Keno in Macau: A Brief Investigation of Its Mechanics and Risks

Ruben Bentley

August 2024

1 Background

After my graduation, I went travelling around Asia, where I visited some casinos in Macau (MGM and Venetian casinos). I was introduced to Keno, a game originating from ancient china with similarities to the modern lottery.
Keno is played on a board with 80 numbers, each round 20 random numbers are highlighted. Players select a specific number of "pins" (numbers), with the payout increasing as they correctly select more. However, the more pins you choose, the lower the payout for each correct pin. This creates an important trade-off: opting for more pins increases the probability of hitting a winning number, while fewer pins offer the potential for higher payouts.

The probability of matching x pins in an N pin game is shown in Eq. (1),

$$P(x|N) = \frac{\binom{N}{x} \times \binom{80-N}{20-x}}{\binom{80}{20}}. \tag{1}$$

where:

- $P(x|N)$ is the conditional probability of matching x numbers given N pins.
- $\binom{N}{x}$ is the combinations the x matching pins take.
- $\binom{80-N}{20-x}$ is the combinations the rest of the drawn pins take.
- $\binom{80}{20}$ is the total combinations.

The expected values and probabilities where calculated for specific instances occurring in this Keno game. 10 pin Keno was shown to have a positive expected value (Eq. 2), for the odds given (shown in Table 1). Using Eq. (3), we can calculate the probabilities of the events shown in Table 1, with the fair decimal odds being the reciprocal of the true odds.

$$
\begin{aligned}
\mathrm{EV} &= \left[\sum_{x=5}^{10} \text{pay-out}(x) \cdot P(x|10)\right] - \text{stake} \\
&= 3 \cdot 0.05143 + 15 \cdot 0.01148 + 100 \cdot 0.001611 + 1,000 \cdot 0.0001354 \\
&+ 25,000 \cdot 0.000006121 + 2,500,000 \cdot 0.00000001122 - 1 = 0.05655 \approx 0.06
\end{aligned} \tag{2}
$$

Matching Pins x	Probability $P(x\|10)$	Fair decimal odds	Real odds
5	0.05143	19.44	3.00
6	0.01148	87.11	15.00
7	0.001611	620.70	100.00
8	0.0001354	7384.00	1000.00
9	6.121e-06	163400	25000.00
10	1.122e-07	8912000	2500000.00

Table 1: 10 pin game statistics to 4 significant figures for a unit stake of 1

$$P(x|10) = \frac{\binom{10}{x} \times \binom{70}{20-x}}{\binom{80}{20}}. \tag{3}$$

2 Simulations

A Keno code was written to simulate the game. The foundation of the code was as follows,

```
class Keno():
    def __init__(self, stake, user_picks, odds):
        self.stake = stake
        self.odds = odds
        self.draws = []
        self.user_picks = user_picks
        assert len(self.user_picks) == 10


    def randomly_draw_keno(self):
        self.draws = [random.randint(1, 80) for _ in range(20)] #get 20
            random integers between 1 and 80

    def find_matches(self):
        self.matching = len([num for num in self.user_picks if num in
            self.draws]) #match the numbers
        return self.matching

    def find_payout(self):
```

```
        self.payout = self.odds.get(self.matching, 0) #work out payout
            from the odds dictionary
        return self.payout
```

The implementation of this class is as follows,

```
    #set odds from sheet
odds = {
    5: 3,
    6: 15,
    7: 100,
    8: 1000,
    9: 25000,
    10: 2500000
}

#select my picks
selection = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

#setup class
game = Keno(
    stake=1,
    user_picks=selection,
    odds=odds
)

#draw the numbers
game.randomly_draw_keno()

#see how many match
matches = game.find_matches()

#find the payout
payout = game.find_payout()

#print statements
print(f"Keno numbers: {game.draws}")
print(f"Your numbers: {game.user_picks}")
print(f"Matches: {matches}")
print(f"Payout: ${payout}")
```

Once running it,

```
Keno numbers: [26, 2, 59, 60, 68, 50, 28, 62, 48, 73, 20, 44, 72, 25,
    77, 38, 62, 31, 67, 59]
Your numbers: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Matches: 1
Payout: $0
```

From this simple class, more complex functions where constructed to run Monte Carlo simulations of the Keno games. This code can be seen in Appendix A. Table 2, shows the parameters of two player archetypes,

Player x	Pot (£)	Bankroll Management Strategy
Ordinary	100	fixed £1 payments
Mathematician	Arbitarily large	Kelly Criterion

Table 2: Player descriptions

The Ordinary player plays with with a small pot and places fixed £1 payments, whilst the mathematician attempts to maximise profits using the Kelly criterion Eq. 4 and has a large pot to mitigate risk.

$$f^* = \frac{bp - q}{b} \tag{4}$$

where:

- b is the decimal odds - 1
- p is the probability of winning (add up probabilities shown in Table 1).
- q = 1-p the probability of losing.

Matching Pins x	p	b	f^*
5	0.05143	2.00	-0.4228
6	0.01148	14.00	-0.05913
7	0.001611	99.00	-0.008474
8	0.0001354	999.00	-0.0008654
9	6.121e-06	24999.00	-3.388e-05
10	1.122e-07	2499999.00	-2.878e-07
sum	-	-	-0.4914

Table 3: Kelly fractions for x pins out of 10

Table 3 tells us the mathematician would avoid this game, whilst using the Kelly criterion bankroll strategy.
The expected value is positive ($\approx$ 6%), which means that the player has an advantage in the long run, but the caveat is that the initial high volatility (large loss potential) may prevent the player from becoming profitable if they have a small pot.
This is shown for a gambler with a £100 pot (small pot) who places £1 bets each game. They are very likely to end their games in ruin, as shown in the bottom panel of Figure 1, where the percentage of simulations that tend to ruin approaches 100%, and also the percentage of gamblers who beat the EV

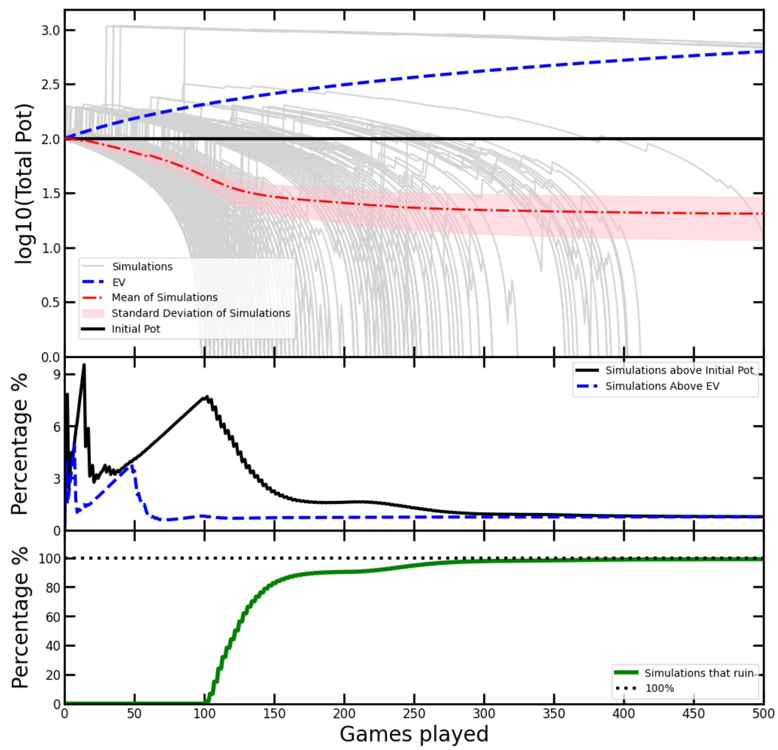

Figure 1: 1,000,000 Monte Carlo simulations for a £100 pot player playing 10 pin Keno. The top panel shows the log10(Total Pot) as the Gambler plays more games, some simulations, the initial pot, as well as the mean and standard error of the simulations. The middle panel shows the percentage of simulations above the EV and initial pot. The bottom panel shows the percentage of simulations that reached ruin (run out of money).

decreases for a higher number of games played (middle panel).
This is contrary to the theoretical understanding of a positive EV and the law of large numbers. This is due to the high volatility present in the game, and negative Kelly bankroll. In addition the middle panel also shows that the percentage of profitable gamblers decreases with a higher number of games played, and the top panel shows that the mean pot for each game (simulation step) decreases as more games are played, and the standard error increases.
The mean pot decreases as more simulated gamblers hit ruin and fail to reach high payouts, let alone reaching EV. The increase in the standard error tells us that there is a higher variability for more games played, meaning high payouts happen, however a pot large enough to sustain you for this to possibly to happen is not practical or prohibited within the casinos of Macau.

The data tells us that this game should not played, because the theoretical implications (positive EV and law of large numbers) are outweighed by the high volatility (and negative Kelly bankroll) which is caused by: the large difference in payout magnitude, small probability of getting a high payout, and the binary nature of the game.

3 Biased Keno Game

You can statistically justify specific numbers being over selected with a hypothesis test,

- Null Hypothesis H_0: the probability of selecting any number during a game of Keno is equal to $\frac{1}{4}$.
- Alternate Hypothesis H_1: the distribution of number selection is not equal.

We will use a 5% significance level (α), and find the p-value.

- If p-value $< \alpha$ we reject the H_0.
- If p-value $>= \alpha$ we accept the H_0,

With the aid of a high number of simulations (for statistical significance), we can find the χ^2 statistic Eq. (5).

$$\chi^2 = \sum \frac{(observed - expected)^2}{expected} \tag{5}$$

The chi-square cumulative distribution function can be used to find the p value, to complete the analysis shown above.
A class was created to store all relevant functions for this experiment.

```
class KenoNumbersDistribution:
    def __init__(self, num_split, draw_split, simulations):
        self.simulations = simulations
        self.split = num_split
        self.draw_split = draw_split

    def number_counter(self):
        self.counts = np.zeros(80, dtype=int)

        for _ in range(self.simulations):
            draws1 = [random.randint(self.split, 80) for _ in
                range(self.draw_split)]
            draws2 = [random.randint(1, self.split-1) for _ in range(20 -
                self.draw_split)]
            self.draws = []
            self.draws.extend(draws1) # Faster than np.concatenate
            self.draws.extend(draws2)
```

```python
        for num in self.draws:
            self.counts[num - 1] += 1 # Adds count to each number in
                the counts array
        return self.counts

    def chi_square_function(self):
        self.expected = 20 * self.simulations / 80 # Draws * sims /
            numbers
        self.chi_square = np.zeros(80)
        run_count = 0
        for value in self.counts:
            self.chi_square[run_count] = ((self.expected - value) ** 2) /
                self.expected
            run_count += 1
        return np.sum(self.chi_square)

    def p_value(self):
        self.p_value = 1 - chi2.cdf(np.sum(self.chi_square), 79) # 79
            degrees of freedom (80 - 1)
        return self.p_value

    def run_simulations(self):
        self.number_counter()
        self.chi_square_function()
        return self.p_value()
```

For proof of concept, the distribution of the drawn numbers was artificially disrupted.
This was accomplished by creating a drawing system that splits the 80 numbers into two at the num_split point, and samples the larger numbers draw_split times out of the 20. This method can be used to create a Keno game where the numbers are not evenly distributed.
The implementation of the class for the disrupted number distribution is as follows,

```python
question6 = KenoNumbersDistribution(
    num_split=76,
    draw_split=4,
    simulations=10000000
)
question6.run_simulations()
```

These parameters ensured that the percentage of numbers that are selected from 76 to 80 was 20% (4% per number) compared to 80% for numbers between 1 and 75 ($\approx 1\%$ per number). The result is as follows,

$$\text{p-value} = 0.04 < \alpha \tag{6}$$

Therefore, there is significant evidence to reject the H_0, and statistically justify that the numbers are not normally distributed.

For completion, tests were done for a fair Keno game, and its implementation is shown below.

```
question6 = KenoNumbersDistribution(
    num_split=80,
    draw_split=0,
    simulations=10000000
)
question6.run_simulations()
```

Which gives,

$$\text{p-value} = 0.5958 > \alpha \tag{7}$$

Thus, we accept the H_0, which is expected.
To perform these experiments, a high number of simulations was used to ensure statistical significance.

4 Further Work

Investigate other Keno games (all pin game possibilities), and develop strategies to mitigate long term risks in mathematically favourable circumstances. Explore the demographics of players, and identify common patterns in player behaviour using machine learning models.

A Appendix: Full Keno Class

```
    class Keno():
    def __init__(self, stake, user_picks, odds):
        self.stake = stake
        self.odds = odds
        self.draws = []
        self.user_picks = user_picks
        assert len(self.user_picks) == 10

    def randomly_draw_keno(self):
        self.draws = [random.randint(1, 80) for _ in range(20)] #get 20
            random integers between 1 and 80

    def find_matches(self):
        self.matching = len([num for num in self.user_picks if num in
            self.draws]) #match the numbers
        return self.matching

    def find_payout(self):
```

```
        self.payout = self.odds.get(self.matching, 0) #work out payout
            from the odds dictionary
        return self.payout

    def play_game(self): #this is used in the
        monte_carlo_simulation_cinematic
        self.randomly_draw_keno()
        self.find_matches()
        payout = self.find_payout()
        self.balance += payout - self.stake
        return self.balance

    def monte_carlo_simulation_cinematic(self, simulations,
        initial_balance, plays): #this monte_carlo stores all the
        simulations data, and is used for plotting
        self.all_balance_history = []
        for _ in range(simulations):
            self.balance = initial_balance
            self.balance_history = []
            step = 0
            while step < plays: #gives the oppurtunity to play a maximum
                number of goes (plays) or until you ruin
                if self.balance > 0:
                    self.balance_history.append(self.play_game())
                else:
                    self.balance_history.append(0)
                step += 1
            self.all_balance_history.append(self.balance_history)
        return self.all_balance_history

    def monte_carlo_simulation(self, simulations, initial_balance,
        plays):
        #this one average every simulation step (game played)
        self.balance = initial_balance
        self.averages = [0] * plays
        self.averages[0] = initial_balance
        self.standard_deviations = [0] * plays
        # counting the number above the EV and inital pot, and the
            number of ruined simulations
        self.count_above_EV = [0] * plays
        self.count_above_pot = [0] * plays
        self.count_ruin = [0] * plays
        #load the current state with initial balance
        self.current_state = [initial_balance] * simulations
        self.next_state = [0] * simulations

        # need to load the next state with payouts from plays #vectorise
            it to make it faster
        for play in range(1, plays):
            for sim in range(0, simulations):
```

```
                if self.current_state[sim] > 0: # Check if the current
                    balance is greater than 0
                    self.randomly_draw_keno()
                    self.find_matches()
                    payout = self.find_payout()
                    self.next_state[sim] = self.current_state[sim] +
                        payout - self.stake # stake
                    if self.next_state[sim] > initial_balance:
                        self.count_above_pot[play] += 100/simulations
                        if self.next_state[sim] > EV(play) +
                            initial_balance:
                            self.count_above_EV[play] += 100/simulations #
                                get a percentage

                else:
                    self.next_state[sim] = 0 # Set the balance to 0 if it
                        goes below 0
                    self.count_ruin[play] += 100/simulations

            self.averages[play] = np.mean(self.next_state)
            self.standard_deviations[play] = np.std(self.next_state)
            self.current_state = self.next_state
           # print(f'play{play+1}', self.current_state, f'avr{play+1}',
               self.averages, f'std{play+1}', self.standard_deviations,
               f'count_EV{play+1}', self.count_above_EV,
               f'count_pot{play+1}', self.count_above_pot,
               f'count_ruin{play+1}', self.count_ruin)
        return self.averages, self.standard_deviations,
            self.current_state, self.count_above_EV,
            self.count_above_pot, self.count_ruin
```

The implementation of the class is as follows (for a gambler with £1000, playing 1000 goes, simulated 1 million times),

```
initial_balance = 1000
simulations = 1000000
plays = 1000
game = Keno(
    stake=1,
    user_picks=selection,
    odds=odds
)
averages, std, current, above_EV , above_pot, ruin =
    game.monte_carlo_simulation(simulations=simulations,initial_balance=initial_balance,
    plays=plays)
```