Date of acceptance Grade

Instructor

Machine Learning Strategies in Cribbage

Sean R. Lang

Helsinki December 15, 2017

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Sean R. Lang

Työn nimi — Arbetets titel — Title

Machine Learning Strategies in Cribbage

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	December 15, 2017	13 pages + 0 appendices

Tiivistelmä — Referat — Abstract

ACM Computing Classification System (CCS):
temporary stuff

Avainsanat — Nyckelord — Keywords
machine learning, cribbage

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

I just want to see if unicode characters will appear. Jag måste se om unicode-characterer funger. minä haluan katsoa kirjain öööö.

Generic introductory stuff giving a sentence or probably a paragraph about each of the sections covered.

1.1 Cribbage

Cribbage is a multi-phase card game, typically played between two opposing players. While variants exist for three or more players, this paper will focus on the two-player variant. The game presents an interesting research area because of its unique scoring methodology: each hand is counted in two slightly different ways within each round and the first player to reach as score of 121 points or more is declared the winner. Players will usually keep track of their points by *pegging* them on a characteristic board. Because of its atypical win condition, different strategies hold differing levels of importance throughout the game.

1.1.1 Rules of the Game

In order to be able to understand the crucial nature of the temporally dependent strategies, the rules and flow of a game of cribbage must be fully understood. While a complete set of tournament rules can be found at [ACC], what follows is an overview complete enough such that a novice player, following the scoring rules found in Appendix A could play a complete game, albeit likely not well.

The zeroth step, taken once per game, is to determine which player will be the dealer for the first round and who will be the pone. In order to determine these roles, each player cuts the deck once to get a single card: the player with the lower-valued card[1] is the dealer; the other player is called the pone. In the case of a tie, this step is repeated until two unique cards are cut from the deck. From there, the usual round structure begins and proceeds in the following steps:

1. Each player is dealt 6 cards.
2. Each player selects 4 cards to keep for their own hand and 2 cards to toss into the crib.
3. A random card is cut from the remaining cards of the deck and placed face-up on top of the deck. If this cut card is a Jack of any suit, the dealer is awarded 2 points and pegs the points accordingly.
4. Starting with the pone, each player alternates playing a single card, keeping track of the total value of all card played so far, until all cards have been played,

[1] Ace $<$ 2 $<$ 3 $<$ 4 $<$ 5 $<$ 6 $<$ 7 $<$ 8 $<$ 10 $<$ Jack $<$ Queen $<$ King

or neither player can play a card without exceeding a collective value of 31. In the latter case, the player last to play a card will be awarded points before the count is reset and play continues. If any of the combination of cards mentioned in Appendix A is seen in the immediately preceding cards, the amount of points earned is immediately pegged on the board for the appropriate player. In cribbage terms, this is called *the play*.

5. After all cards have been played, the pone then counts his or her hand using the randomly cut card as a 5^{th} card in hand before pegging these points on the board.

6. The dealer then proceeds to count his or her hand and peg the points in the same fashion, also considering the randomly cut card to be the 5^{th} card in the hand.

7. The dealer then does the same for the crib.

8. The dealer and the pone swap roles and repeat from step 1.

If at any point a player achieves a score greater than or equal to 121, that player is immediately declared the winner and the game is over.

The win condition for this game can occur at any moment of the game, even beyond either player's control (note Step 3). Because of this, it is crucial to play according to different strategies during different times of the game, where time can be defined by what score the player has, combined with what score their opponent has and which player is the dealer for that round. Typically, during early and middle-game play, the pone will attempt to maximize their own hand, while avoiding giving too much opportunity for the dealer to score points from the crib. However, in later play, this may no longer be a concern. For example, should the player be the pone and their score is 116 and the dealer has 117 points, due to the counting precedence, the player needs not concern themselves with what points the dealer will obtain through the crib if their own hand has at least 5 points guaranteed since the pone will count first and win. This only works, however, if the pone does not allow the dealer to score 4 points from *the play*. As can be seen, the player must balance multiple competing factors with varying emphasis over the course of the game.

2 Literature Review

Overview of the current literature surrounding this topic.

- Research done in cribbage
- Research done in related imperfect information games (e.g. poker)
- Overview on expert witness machine learning
- Any other topic that ends up getting used (Bayesian logic, statistics?)

2.1 Prior Cribbage Research

3 Data and Methods

Walk-through of how I went about researching the topic. (Maybe I should keep a diary or log or something so this isn't half made up at the end.) When all is said and done, include the final "output" graph in some easily viewable format (121-by-121 table of miniature bar graphs, RGB combination in each cell?).

- Creation of framework
- Creation of "expert advisor"
- Training method for listener/combiner
- Initializations for training

3.1 Methods

3.1.1 Loss Function

A good loss function is critical to the proper training of a machine learning classifier. In classroom examples in which data can be mapped into a multidimensional space, this is typically accomplished by either a euclidean distance, or some other measure of how "wrong" the prediction is based on how much it varies from what is classified as "correct." In learning to predict using expert witnesses' testimony, this is measured in terms of *regret*. However, even though it is inevitable for the human player to think back on what could have been if he or she had chosen a different arrangement of cards to keep and throw, this is not a practically applicable loss function measure for the following reasons:

1. To evaluate how each hand "would" have done in any specific point in the game, a new "branching" game would have to be run through at that specific point. Basically, this means that a giant search tree would need to be made where the branches are randomly created using draws from the deck. Therefore the notion of "regret" is not easily practically computable.
2. It is difficult to determine what a better hand would have been in any case. For example, a hand that scored higher would not always be better. Furthermore, a hand that uses a higher score may not peg as well, etc. Therefore, it is difficult to be able to determine what is "correct" to calculate a distance from that.

Therefore, it may be useful instead to create a different system of punishment and reward rather than pure Loss. There are two possible routes I can think of at this junction.

1. Use the point spread between yourself and the opponent for the round.

   The problem with this strategy is that it depends on the opponent's performance as well as your own. In the case of our simulation, this is further problematic because the opponent's playing strategy and style is not known, which means this can't be rightfully evaluated.

2. In what position was the player left after playing with this hand.

   (a) Using intrinsic values for each playing state.

   This method relies upon previous knowledge of the game in order to set up position in which the player is at an advantage or disadvantage. While this may be acceptable for this small application, it should not be used in the general case.

   (b) Using a form of back-propagation.

   Each positions value will be computed based on the amount of times a player at that position ended up winning the game, and by how much. This would require the agent to be trained in reverse, so to speak. Early games could start at scores like 119 to 115 and compute which are likely to win, then make their way backwards to lower scores. Another option would be for each agent to keep track of its own path of scores and reward/punish all at the end, perhaps with different weights.

3.1.2 Strategies

A few basic behavioral strategies were coded for the agent to learn over the course of multiple simulated games. These represented most of the basic factors which a player will consider when making their decision as to which cards to keep. These included:

- `hand_max_min`: The hand(s) which has the maximum minimum possible score for the kept cards will be more highly desired. This is equivalent to choosing the hand with the largest guaranteed points.

- `hand_max_avg`: The hand(s) with the maximum average points over all possible cut cards will be the most highly desired. This strategy is useful for trying to maximize the expected score of one's own hand.

- `hand_max_med`: The hand(s) with the maximum median points possible to score will be the most highly desired. Similar to `hand_max_avg`, this strategy is useful when trying to maximize one's hand's expected score.

- `hand_max_poss`: The hand(s) with the maximum possible score will be the most highly desired. This strategy can be thought of as Hail Mary choice for the player trying to score as many points as possible, not taking into account its unlikelihood.

- `crib_min_avg`: The hand(s) with the minimum average amount of points scored in the crib is the most highly desired. This strategy would be a very defensive strategy typically used by the pone to avoid giving points to the dealer.

- `pegging_max_avg_gained`: The hand(s) with the maximum average points gained through pegging is the most highly desired. This strategy would be useful for end-game play in a tight game. For instance, if both players are close to winning, the dealer may choose to forego placing points into their hand and instead try to peg out since it is unlikely that they will get a chance to count their hand at all.

- `pegging_max_med_gained`: The hand(s) with the maximum median points scored during play will be the most highly desired. This strategy is similar to `pegging_max_avg_gained` in its use, but uses a slightly different internal measure for ranking.

- `pegging_min_avg_given`: The hand(s) with the minimum average points scored by the opposing player will be the most highly desired. This is a very defensive strategy also useful at the end of the game to prevent the opposing player from pegging out.

The above definitions refer to a hand's desirability. This can be thought of as an internal ranking of how likely a given strategy would be willing to choose a particular combination of cards. This desirability score is then scaled to to lie in the range $[0, 1]$ with 1 representing the best possibilities. This scaling allows for possibilities which are "almost as good" to not be ignored in later weighting stages.

Because of the massive amount of possible combinations—many of which were unique due to the cards' position affecting the scoring outcome—the evaluation of some of these possibilities in which the crib was involved in real-time took a handful of seconds to evaluate during development. This was deemed much too slow as over the course of hundreds or thousands of simulated games, this delay would very quickly accumulate to performance-affecting delays. As a result, an alternative strategy of pre-computing the required knowledge and storing the values of interest into a database was implemented instead.

There are $\binom{52}{6}$ possible combinations of card which can be dealt to either player. Of these possibly dealt hands, there are then $\binom{6}{4} = 15$ possible combinations of cards that can be kept and "tossed" to the crib. For each of these 15 possible there are a further $(52 - 6) = 46$ possible cards that must be considered as possible cut cards for the kept hand. Additionally, for the thrown set of cards, there are $\binom{46}{2} = 1035$ possible combinations of cards which can be thrown by the opposing player into the crib as well as the $(46 - 2) = 44$ remaining possible cut cards. These cut cards must be considered separately in such a manner of evaluating $\binom{46}{2} \times 44 = 45540$ possibilities rather than simply looking at each of $\binom{46}{3} = 15180$ possibilities because

whether or not the card is in the crib or not can affect the score.[2] Just as crucially, there are small differences in scoring the crib as opposed to scoring the player's own hand that mean that previous evaluations results are not reusable. This means that, altogether, there are

$$\binom{52}{6}\binom{6}{4}\left((46)+\left(\binom{46}{2}\times 44\right)\right)\approx 1.3921\times 10^{13}$$

possible combinations of cards that need to be evaluated in total to fully understand the statistics of a cribbage game for each hand.

Although the majority of this project was coded in Python for its ease of use and speed of development, due to the performance-crucial, basic mathematical operations involved, the overheads of using a higher-level language was deemed too critical and this particular tool was developed in C instead. To put the performance gains into perspective, at its fastest, most parallelly processed state, the Python database populator was estimated to take approximately 4 months to simply list and evaluate all possible scores on a development machine, disregarding the file I/O operations required to write the results to the database. The same functionality, with the addition of file I/O, in the C program would take a relatively mere 14 days on that same machine. Thankfully, most of the framework for this rewrite had actually been developed previously as part of a mental exercise, so the loss in time for the rewrite was minimal and well worth the time saved in execution. Furthermore, access to a high-performance server allowed for futher paralellization which cut the final run time down to approximately 5 and a half days.

Careful consideration and preparation needed to be taken for the retrieval of this information, however. The trillions of combinations could not be quickly searched by card value as doing so would search the entirety of the database on each lookup, decreasing the performance so much as to be worse than simply enumerating the combinations on-demand. A rather simple solution to this problem was to search by index instead of by card comparison. These indices could not simply be stored in the memory of the running program because the sheer size required to store all of the indices at all times would rival that of the database, and its population would take considerable enough time. Thus, a quick and reliable method for creating and recreating these keys needed to be used. As the cards were represented internally as an integer between 0 and 51 (inclusive) and there were only 6 cards for indexing, the concatenation of the cards' digits in (keep,toss) order would create a number with at most twelve digits, with an absolute maximum value of 484950514647 [3], well within the range of numbers addressable by an integer. This index could be created by a very simple process of 6 multiplications and 5 additions while still being guaranteeably unique.

While the combinations of chosen and tossed scores could be evaluated before any games had actually been played, the hands' usefulness during the pegging portion

[2] TODO: example of right jack or something and point to rulebook

[3] Thanks to implementation, the order of cards was guaranteed to be sorted within each tuple.

of the round needed to be evaluated in semi real-time. A single pegging agent was programmed with a simple one-ahead greedy heuristic: the card that gained the most points when played next was selected with ties broken by choosing the lower-valued card that reached that score. This agent was then played against a copy of itself with randomly allocated cards and the results of that round were recorded for each agent to provide an initial knowledge base. These results would then be queried by the agent during the choose phase of the game and contributed to at the end of the pegging phase.

3.1.3 Weighting

For the purposes of this project, how well certain combinations of cards are played with different strategies is not explored. Instead, only the player's position in score-space affects the decision as to which strategies to play by. Put another way, the agent does not care what cards it is dealt as much as where it is located on the board. Each possible score-space location can be thought of as a discrete coordinate defined by the parameters *PlayerScore* $\in [0, 120]$, *OpponentScore* $\in [0, 120]$, and *Dealer?* $\in \{0, 1\}$. At each score-space location is a vector $w_{p,o,d} = [w_1, w_2, \ldots, w_m]$ where m is the number of all possible strategies to be considered. At the beginning of each round, each of m strategies is evaluated for all $n = \binom{6}{4}$ possible combinations of cards kept to produce an $m \times n$ matrix S where $S_{i,j}$ is the desirability of the i^{th} keep/toss combination according to the j^{th} strategy further constrained by $\sum_{i=1}^{m} S_{i,j} = 1 \ \forall j$ and $\sum_{j=1}^{n} S_{i,j} = 1 \ \forall i$. A value vector P of length n representing the total perceived value of a possible keep combination can then be computed by $P = wS$.

3.1.4 Training

After a complete game had been played, the winning and losing agents need to modify their weights in order to decide a "correct" strategy at that time coordinate. The goal of this operation in the case of good choice was to increase the distance between the winning strategy and the rest, and opposite for losses. This distancing operation was easily accomplished by scaling each element of the weights vector proportional to its square before normalizing the vector again. In other words:

$$w_{i,new} = c w_{i,old}^2$$

where c is some constant. Similarly, the adjustment to re-level the playing field was accomplished by scaling in proportion with the inverse square, i.e.:

$$w_{i,new} = \frac{c}{w_{i,old}^2}$$

One noticeable issue with this method, however, is the handling of uniform weights. If a weight vector w exists such that $w_i = w_j \ \forall i, j$, then the scaling operation will equally scale and normalize each strategy, meaning that no adjustment will be made,

effectively leaving uniformly weighted points dead. This was obviously undesired since a blank-slate learning path and end state was a highly desired result of this project. The uniform weights adjustment problem was combated by ... TODO ....

The reinforcement training framework operated very simply. Given an initialization weights file for each agent, a new agent would be created with those stored weights. These agents would then be placed into a game and played against each other. After the game had been completed, the weights for each agent would be adjusted accordingly along the path which the agent took. After a set number of epochs, the agent would save its weight-state to a checkpoint file. This allowed not only for tracking of weight adjustments over time, but also allowed for the ability to more easily recover potential issues and for more promising states to be explored further by using a checkpoint as an initialization state on a subsequent training run. This checkpointing system allowed agents to be mixed and matched together to allow different combinations to learn from one another.

After a certain amount of training, the agent would be manually checked by inspection of the weights file and by playing against a human.

4 Findings

What are the final results of the "experiment."

- Where did each initialization family end up going?
- Were they different or did the agent learn a "single" strategy in general?
- All this and more will be answered . . . after the break!

5 Discussion

This is the difficult part. What does this part mean? And how does it differ from what I've already covered above in the Findings section.

6 Conclusion

Generic closing remarks and rephrasing of the original introduction again in parting. Because this is a thesis and likely fairly long, "In section X, we covered Y" is probably allowed and not tacky.

References

ACC URL `http://www.cribbage.org/rules`.

BDSS02 Billings, D., Davidson, A., Schaeffer, J. and Szafron, D., The challenge of poker. *Artificial Intelligence*, 134,1-2(2002), pages 201–240.

BFGL17 Brown, A., Fisher, G., Gilman, S. and Lang, S., Overhead delivery system for transporting products, April 13 2017. URL `https://www.google.com/patents/US20170101182`. US Patent App. 14/881,217.

KS02 Kendall, G. and Shaw, S., Investigation of an adaptive cribbage player. *International Conference on Computers and Games*. Springer, 2002, pages 29–41.

Mar00 Martin, P. L., *Optimal Expected Values for Cribbage Hands*. Ph.D. thesis, Harvey Mudd College, 2000.

O'C00 O'Connor, R., Temporal difference reinforcement learning applied to cribbage. Technical Report, University of California, Berkeley, 2000. URL `http://r6.ca/cs486/`.

PMASA06 Ponsen, M., Munoz-Avila, H., Spronck, P. and Aha, D. W., Automatically generating game tactics through evolutionary learning. *AI Magazine*, 27,3(2006), page 75.

Pon04 Ponsen, M., *Improving adaptive game AI with evolutionary learning*. Ph.D. thesis, TUDelft, 2004.

RW11 Rubin, J. and Watson, I., Computer poker: A review. *Artificial Intelligence*, 175,5-6(2011), pages 958–987.

SPSKP06 Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I. and Postma, E., Adaptive game ai with dynamic scripting. *Machine Learning*, 63,3(2006), pages 217–248.

A Cribbage Scoring Rules

A.1 During Play Round

A.2 During Counting phase

B Something

C Else