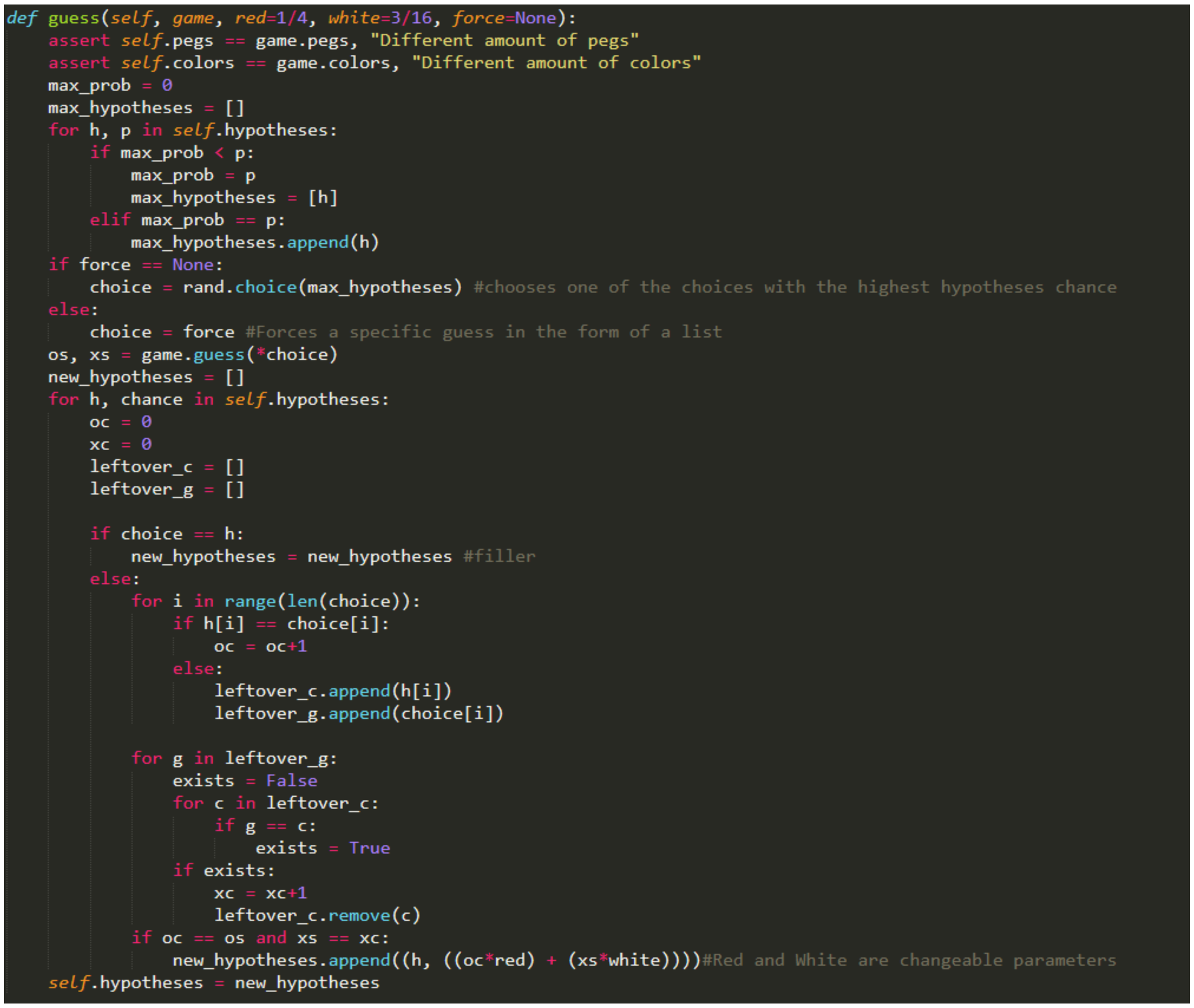
Creating a Mastermind Solver

For my project I decided to create a mastermind solver using hypotheses testing similarly to how we did things in the Number Game project to see how we could apply this type of Bayesian learning to other games. The rules of Mastermind for reference can be found here. The notebook contains some example code for how to get everything running.

To keep things consistent, the vocabulary I use for this paper will consist of "pegs", which can generally mean the guesses or the white/red pegs, "white and red pegs" which represent the correct peg in the wrong position and the correct peg in the right position, respectively (this correlates to x's and o's in my code), and finally in the results section, the code references the code in the game that needs to be cracked, not the actual code I used to create this.

Candidate Hypotheses

The main feature of my project of course is the solver itself. The solver uses hypotheses to keep track of what it should be guessing. The way it does this is by running a "guess" method repeatedly until it reaches the correct point. The code is below for reference. (The force parameter is there only if I want to force a specific guess)



```
def guess(self, game, red=1/4, white=3/16, force=None):
    assert self.pegs == game.pegs, "Different amount of pegs"
    assert self.colors == game.colors, "Different amount of colors"
    max_prob = 0
    max_hypotheses = []
    for h, p in self.hypotheses:
        if max_prob < p:
            max_prob = p
            max_hypotheses = [h]
        elif max_prob == p:
            max_hypotheses.append(h)
    if force == None:
        choice = rand.choice(max_hypotheses) #chooses one of the choices with the highest hypotheses chance
    else:
        choice = force #Forces a specific guess in the form of a list
    os, xs = game.guess(*choice)
    new_hypotheses = []
    for h, chance in self.hypotheses:
        oc = 0
        xc = 0
        leftover_c = []
        leftover_g = []

        if choice == h:
            new_hypotheses = new_hypotheses #filler
        else:
            for i in range(len(choice)):
                if h[i] == choice[i]:
                    oc = oc+1
                else:
                    leftover_c.append(h[i])
                    leftover_g.append(choice[i])

            for g in leftover_g:
                exists = False
                for c in leftover_c:
                    if g == c:
                        exists = True
                if exists:
                    xc = xc+1
                    leftover_c.remove(c)
            if oc == os and xs == xc:
                new_hypotheses.append((h, ((oc*red) + (xs*white))))#Red and White are changeable parameters
    self.hypotheses = new_hypotheses
```

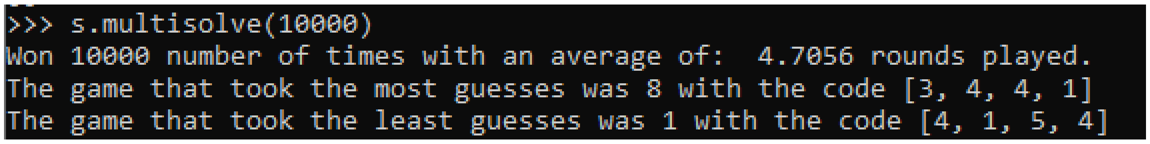
The code is slightly messy so I will break it down. All hypotheses are stored as tuples in the solver class, with the first element being the actual hypothesis, and the second being the probability that that specific hypothesis is chosen. To begin, out of all the hypotheses possible, this algorithm will select one of the hypotheses that has the highest probability of being chosen, which equates to guessing one of the most likely possibilities based on information known.

The way this solver calculates the probability is in the second last line where (oc*red) + (xs*white). "Oc" and "xs" represent the total number of correct pegs in the correct positions, and the total number of correct pegs in the incorrect positions, respectively. "Red" and "White" are merely variables used to give weighting to how much value each piece is worth. As a default, for most of the experiments ran in this paper I had red=1/4 and white=3/16. The reasoning for this was that when we have 4 reds, we have a probability of 1 that we have the correct result, so therefor each red peg should be weighed equally to add up to 1. As for the white pegs, I did not think that a white peg was as useful as a red peg for information, but I also felt that two white pegs was worth more than a single red peg, so I decided to find a happy medium with 3/16ths. I initially wrote the hypothesis probability so that the previous hypothesis would also be factored (it would be multiplied by the current hypothesis probability), however after testing, it did not seem to change the results much.

It is worth noting that I did not end up implementing a proper min-max strategy. This was mainly due to not having enough time and the process being exceedingly difficult to integrate with candidate hypothesis testing. Given more time, ideally, I could have a specific case to case hypothesis for every variation of red/white pegs with a certain combination of colors.

Testing

My algorithm seemed to work and was able to always solve within 8 turns at most, averaging around 4.7 turns total. For reference, Donald Knuth, famous algorithms researcher and the first person to attempt finding an optimal solver for Mastermind, was able to create a min-max strategy with a worst case of 5 rounds being played. In 1993, Kenji Koyama and Tony Lai were able to show solving a random code would take on average 4.34 turns (unfortunately I was unable to find the paper, but this finding was referenced on Wikipedia and various research papers about optimizing Mastermind). While I was unable to come up with an algorithm to go as low or as optimal, averaging around 4.7 in 10,000 games was better than I had hoped.



```
>>> s.multisolve(10000)
Won 10000 number of times with an average of:  4.7056 rounds played.
The game that took the most guesses was 8 with the code [3, 4, 4, 1]
The game that took the least guesses was 1 with the code [4, 1, 5, 4]
```

Result from testing 10,000 times. Can be tested again in notebook

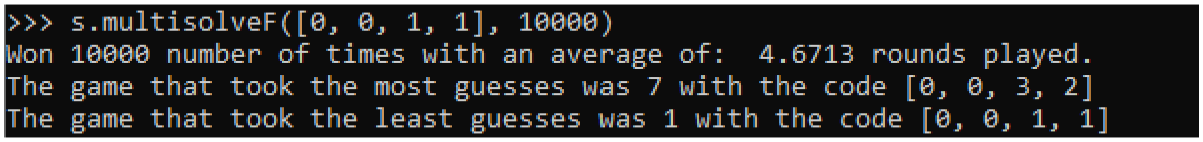
Taking a deeper look at the data

I ran each possible code (1296 possibilities) 50 times and recorded the results in the excel document labelled "Results". The average was as expected, and the standard deviation was 0.914, with the average number of rounds played for each code ranging from 3.96 to 5.12. This leads me to believe that no code seems to be the "hardest" to solve overall. There were not too many interesting patterns present to help make solid judgements. The only noticeable ones were probably having 4 of the same color tends to be solved more easily and having 3 of the same color is similarly easy, but much less obvious, so it is impossible to draw a conclusion on whether 3 colors is a bad idea.

Forcing the first guess

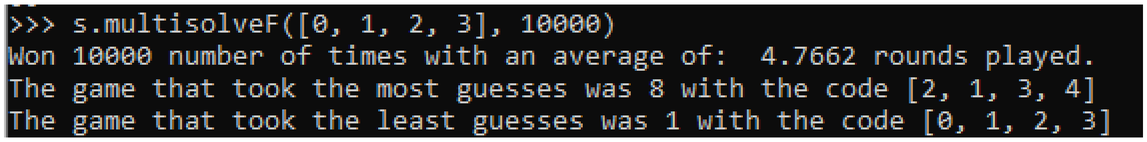
In Knuth's paper, he showed that it was most optimal to have a first guess as [0, 0, 1, 1] to achieve at most 5 rounds. I decided to test this along with other possible first guesses to see how well my model would do. As all of these hypotheses are randomly chosen and there is no distinction between a number besides representing different pegs, I assume all of the first guesses apply without loss of generality (i.e. [0, 0, 1, 1] is equivalent to [0, 0, 1, 2]). Additionally, I found from testing that 10,000 was enough to ensure I was very close to the true average.

First I tested [0, 0, 1, 1] as my first guess, and found that there was a very slight improvement. I was consistently getting around 4.67. While this was only 0.03 lower than guessing randomly first, the fact that it was consistent showed it was at least a good first idea.



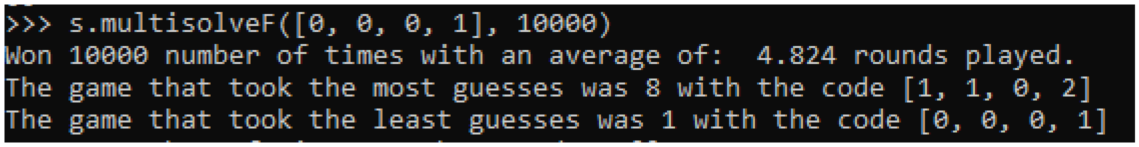
```
>>> s.multisolveF([0, 0, 1, 1], 10000)
Won 10000 number of times with an average of:  4.6713 rounds played.
The game that took the most guesses was 7 with the code [0, 0, 3, 2]
The game that took the least guesses was 1 with the code [0, 0, 1, 1]
```

Next I tested doing [0, 1, 2, 3] as my first guess, and found that it actually performed worse than guessing randomly first.



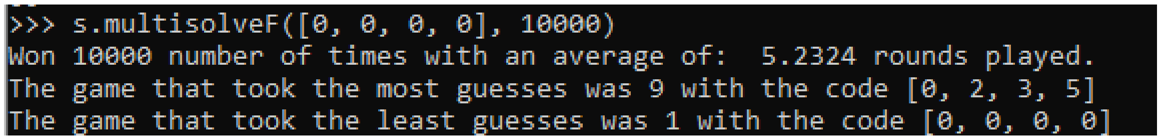
```
>>> s.multisolveF([0, 1, 2, 3], 10000)
Won 10000 number of times with an average of:  4.7662 rounds played.
The game that took the most guesses was 8 with the code [2, 1, 3, 4]
The game that took the least guesses was 1 with the code [0, 1, 2, 3]
```

In a similar vein, I decided to see how bad the guesses would go, and had the first guess for my next test be [0, 0, 0, 1]. This sure enough gave a worse result, of an average of 4.824 rounds being played. Not terribly worse, and it would still solve within 10 turns at most, but it was definitely less than ideal for a first move.



```
>>> s.multisolveF([0, 0, 0, 1], 10000)
Won 10000 number of times with an average of:  4.824 rounds played.
The game that took the most guesses was 8 with the code [1, 1, 0, 2]
The game that took the least guesses was 1 with the code [0, 0, 0, 1]
```

Finally, the only way to go is up, so I tested what the result would be if I used 4 of the same color as my first guess. Sure enough, the average number of rounds played went up to 5.2324, which was much worse. Additionally, the solver took 9 rounds for the longest round, which one more than this algorithm was used to seeing.

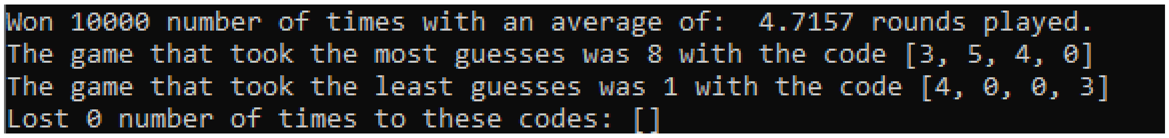


```
>>> s.multisolveF([0, 0, 0, 0], 10000)
Won 10000 number of times with an average of:  5.2324 rounds played.
The game that took the most guesses was 9 with the code [0, 2, 3, 5]
The game that took the least guesses was 1 with the code [0, 0, 0, 0]
```

Red and White Values

The next thing I wanted to test was the values I assigned to my red and white pegs (as explained in the previous section). As the "probabilities" being calculated are more akin to a specific value that I assign each of them for how likely they are, I only messed around with the value of the white pegs to see if there would be a difference in the base model.

To my surprise, there were no noticeable differences with changing the value of the white pegs. I first started slowly lowering the value to see if there would be any effects, but there were none. In fact, I set the value for a white peg to be 0, and it still did roughly the same as the baseline of 3/16. This leads me to believe that you could potentially solve mastermind effectively by only focusing on the getting red pegs, however this is most likely suboptimal for human players. I then decided to go in the other direction of changing the white peg value, and found that a white peg being equivalent in value to a red peg made no difference either. Just to see what would happen, I tested making the value of a white peg larger (setting it to 5/16), and the average was roughly 7.1, so while it was not too different from the baseline, it will perform worse. (I changed the value back to 3/16 for consistency for the last tests)



```
Won 10000 number of times with an average of:  4.7157 rounds played.
The game that took the most guesses was 8 with the code [3, 5, 4, 0]
The game that took the least guesses was 1 with the code [4, 0, 0, 3]
Lost 0 number of times to these codes: []
```

Different Colors and # of Pegs

The final component I played around with for my solver was how many pegs and colors there were. This is the most "applicable", as varying versions of mastermind can have varying numbers of colors and pegs. I specifically built the mastermind module to be able to vary this as well, so I thought I would see how well my model could go. I tested up to 10 colors and my solver unfortunately can only handle 4-6 pegs. To simplify the results I found, I have placed all of them on the following table. Note that the total possible number of hypotheses are (# of colors)$^{(\# \text{ of pegs})}$, so some of the results for more pegs and colors may not be as representative as I would hope, however the differences are large enough that it gives a good idea of what type of impact changing the number of colors/pegs does.

	6 Colors	7 Colors	8 Colors	9 Colors	10 Colors
4 Pegs	4.715	5.128	5.535	5.93	6.292
5 Pegs	5.117	5.556	5.93	6.279	6.708
6 Pegs	5.532	5.952	6.387	6.83	7.26

As seen above, as expected, the average number of rounds to solve is increased. What I did not expect was that the increase seems almost symmetrical for increasing colors by 1 or increasing number of pegs by 1. For some of the last few tests, I changed the maximum number of guesses just so that the averages would be more representative of what they should be, but the largest I ever found the number of guesses required was 11 guesses required. There is still a lot of testing to be done here, but the main takeaway is that the average number of rounds needed for the solver to solve a game of mastermind is linearly proportional to the number of pegs and colors.

Conclusion

Overall, using a solver that used hypotheses to determine its next move worked out better than I anticipated. Although I was unable to implement an effective algorithm, I found some interesting results that are harder to notice as a human who plays the game. I chose mastermind because handling a game with min-maxing elements was more interesting, however in the future it will be interesting to see how computers can possibly learn other types of games where the right choice may not seem obvious, such as Magic the Gathering, or even if the right choice is hard to determine because of incomplete information, such as Hanabi.