The Probability of Winning at Blackjack

Will Dudley

December 10, 2015

Abstract

Blackjack, also known as twenty-one, is the most widely played casino banking game in the world[1]. It is a game of high random chance between the player and the dealer, thus heavily relying on a decent knowledge of probability. How risky should one be when playing to win such a game?

This piece of work explains the key parts in assessing how risky one should be when playing Blackjack, often referring to the code which can be accessed through the following link: *http://tiny.cc/shipit*

1 Rules of Blackjack

All participants (including the dealer) are dealt two cards. Face cards score 10 each, Aces can score 1 or 11 and the other cards score there respective values. To win, the player must have a hand with a score total **strictly** higher than the dealer's, without going over 21 (known as going 'bust'). The player may draw as many cards as they wish (known as 'hitting') until they stand or go bust. If the player goes bust, they instantly lose. If the player sticks to a total without going bust and the dealer goes bust, the player wins. This is summed up in *lines 117-133* in the code:

```
def compare(self):
    if self.total > 21:  # if the player goes bust, the player automatically loses
        winlosslist.append("Lose")
    elif dealertotal > 21:  # if the dealer goes bust where the player hasn't, the player wins
        winlosslist.append("Win")
    elif dealertotal >= self.total:  # if the dealer's total is greater or equal to the player's total, the
        player loses
        winlosslist.append("Lose")
    else:  # if the player's total is greater than the dealer's total, the player wins
        winlosslist.append("Win")
```

Note that there are more options to the player than hitting or standing. These are doubling, splitting and surrendering[2]. For sake of simplicity it will be assumed that the player can only hit or split. In reality and at a high level of play, however, the consideration of these options have a big impact on strategy.

2 Simulating Cards

2.1 Card Lists

In order to play a card game, cards are needed! The first step to modelling blackjack is to create card lists. However, some dealers play with different numbers of decks, so to create the card list the following code (*lines 15-16*) is used:

```
for i in range(numberofdecks):
            cardlist = [11] * 4 + range(2, 11) * 4 + [10, 10, 10] * 4
```

2.2 Dealing Cards

Once the card list is formed the dealer deals cards from the card list (which acts like the deck(s) of cards that are in play). In reality the cards are shuffled, so a random element in the card list needs to be chosen, and removed from the list. The pop function is useful for this (*lines 64-75*):

```
def dealcard(self):
    e = cardlist.pop(random.randrange(len(cardlist)))  # 'pop' returns an element from cardlist and
removes it from cardlist, and 'random.randrange' generates a random index in the range 1 to the first
argument (in this case, the length of cardlist)
    self.hand.append(e)  # adds the element to the hand of the target player
    self.total = sum(self.hand)  # recalculates total
    self.revalueace()  # re-checks if you can reduce an ace's score to avoid going bust
```

The above function is carried out twice per participant every round (dealing two cards), and again whenever a participant hits.

3 Calculating Probability

When a player hits, they do not want their current total plus the score of the card about to be drawn to be over 21, as they'd go bust and instantly lose. The probability of going bust if hitting can be calculated by the equation (3.1) shown below, recalled from the basics of probability:

$$P(\textit{bust on next card draw}) = \frac{\textit{Number of unknown cards left that would make you go bust}}{\textit{Number of unknown cards}} \tag{3.1}$$

Usually, the dealer only shows the player one of their cards (known as the 'hole card'), so the player doesn't know the dealer's other card. Hence to calculate the probability from the player's perspective that other card needs to be included in the formula (explaining the use of the word 'unknown').

The calculation of said property is located in *lines 78-99*. It contains a 'for' loop as well as 'if' statements to individually identify and count all unknown cards that would cause a bust, taking into account that aces can score 1, not only 11. It does this by appending an empty list of said cards, then taking the length of the list. This can also (in a more pythonic manner) be achieved by creating a 'count' variable starting at 0 and adding 1 for each said card. However, if a list is made, potential errors are easier to spot and debug as it is possible to print the list and see its elements.

4 Playing the Game According to the Calculated Probability

Now that the probability of going bust if hitting can be calculated, it can be used for assessing whether to hit or stand. The process of the game is as follows:

1. The dealer takes two random cards from the "shuffled" list of cards and makes his first card (the hole card) known to the player.
2. The player receives two random cards.
3. A list of unknown cards is generated from the player's hand and the hole card (in *lines 45-48* and *86*), used to calculate $P(bust\ on\ next\ card\ draw)$ as explained in Section 3 for use in deciding whether to hit or stick. To do this, $P(bust\ on\ next\ card\ draw)$ is compared to a predetermined threshold and if the threshold is **not** higher (and the player hasn't gone bust), the player hits.
4. Step 3 is repeated until the threshold is higher or the player has gone bust by use of a 'while loop' in the code. Finally, if the player is bust and they have one or more aces, the code sets the score of all of the aces in the player's hand to 1. The code being referred to in steps 2 and 3 may be found in *lines 102-114*.
5. The dealer then hits or stands based on whether or not their total is under 17 or not, as per standard blackjack rules. If it is, the dealer hits until it isn't. Note there is usually a condition where if the dealer gets 17 with an ace (known as a 'soft 17'), they hit - this rule is not acknowledged in this model, however.
6. The player total is then compared with the dealer total and a winner is determined by the process mentioned in section 1.

The above cycle can be seen in *lines 162-170*, however this is purely initializing/executing predetermined classes/functions in a logical order. All of this is defined in a function called `wlr()` (as it approximates win/loss ratios, explained in Section 5) which just runs the whole process described, finds out if the player wins or loses and counts how many wins occur in a given number of games.

5 Approximating Win/Loss Ratios

The arguments that `wlr()` takes are the probability threshold, number of decks in play (usually 8 in casinos, but for now it's set to 1) and the sample size (number of games played). By simulating a large number of rounds (set to one million games played by default), a win/loss ratio (abbreviated to WLR, also known as win rate) can be calculated for a given probability threshold (keeping the number of decks in play set to 1) using equation (5.1) given below.

$$WLR = \frac{Number\ of\ wins}{Number\ of\ games\ played} \tag{5.1}$$

The code does this as the final part of the `wlr()` function in *line 173* (also making the value a global variable from the previous line to enable use outside the function):

```
global wlratio  # Allows the win/loss ratio to be called outside the function.
wlratio = float(winlosslist.count("Win") / samplesize)  # Calculates the win/loss ratio...
```

5.1 Varying the Probability Threshold

The `wlr()` function can now be simply performed multiple times for varying probability thresholds. This is done by creating a list of varying thresholds as *line 178* shows, then performing the function for each element in the list:

```
listofprobabilitythresholds = [round(x * 0.05, 2) for x in range(0,21)]  # obtains a list of numbers from 0
    to 1 with step 0.05 rounded to 2 decimal places (Python doesn't like dealing with steps which are floats
     in its range function)
listofwlratios = []  # creates an empty list for the win/loss ratios to go into, so they can then be
    compared with the corresponding probability thresholds
for i in listofprobabilitythresholds:  # adds the (estimated) win/loss ratio corresponding to the respective
     probability threshold to listofwlratios
    wlr(i)
    print wlratio  # useful for keeping a rough track of progression when the calculations are running
    listofwlratios.append(wlratio)
```

The new list of WLRs can then be compared with their respective thresholds in table and scatter graph form to help answer the initial question "How risky should one be when playing to win such a game?"

6 Results/Conclusion

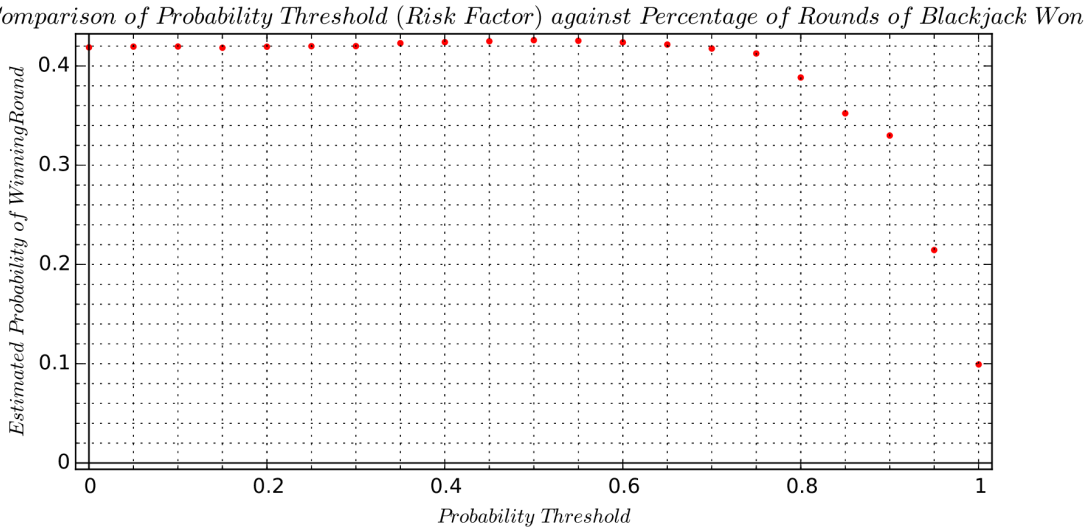

Figure 6.1: Comparison of win/loss ratio and probability threshold

x	0.0	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45	0.5
y	0.418734	0.419479	0.419588	0.418349	0.419371	0.4199	0.419972	0.422933	0.424007	0.424899	0.425951
x		0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95	1.0
y		0.425359	0.423809	0.421572	0.417518	0.412548	0.388357	0.352263	0.329886	0.214548	0.099305

Table 6.1: A table of the data used in Figure (6.1)

Figure (6.1) shows how as greater risks are taken in Blackjack, on average (in the long term), the player's win rate decreases dramatically. It is worth noting that, as Table (6.1) shows, when the threshold is 0, the player always sticks - in this case they would win 41.9% of games. Furthermore, when the threshold is 1, the player always hits unless he gets blackjack - so they would only win if they get blackjack and the dealer doesn't. This happens in about 2% of cases.

The WLR (y axis) starts at 41.9%, appears to peak at 42.6% when the threshold is 0.5. The WLR stays above 40% until past a threshold of 0.75, then it starts to decrease dramatically. It is interesting to note that never hitting would be almost as successful as the use of higher thresholds up to around 0.75.

To conclude, the variation in WLR is so low between 0 and 0.75 (under 0.7%) that the level of risk one takes doesn't really matter until $P(\textit{bust on next card draw})$ reaches over 0.75 which would cause the average win rate to decrease. The graph only heavily decreases at the higher side of the 'optimal threshold' of 0.5: This suggests that in general it is far better to 'play safe than sorry'. This conclusion is useful as in a real-life game of blackjack the player would not be able to perform exact computations on their own so if the player feels like they have a probability of going bust of more than 75%, they should stick. However as the WLR is always below 0.5, it is near impossible to profit on this specific version of blackjack in the long run.

7 Assumptions and Further Investigations

Some assumptions about the model have already been mentioned. Other assumptions include:

- All of the cards in play are returned and reshuffled every game.
- When ace scores are reduced to 1, **all** aces in the hand are reduced to 1.

Possible further investigations include:

- The effects of card counting on the win/loss ratio under varying numbers of decks in play (where the cards wouldn't be returned).
- The strategy of comparing of player's probability to dealer's apparent probability.
- The effects of having a varying number of players on the win/loss ratio.

References

[1] John Scarne. *Scarne's New Complete Guide to Gambling*. Simon & Schuster, 1986. Cited in https://en.wikipedia.org/wiki/Blackjack#cite_node-1.

[2] Michael Shackleford. Wizard's simple strategy, mar 2013. http://wizardofodds.com/games/blackjack/basics/#toc-BasicStrategy.