Spades: A Bayesian Approach

Cameron Anderson and Emily Guthrie

Introduction

Our goal for this project was to use a Bayesian method of probability to produce a useful tool that would aid in the bidding process of a typical game of spades. Spades is a four person trick based card game. Trick based means that each round of play consists of each player putting one card down, with the highest card of the led suit winning the round, or trick. Spades has an added element of complexity, a trump card, or a suit that can override the led suit. In this case, no matter what suit is led at the beginning of the trick, if a player places down the highest spade of everyone, he or she wins. Prior to the beginning of play, there is a round of bidding, where each player looks at the hand that has been dealt to them, and makes an estimated guess at how many tricks they believe they can win. This is the portion of the game that we have chosen to analyze and, hopefully, optimize.

The Theory Behind our Work

In general, the idea behind our project is to use the knowledge of the cards in our hand, and the bid of our opponent, to help us decide on a strategy for how we should bid. The first step in this process is to develop a list of possible hands our opponent could have, and then map a probability to each of these. By this we mean that after shuffling the remaining cards in the deck after our hand is dealt, we deal out a large amount of possible random hands. We then want to calculate the likelihood that with the random hand dealt, our opponent would have given the provided bid. This allows us to look at the bid given, and determine the probability that our opponent has each of the possible random hands.

Calculating Bids

We began by developing a dictionary of cards with a weighting based on how many other cards it can beat. Because spades are the trump suit in this game, they can beat out any other suit played. This means that when we are calculating the probability of a spade winning a trick, the ace of spades wins 52 out of 52 times, the king of spades wins 51 out of 52 times, and so on, all the way down to the 2 of spades, which wins 40 times out of 52. With regard to the other three suits, cards can only win a trick if that suit is led. For the sake of simplicity, we assumed that each suit had an equal chance of being led. Because a card can only win a trick if its suit was led, for every suit that is not a spade, each card has a probability of winning of zero when the suit that is led is not the same as the card. In the case where the led suit is the same, the ace of the suit will

beat out any card other than a spade. This gives it a 39 out of 52 chance of winning. This pattern continues down for each card in the suit as seen in the initialization of Card in figure 1. In order to calculate the weighting of non spade cards, we multiply the probability of winning assuming the suit of the card is led by the probability that this suit will be led, which in this case is 1 out of 4.

```
21 class Card(object):
22     """Represents a standard playing card.
23
24     Attributes:
25       suit: integer 0-3
26       rank: integer 1-13
27     """
28
29     suit_names = ["Clubs", "Diamonds", "Hearts", "Spades"]
30     rank_names = [None, "Ace", "King", "Queen", "Jack", "10",
31                   "9","8","7","6","5","4","3","2"]
32
33     def __init__(self, suit=0, rank=2):
34         self.suit = suit
35         self.rank = rank
36         if suit == 3:
37             self.prob= (53-rank)/52.0
38         else:
39             self.prob= .25*((39-rank)/52.0)
```

Figure 1: Initialization of the card object that now includes our scoring probability attribute with the algorithm described above

In a hand of thirteen cards, the value of the hand is calculated by adding up the probabilities that each card will take the trick. In our simplified model of bidding, this value corresponds to an expected bid for this hand. One limitation of this model is that bids are unnaturally high due to a lack of consideration of suit distribution across bids as well as individual player strategy.

Code Architecture

We have 4 defined classes in our script: card, deck, hand, and scenario. Card, deck, and hand are all objects where deck's are made up of 52 cards with the assigned values, suits, and probabilities of normal playing cards. Hands are comprised of cards moved from the deck and in the case of the game spades there are 13 cards in a hand. Scenario is a pmf based class and has the hands and their probabilities stored within. The run_scenarios function is what actually executes the experiment for n number of trials.

Setting up Trials

Now that we can score hands based on their predicted trick values we came up with an implementation that would compute a large random sample of hands and the value of the cards within them. After pulling out the cards that make up our hand, we shuffle the remaining cards, deal 13 cards into an opponent's hand, and then evaluate this hand using the same logic as we use on our own hand. This is repeated 1000 times using the remaining deck after removing our hand to create a dictionary of trials and their initial probability of 1.

```
def run_scenarios(mydeck, theirbid, num):
    """runs a number of scenarios of possible hand combinations
    given a certain deck configuration where there is a hand's w
    of cards removed.
    num: integer number of trials
    returns scen_distributions"""
    mytrialhand= Hand()
    mydeck.shuffle()
    mydeck.move_cards(mytrialhand,13)
    scen=Scenarios()
    for i in range(num):
        theirtrialhand=Hand()
        mydeck.move_cards(theirtrialhand,13)
        theirtrialhand.update()
        scen.Set(deepcopy(theirtrialhand),1)
        theirtrialhand.move_cards(mydeck,13)
        mydeck.shuffle()
    scen.Update(theirbid)
    itms=[]
    for key, value in scen.d.items():
        itms.append((value,key))
    return sorted(itms, reverse=True)[:5], mytrialhand
```

Figure 2: the scenarios code that shows the creation of trial hands and the storing of them in the pmf scen. Then resetting the deck and reshuffling for num number of trials.

In order to generate useful information from this dictionary, we need to compare the score of each potential hand to the bid made by the opponent. We do this by creating a PDF across all potential bid values for each scenario, where the function looks like a normal gaussian curve centered around the value of the hand minus some amount (0.5) to account for the artificially high number that our model calculates as seen in figure 3.

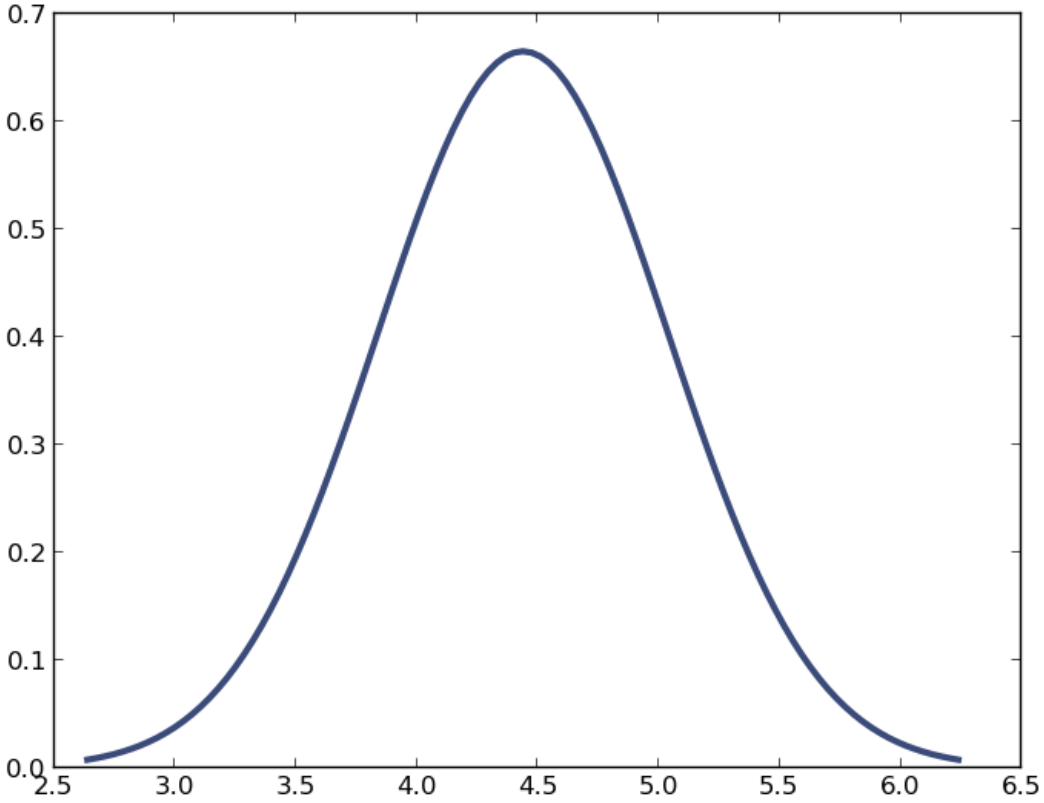

Figure 3: PDF for possible bids on a hand with a score of 4.85. The PDF has properties of u=score-.5 and sigma= .6 because we reasonably assume people bid within a range of three below or above.

When we evaluate the PDF at the bid value that our opponent provides, we can find a value similar to the probability that the scenario we are looking at is the hand our opponent actually has. From there it is a matter of using the possible hands with the highest probability to help determine what your strategy should be when making your bid.

Conclusions

Our system is now outputting the probability a random hand dignified as mytrialhand and then a pmf of random trial hands and their probabilities of being our opponents' hand given the bid our opponent has made, as seen in implementation in figure 4.

```
163 if __name__ == '__main__':
164     mydeck=Deck()
165     theirbid=5
166     tophands, mytrialhand = run_scenarios(mydeck,theirbid,1000)
167     print 'Their bid was ',theirbid, '\n My hand was:\n', mytrialhand, '\n\n Their most likely hands are '
168     for pair in tophands:
169         print '\n', pair[1], '\n Probability \n', pair[0], '\n\n'
170 #   print hands[0][1], '\n Prob = ', hands[0][0], '\n\n', hands[1][1], '\n Prob = ', hands[1][0], '\n\n',
```

Figure 4: The running of our trials and output of the bid, hand, and most likely trial values of the 1000 run.

```
eguthrie@EGPC:~/Documents/Bayesian-Spades$ python __weights__.py
Their bid was  5
 My hand was:
Ace of Clubs
Ace of Hearts
Ace of Diamonds
4 of Spades
8 of Hearts
3 of Clubs
6 of Spades
9 of Spades
2 of Spades
7 of Diamonds
4 of Hearts
Queen of Diamonds
4 of Diamonds

 Their most likely hands are

2 of Diamonds
5 of Spades
10 of Spades
8 of Clubs
8 of Spades
9 of Diamonds
10 of Diamonds
3 of Hearts
6 of Clubs
7 of Spades
3 of Spades
King of Clubs
Jack of Diamonds
 Probability
0.00355064012055
```

Figure 5: Partial output of the running of the script showing the trial results.

At this point in time, our code gives us the ability to make assumptions about the hand of our opponent based on their bid, as it provides us with the knowledge of which hands had the highest probability of being the actual hand, out of a pool of 1000 random possibilities. Because we are only running 1000 possible scenarios, this is not going to provide us with the exact hand of our opponent. There are too many possible permutations of the 39 remaining cards to evaluate them all, and there is a very good chance that a hand that would match the bid of the opponent better is in one of these permutations that have not been evaluated. That being said we can get an estimate of what the caliber of a hand could look like and if we have a specific hand in mind we can work forward to estimate what the probability is that they have it.

Future work could include the improvement of assumptions in our model such as making the weightings of the cards be dependent on the distribution of cards in your hand, modifying the distribution of bids people make to be more accurate than a shifted gaussian, and adapting for different strategy modes such as the concept of going nill in spades (not super relevant to know what this is, but it substantially changes strategy). We also could have worked on a mechanism to play forward the hands between two players showing how the two hands matched up, making assumptions about strategies along the way.

Git Repo: https://github.com/eguthrie/Bayesian-Spades Our code is written in the file __weights__.py