python seems to be the programming language of the moment, and after a few misgivings about it, I have grown rather fond of it, though I'm still a bit of a tyro.

5.1 Playing a game of NIM

So, the first step was to build a bit of logic to play the game. To 'play' you need to know how many balls there are to start with, the maximum you can take at any time, and 'who' the players are. A 'player' is simply something with a name (so the output is more readable), and a *strategy*, which is no more than a way of deciding what move to make when it is their turn. So, the core of the game player was developed as a Python function **playGame** which takes these three things as parameters. The logic simply is a *while* loop, which keeps 'asking' the alternating players how many balls they want to take until one of them finally loses.

```
def playGame(ballCount, maxTake, players):
    playerIdx = 0
    player = players[playerIdx]
    moves = []
    while ballCount > 0:
        take = player['strategy'](ballCount, min(maxTake, ballCount))
        moves.append(take)
        ballCount -= take
        playerIdx = (playerIdx + 1)%2
        player = players[playerIdx]

    # On exiting the loop, the variable player will refer to the one who won
    print("Player %s Wins! (Game was %s"%(player['name'], moves))
    return(moves)
```

Some sample outputs are shown in section 10 with human against human, human against computer and computer against computer.

While it is nice to be able to see how each game is played out, the key thing about **playGame** is it records and returns the moves made during the game. It is this information which gets fed into the Machine Learning logic.

Before getting around to doing the Machine Learning I needed to make sure everything worked so I implemented some basic strategies. Each strategy depends on how many balls are available, and the maximum number which can be taken (which cannot exceed the number available of course):

- **takeBallsHuman** – basically this is an interactive option, the system prompts a (human) player to type in how many balls they want to take on the keyboard

- **takeBallsRandom** – in this strategy the system randomly choses how many balls to take assigning equal probability to all the options

- **takeBallsBest** – in this strategy the system chooses according to the rule given in section 2 if there is a forced win for the current player, and randomly chooses how many balls to take otherwise. It is worth a quick look at the logic as it is a concrete implementation of the 'rule':

```
def takeBallsBest(ballCount, maxTake):
    take = (ballCount-1)%(maxTake + 1)
    return take if take > 0 else random.randint(1,maxTake)
```

Later I added a fourth strategy **takeBallsLearn** which uses the output of the Machine Learning algorithm to choose how many balls to take.

5.2 Outline of the Learning Logic

The basic idea of the learning logic was as follows:

1. Play a game between two computer 'players' (both using the **takeBallsLearn** strategy)
2. For each game and each 'sub game' it contains, see how many balls were taken in the first move and record if the result was a win or a loss
3. Adjust the probability of taking a given number of balls with a given number remaining. The probability of taking the same number of balls with the same remaining number should increase if the current game was won and decrease if it was lost.
4. Go back to 1 and repeat until you think the model is 'good' enough.

The learning is orchestrated by a simple function **learnGame** which looks like this:

```
def learnGame(ballCount, maxTake, count):
    players = []
    players.append({'name': 'Player 1', 'strategy': takeBallsLearn})
    players.append({'name': 'Player 2', 'strategy': takeBallsLearn})
    initModel(maxTake)
    for _ in range(count):
        moves = playGame(ballCount, maxTake, players)
        updateModel(moves)
    printModel(ballCount)
```

It initialises the model using the **initModel** function, plays the game using the **playGame** function a number of times (determined by the **count** parameter) calling the **updateModel** function after each game to 'learn' from the outcome, and finally it calls the **printModel** function to report on the outcome of the learning process. To understand these three '**_Model**' functions, we need to first discuss the 'model'.

5.3 The Model

The 'model' is a data structure that holds the accumulated data and the strategy which we think is winning. In an ideal world it would be a Python class, but I could not be bothered so it's just a dictionary (or a HashMap in Java speak) with five fields:

- **maxCount** – the maximum number of balls to start a game with that we can handle with this model.

- **maxTake** – the largest number of balls one can take in any go

- **previousResults** – a 3-D array giving wins and losses when making a specific move from a specific starting point. So, the number of losses experienced when taking 2 balls from 10 remaining balls is held in:

  learntModel["previousResults"][10][2][0]

  and the number of wins in this situation is held in:

  learntModel["previousResults"][10][2][1][5]

- **distributions** - a 2-D array giving the (learned) probability of winning when making a particular move from a particular starting point, so the probability of the **takeBallsLearn** strategy 'deciding' to take 2 balls from 10 remaining balls is held in:

  learntModel["distributions"][10][2][6]

- **cumulative**- a 2-D array giving the cumulative distribution based on the number of balls taken so the probability of the **takeBallsLearn** strategy 'deciding' to take either 1 or 2 balls from 10 remaining balls is held in:

  learntModel["cumulative"][10][2][7]

Of these, only **maxTake** and **previousResults** are strictly required. A **maxCount** is provided only so the model does not have to dynamically resize itself. The values in **distributions** field are derived from the **previousResults** field, so in principle could be computed on the fly, but this would be decidedly inefficient. I've not actually checked if it makes any difference, but the idea of building the **cumulative** field from the **distributions** field was to further enhance performance.

The **initModel** function mentioned above simply builds an initial version of the model with default values for the wins and losses for each of the permitted moves with each of the possible ball counts up to **maxCount** (ie zero for both), and flat distributions for the probabilities of them being selected (ie for a take 3 game, usually 1/3 except for a starting count of 2 when it is 1/2 and of 1 when it is 1) in **distributions** and **cumulative**.

The **updateModel** function is more interesting. What it does is update the model based on a list of moves. We make use of the idea described in section 4, that a game

[5] Well actually learntModel["previousResults"][9][1][0] and learntModel["previousResults"][9][1][1], since the arrays are indexed from zero, but it's easier to understand if we use the actual number of balls 😊
[6] See previous note
[7] Ditto

contains ‘sub-games’. Assuming we can take up to three balls, and we have nine to start, **playGame** might return the list [3,3,2,1]. This means the first player took 3 balls, the second player took 3 balls (a mistake by the way, they should take only 1), next the first player took 2 balls and finally the second player took the last ball losing. We have three ‘sub-games here’ [3,2,1], [2,1] and [1] (the last being pretty boring). What this means is we have a ‘win’ for taking 3 balls when there are 9 balls left and we have a ‘win’ for taking 2 balls when there are 3 balls left. We also have a ‘loss’ for taking 3 balls when there are 6 balls left and a ‘loss’ for taking 1 ball when there is 1 ball left (no surprise there!). So, to update the model we increment the following four entries in the model[8].

learntModel["previousResults"][9][3][1]	- win taking 3 balls from 9
learntModel["previousResults"][6][3][0]	- loss taking 3 balls from 6
learntModel["previousResults"][3][2][1]	- win taking 2 balls from 3
learntModel["previousResults"][1][1][0]	- loss taking 1 ball from 1

As is obvious looking at the literature, building a learning model is a learning process in itself. On my first attempts, instead of recording wins and losses separately, I just recorded the difference. This did not work very well, possibly because the model then treated a situation where it considered 10 wins and 0 losses as no better than 20 wins and 10 losses, where intuitively a move which ‘always’ seems to win should be better than one which only wins two thirds of the time.

Once the **previousResults** are updated, then the probabilities of making particular choices can be updated to favour successful choices over unsuccessful ones, a task delegated to the **rebuildDistribution** function. Thus, the logic of **updateModel** looks like this:

```
def updateModel(moves):
    previousResults = learntModel["previousResults"]
    ballCount = sum(moves)
    winOffset = len(moves) - 1
    for idx, move in enumerate(moves):
        scoresForCount = previousResults[ballCount - 1]
        scoresForCount[move - 1][(winOffset - idx) % 2] += 1
        rebuildDistribution(ballCount)
        ballCount -= move
```

The final function of interest in **learnGame** is **printModel**. This pretty prints the model after training and takes advantage of the fact that we ‘know’ the right answer (as given in section 2), so we can see how well the training is working. We’ll see some sample outputs in section 6

5.4 The Hard Part – Updating the Probabilities

Most of what has gone before was straightforward for me to come up with. The next bit was a struggle. Having built a history of success and failure from past games, what is the best way to update the model, so that when the **takeBallsLearn** strategy uses it, it was more likely to win than it was before?

[8] Yet again the indexes have been ‘adjusted’ to make things a bit more readable

I spent hours trying out one idea after another, including using maths functions like logs and exponentials and trying peeking ahead to see how successful the next move was. Most of the time I got reasonable results for games with less than about 10 balls, but nothing seemed to work much beyond this. I was in truth despairing a bit. Then I woke up one morning with an idea for a new, but ludicrously simple approach. One so obvious it should have been the first thing I thought of.

Before we see that, let us take a quick look at how **rebuildDistribution** works.

```
def rebuildDistribution(ballCount):
    maxTake = min(learntModel["maxTake"], ballCount)
    distribution = learntModel["distributions"][ballCount - 1]
    cumulative = learntModel["cumulative"][ballCount - 1]
    previousResults = learntModel["previousResults"][ballCount - 1]
    divisor = 0

    # get the relative (unnormalised probabilities)
    for take in range(maxTake):
        pValue = winProb(previousResults[take])
        distribution[take] = pValue
        divisor += pValue

    # normalise and build cumulative probabilities
    for take in range(maxTake):
        distribution[take] /= divisor
        cumulative[take] = distribution[take] +
            (0 if take == 0 else cumulative[take - 1])
```

It looks complicated but basically, all it does is figure out which set of probabilities need updating, and calls **winProb** with the wins and losses to work out a relative probability[9] of a win for each possible move from the current position (the first for loop). It then normalises the probabilities and creates a cumulative distribution (the second for loop).

It was getting the right version of **winProb** which took all the time, which is very embarrassing given what it finally looked like. Eventually I thought why not simply set the probability of winning to the ratio of wins to losses? And it worked, it worked very nicely! The only minor consideration is that the probability of a given choice should never be exactly zero (in which case it will never be chosen again) nor should one try to divide by zero, so here is the very simple version of **winProb** which finally worked:

```
def winProb(previousResults):
    wins = max(previousResults[1], 1)
    losses = max(previousResults[0],1)
    return wins/losses
```

5.5 And what do you do with the model when it has learnt to play?

The whole point of going through the Machine Learning process is to get an agent with a strategy which works well. You don't want to have to go through the whole process each time you play the game. Fortunately, Python provides **pickle** which is a very simple way of writing out data structures to disk in binary format. So, after learning, all you need to do it save the model to disk and then later when you want to use it read it back again. Loading and saving functions are only a couple of lines long.

[9] Relative in the sense that if **winProb** returns say 5.5 for taking 1 ball and 0.5 for taking 2 balls, the normalised probability of taking 1 ball should be 11 times (ie 5.5/0.5) that of taking 2 balls

6 Results

So, did all this work? Let us look at some results!

Every time you run a learning session you will get different results, since there is a level of randomness in the process[10]. However below is a typical output from running **learnGame**, starting with 17 balls allowing up to 3 balls to be taken, and running 50 iterations.

```
learning with 50 iterations
distinct games played is: 49
Max number of balls to take is 3
---  1 balls, best 0 probs: 1 with p=1.000 (w=0,l=47),
OK!  2 balls, best 1 probs: 1 with p=0.933 (w=14,l=0), 2 with p=0.067 (w=0,l=1),
OK!  3 balls, best 2 probs: 2 with p=0.953 (w=17,l=0), 3 with p=0.028 (w=0,l=2), 1 with p=0.019 (w=1,l=3),
OK!  4 balls, best 3 probs: 3 with p=0.923 (w=16,l=0), 1 with p=0.058 (w=1,l=1), 2 with p=0.019 (w=0,l=3),
---  5 balls, best 0 probs: 2 with p=0.459 (w=3,l=15), 3 with p=0.287 (w=0,l=8), 1 with p=0.255 (w=1,l=9),
OK!  6 balls, best 1 probs: 1 with p=0.842 (w=12,l=3), 3 with p=0.105 (w=1,l=2), 2 with p=0.053 (w=1,l=4),
OK!  7 balls, best 2 probs: 2 with p=0.915 (w=9,l=0), 3 with p=0.051 (w=2,l=4), 1 with p=0.034 (w=1,l=3),
OK!  8 balls, best 3 probs: 3 with p=0.876 (w=11,l=1), 1 with p=0.064 (w=4,l=5), 2 with p=0.060 (w=3,l=4),
---  9 balls, best 0 probs: 1 with p=0.370 (w=5,l=7), 3 with p=0.370 (w=5,l=7), 2 with p=0.259 (w=2,l=4),
OK! 10 balls, best 1 probs: 1 with p=0.632 (w=6,l=3), 2 with p=0.263 (w=5,l=6), 3 with p=0.105 (w=1,l=3),
NO! 11 balls, best 2 probs: 1 with p=0.504 (w=9,l=5), 2 with p=0.440 (w=11,l=7), 3 with p=0.056 (w=0,l=5),
NO! 12 balls, best 3 probs: 1 with p=0.532 (w=10,l=8), 2 with p=0.255 (w=3,l=5), 3 with p=0.213 (w=1,l=2),
--- 13 balls, best 0 probs: 1 with p=0.609 (w=7,l=3), 2 with p=0.261 (w=0,l=1), 3 with p=0.130 (w=0,l=2),
NO! 14 balls, best 1 probs: 2 with p=0.422 (w=7,l=9), 3 with p=0.345 (w=7,l=11), 1 with p=0.233 (w=3,l=7),
OK! 15 balls, best 2 probs: 2 with p=0.522 (w=2,l=0), 1 with p=0.348 (w=4,l=3), 3 with p=0.130 (w=1,l=2),
NO! 16 balls, best 3 probs: 2 with p=0.778 (w=7,l=1), 1 with p=0.111 (w=1,l=1), 3 with p=0.111 (w=1,l=0),
--- 17 balls, best 0 probs: 3 with p=0.581 (w=16,l=13), 2 with p=0.315 (w=4,l=6), 1 with p=0.105 (w=2,l=9),
```

Basically, this shows the probability of taking a given number of balls given the number of balls left at each stage, ordered by left to right by probability. So for example we see from the yellow highlighted line that if there were 10 balls to start, the **takeBallsLearn** strategy would 'decide' to take 1 ball about 63% of the time, 2 balls about 26% of the time and 3 balls about 11% of the time. For comparison, the 'best' strategy is to take 1 ball. Note there is no 'best' move when in a losing scenario (e.g. the cyan highlighted line

[10] Though you can make things repeatable by explicitly seeding the random number generator of course

showing the probability distribution for 13 balls). For these cases the probability of any move tends to be below 0.5, although when only a few training iterations have been run it is can be rather higher as is the case here.

Even after only 50 training trials, the strategy chosen is correct more than 90% of the time when you get down below 5 balls and 80% of the time below 10 balls. More training improves matters of course.

Upping the number of iterations to 500, the **takeBallsLearn** strategy is performing respectably:

```
learning with 500 iterations
distinct games played is: 239
Max number of balls to take is 3
---  1 balls, best 0 probs: 1 with p=1.000 (w=0,l=490),
OK!  2 balls, best 1 probs: 1 with p=0.999 (w=142,l=0), 2 with p=0.001 (w=0,l=5),
OK!  3 balls, best 2 probs: 2 with p=0.998 (w=201,l=0), 1 with p=0.001 (w=0,l=4), 3 with p=0.001 (w=0,l=5),
OK!  4 balls, best 3 probs: 3 with p=0.993 (w=147,l=0), 2 with p=0.005 (w=2,l=3), 1 with p=0.002 (w=1,l=3),
---  5 balls, best 0 probs: 2 with p=0.421 (w=7,l=191), 1 with p=0.324 (w=4,l=142), 3 with p=0.255 (w=3,l=135),
OK!  6 balls, best 1 probs: 1 with p=0.986 (w=173,l=5), 2 with p=0.010 (w=0,l=3), 3 with p=0.004 (w=1,l=7),
OK!  7 balls, best 2 probs: 2 with p=0.987 (w=133,l=3), 3 with p=0.009 (w=2,l=5), 1 with p=0.004 (w=0,l=5),
OK!  8 balls, best 3 probs: 3 with p=0.972 (w=162,l=6), 2 with p=0.021 (w=4,l=7), 1 with p=0.007 (w=1,l=5),
---  9 balls, best 0 probs: 1 with p=0.379 (w=11,l=144), 3 with p=0.337 (w=11,l=162), 2 with p=0.285 (w=7,l=122),
OK! 10 balls, best 1 probs: 1 with p=0.954 (w=118,l=5), 3 with p=0.025 (w=5,l=8), 2 with p=0.020 (w=3,l=6),
OK! 11 balls, best 2 probs: 2 with p=0.959 (w=189,l=15), 1 with p=0.023 (w=4,l=13), 3 with p=0.018 (w=4,l=17),
OK! 12 balls, best 3 probs: 3 with p=0.961 (w=121,l=9), 1 with p=0.026 (w=5,l=14), 2 with p=0.013 (w=2,l=11),
--- 13 balls, best 0 probs: 2 with p=0.419 (w=32,l=150), 1 with p=0.331 (w=14,l=83), 3 with p=0.250 (w=13,l=102),
OK! 14 balls, best 1 probs: 1 with p=0.874 (w=166,l=34), 2 with p=0.082 (w=17,l=37), 3 with p=0.043 (w=8,l=33),
OK! 15 balls, best 2 probs: 2 with p=0.913 (w=77,l=9), 1 with p=0.047 (w=4,l=9), 3 with p=0.040 (w=3,l=8),
OK! 16 balls, best 3 probs: 3 with p=0.853 (w=92,l=16), 2 with p=0.092 (w=13,l=21), 1 with p=0.056 (w=3,l=8),
--- 17 balls, best 0 probs: 3 with p=0.429 (w=87,l=161), 1 with p=0.331 (w=45,l=108), 2 with p=0.240 (w=23,l=76),
```

The ‘best’ move is chosen with at least 85% probability. Not surprisingly the accuracy with which the best move is chosen improves the fewer balls remaining. By 5000 iterations the model is more than 98% accurate in all cases. I haven’t timed it in detail, but this only takes about a second or so on my laptop.

Another interesting feature is the number of distinct games played. Evidently there are a finite number of distinct games you can play – there are only 13 possible games if you start with only 5 balls for example. At an early stage in working I wasn’t sure if playing a game again would help or hinder matters (it helps – a lot) so I set up a record of all the distinct games played. I'm not sure exactly how many distinct games are possible starting with 17 balls, but from simulations it is around 18,000. However, when you run 5000 training iterations, only about 350 or so distinct games are actually played. The learning process weeds out 'silly' games.

Last, but not least, the code was developed from the start to allow for variable numbers of both starting balls and upper limit on the number of balls which can be taken at each stage. So, we can also look at what happens when we train a model to cope with being able to remove a different number of balls to 3. As an example, here is the output for one run of 5000 iterations where the game starts with 20 balls and one can take up to 4 balls.

```
learning with 5000 iterations
distinct games played is: 842
Max number of balls to take is 4
---  1 balls, best 0 probs: 1 with p=1.000 (w=0,l=4988),
OK!  2 balls, best 1 probs: 1 with p=1.000 (w=516,l=0), 2 with p=0.000 (w=0,l=4),
OK!  3 balls, best 2 probs: 2 with p=0.999 (w=890,l=0), 1 with p=0.001 (w=2,l=4), 3 with p=0.000 (w=0,l=3),
OK!  4 balls, best 3 probs: 3 with p=1.000 (w=2081,l=0), 1 with p=0.000 (w=3,l=11), 2 with p=0.000 (w=2,l=8), 4 with p=0.000 (w=0,l=5),
OK!  5 balls, best 4 probs: 4 with p=1.000 (w=1501,l=0), 3 with p=0.000 (w=0,l=4), 2 with p=0.000 (w=0,l=5), 1 with p=0.000 (w=0,l=7),
---  6 balls, best 0 probs: 2 with p=0.396 (w=18,l=2053), 1 with p=0.306 (w=10,l=1474), 3 with p=0.208 (w=4,l=867), 4 with p=0.090 (w=0,l=500),
OK!  7 balls, best 1 probs: 1 with p=0.998 (w=1118,l=7), 2 with p=0.001 (w=1,l=7), 4 with p=0.001 (w=0,l=9), 3 with p=0.001 (w=1,l=10),
OK!  8 balls, best 2 probs: 2 with p=0.995 (w=1338,l=8), 4 with p=0.002 (w=5,l=16), 1 with p=0.001 (w=1,l=4), 3 with p=0.001 (w=2,l=10),
OK!  9 balls, best 3 probs: 3 with p=0.994 (w=1018,l=8), 4 with p=0.002 (w=3,l=10), 1 with p=0.002 (w=2,l=8), 2 with p=0.001 (w=1,l=6),
OK! 10 balls, best 4 probs: 4 with p=0.994 (w=1420,l=9), 3 with p=0.003 (w=7,l=17), 1 with p=0.002 (w=8,l=22), 2 with p=0.001 (w=1,l=6),
--- 11 balls, best 0 probs: 1 with p=0.295 (w=37,l=1370), 3 with p=0.259 (w=31,l=1307), 4 with p=0.240 (w=24,l=1093), 2 with p=0.205
(w=18,l=961),
OK! 12 balls, best 1 probs: 1 with p=0.986 (w=1139,l=25), 2 with p=0.007 (w=10,l=33), 3 with p=0.004 (w=4,l=23), 4 with p=0.003 (w=4,l=25),
OK! 13 balls, best 2 probs: 2 with p=0.983 (w=949,l=20), 1 with p=0.008 (w=15,l=37), 3 with p=0.006 (w=6,l=21), 4 with p=0.002 (w=2,l=18),
OK! 14 balls, best 3 probs: 3 with p=0.980 (w=1369,l=40), 2 with p=0.009 (w=15,l=46), 1 with p=0.009 (w=14,l=47), 4 with p=0.002 (w=1,l=12),
OK! 15 balls, best 4 probs: 4 with p=0.982 (w=1274,l=25), 3 with p=0.008 (w=20,l=46), 1 with p=0.005 (w=9,l=33), 2 with p=0.005 (w=5,l=21),
--- 16 balls, best 0 probs: 2 with p=0.282 (w=88,l=1239), 1 with p=0.266 (w=78,l=1165), 3 with p=0.236 (w=50,l=840), 4 with p=0.216
(w=56,l=1028),
OK! 17 balls, best 1 probs: 1 with p=0.891 (w=313,l=32), 3 with p=0.040 (w=32,l=72), 4 with p=0.038 (w=27,l=64), 2 with p=0.030 (w=20,l=61),
OK! 18 balls, best 2 probs: 2 with p=0.866 (w=249,l=30), 1 with p=0.074 (w=49,l=69), 4 with p=0.030 (w=16,l=55), 3 with p=0.030 (w=12,l=42),
OK! 19 balls, best 3 probs: 3 with p=0.837 (w=166,l=22), 1 with p=0.076 (w=43,l=63), 2 with p=0.046 (w=14,l=34), 4 with p=0.042 (w=15,l=40),
OK! 20 balls, best 4 probs: 4 with p=0.912 (w=3544,l=188), 1 with p=0.032 (w=159,l=238), 2 with p=0.028 (w=153,l=263), 3 with p=0.028
(w=166,l=289),
```

As one would expect, the more balls to start with and the more balls one can take at a time means there are far more possible games, so it takes longer to get a model which is highly proficient, but this is one is pretty good choosing the best move at each point with at least 83% accuracy.

7 Crystallising the outcome and other things to try

One slight issue with the learnt model is that even after training it is not perfect. The **takeBallsLearn** strategy still uses random numbers to make its choice and there is a small possibility that a non-optimal move might be made in a game. One option would be to ‘crystallise’ the distributions after training, setting the move with the highest probability to one and making the strategy full deterministic. This would be a pretty simple modification to the existing code. However, while this seems reasonable if the highest probability is close to one to start with (say greater than 0.9), but what if it is only about 0.5 which happens in the ‘losing’ scenarios?

Another observation is that at the start of training, several ‘bad’ options will be explored before the system learns to focus only on ‘good’ ones. One way to speed up training might be to arrange to ‘forget’ the initial earlier parts of the training process as it progresses. This would require some considerable changes to the code as it stands as the model records only wins and losses and not the order in which they occurred.

8 Afterthoughts

In lockdown, working from scratch was a good way to pass the time and this proved more fun and more challenging than expected. At the end, though, combining three simple ideas: having the learning model play itself; exploiting the notion of sub-games within games; and a simple way of updating the model to reward wins and punish losses was all that was needed. But it also makes it clear I have a lot more to learn about Machine Learning!

9 Source code not included?

Using some off-the-shelf libraries, you can create some pretty sophisticated Machine Learning systems with only a few lines of code. However, since I built everything from scratch for the challenge, the full source code is a bit more extensive (over 300 lines including comments). This includes some odds and ends associated with me improving my Python skills on the way the job, rather than being directly associated with solving the problem. Since there is so much code, I’ve not included it here, but if you want a copy just drop me an email.

10 Sample Outputs of Game Play

To give a flavour of what happens when a real game is played, here are some sample games made by calling the **playGame** function in a Python console

10.1 Two humans playing one another

```
Input ball count  (between 1 and 20): 16
Input maximum number of balls which can be taken in one go  (between 1 and
5): 3
Select strategy of first player (Alice): (I)teractive (default), (R)andom,
(B)est, or (L)earn,
i
Select strategy of second player (Bob): (I)teractive (default), (R)andom,
(B)est, or (L)earn,
i
There are 16 balls left. How many balls will you take?  (between 1 and 3): 3
Alice removed 3 balls using strategy takeBallsHuman and leaving 13 remaining
There are 13 balls left. How many balls will you take?  (between 1 and 3): 2
```

```
Bob removed 2 balls using strategy takeBallsHuman and leaving 11 remaining
There are 11 balls left. How many balls will you take?  (between 1 and 3): 2
Alice removed 2 balls using strategy takeBallsHuman and leaving 9 remaining
There are 9 balls left. How many balls will you take?  (between 1 and 3): 1
Bob removed 1 balls using strategy takeBallsHuman and leaving 8 remaining
There are 8 balls left. How many balls will you take?  (between 1 and 3): 3
Alice removed 3 balls using strategy takeBallsHuman and leaving 5 remaining
There are 5 balls left. How many balls will you take?  (between 1 and 3): 1
Bob removed 1 balls using strategy takeBallsHuman and leaving 4 remaining
There are 4 balls left. How many balls will you take?  (between 1 and 3): 3
Alice removed 3 balls using strategy takeBallsHuman and leaving 1 remaining
There are 1 balls left. How many balls will you take?  (between 1 and 1): 1
Bob removed 1 balls using strategy takeBallsHuman and leaving 0 remaining
Player Alice Wins! (Game was [3, 2, 2, 1, 3, 1, 3, 1]
```

10.2 A human loses to the computer using the ‘best’ strategy

```
Input ball count  (between 1 and 20): 16
Input maximum number of balls which can be taken in one go  (between 1 and
5): 3
Select strategy of first player (Alice): (I)teractive (default), (R)andom,
(B)est, or (L)earn,
i
Select strategy of second player (Bob): (I)teractive (default), (R)andom,
(B)est, or (L)earn,
b
There are 16 balls left. How many balls will you take?  (between 1 and 3): 3
Alice removed 3 balls using strategy takeBallsHuman and leaving 13 remaining
Bob removed 3 balls using strategy takeBallsBest and leaving 10 remaining
There are 10 balls left. How many balls will you take?  (between 1 and 3): 3
Alice removed 3 balls using strategy takeBallsHuman and leaving 7 remaining
Bob removed 2 balls using strategy takeBallsBest and leaving 5 remaining
There are 5 balls left. How many balls will you take?  (between 1 and 3): 2
Alice removed 2 balls using strategy takeBallsHuman and leaving 3 remaining
Bob removed 2 balls using strategy takeBallsBest and leaving 1 remaining
There are 1 balls left. How many balls will you take?  (between 1 and 1): 1
Alice removed 1 balls using strategy takeBallsHuman and leaving 0 remaining
Player Bob Wins! (Game was [3, 3, 3, 2, 2, 2, 1]
```

Note that Alice should have taken only one ball on her second turn, which would have allowed her to go on and win.

10.3 The two computer players making random actions

```
Input ball count  (between 1 and 20): 16
Input maximum number of balls which can be taken in one go  (between 1 and
5): 3
Select strategy of first player (Alice): (I)teractive (default), (R)andom,
(B)est, or (L)earn,
r
Select strategy of second player (Bob): (I)teractive (default), (R)andom,
(B)est, or (L)earn,
r
Alice removed 3 balls using strategy takeBallsRandom and leaving 13 remaining
Bob removed 1 balls using strategy takeBallsRandom and leaving 12 remaining
Alice removed 2 balls using strategy takeBallsRandom and leaving 10 remaining
Bob removed 1 balls using strategy takeBallsRandom and leaving 9 remaining
Alice removed 1 balls using strategy takeBallsRandom and leaving 8 remaining
```

```
Bob removed 1 balls using strategy takeBallsRandom and leaving 7 remaining
Alice removed 3 balls using strategy takeBallsRandom and leaving 4 remaining
Bob removed 2 balls using strategy takeBallsRandom and leaving 2 remaining
Alice removed 2 balls using strategy takeBallsRandom and leaving 0 remaining
Player Bob Wins! (Game was [3, 1, 2, 1, 1, 1, 3, 2, 2]
```

Note random play leads to silly actions like taking the all the remaining balls when one is not obliged to.