Project Report
Poker Agent
Caesar

DT8042: Artificial Intelligence
2022

Vincenzo Buono, Isak Åkesson

Date

April 20, 2022

Contents

1 Abstract

The following paper formulate, discuss and analyses the implementation and specifics of the 5 card poker AI agent, namely *Caesar*, carried out as a project assignment for the course *DT8042 Artificial Intelligence* at *Halmstad University*. The agent thanks to its *learning capabilities*, *fast hand estimations*, *optimal draw simulations* and *regression prediction* aims to achieve *optimality* by *maximizing its expected utility*(EV). During *preliminary testing phase* the agent achieved a *win-rate* of 96% against a *Random Adversarial Agent* and 98% when ran against a *Reflex Agent*. The **pre-tournament** resulted in *Caesar* consolidating a *win-rate* of 83.3% (considering only valid matches, out of 15 games), while the **final tournament**, revealed a lower 4 out of 6 *win-rate* due to the errors involved in parameter exploration; despite the *high computation overhead Caesar showed promising results* **winning both tournaments and scoring first place**.

2 Introduction

The agent discussed throught the following paper has been designed to compete in an *imperfect information* poker variant named *5 card draw poker*. The game flow has been kept simple, with each round alternating the following phases: *ante, betting stage, draw*, and ultimately, the *showdown* where the round's winner is declared and the pot's value is distributed. In this variant, the winner is identified as the last player still having chips. *Table 1* shows the default server settings. *Imperfect information* games, where only partial knowledge of the environment and game state is known, are typically computational intensive as often characterized by a large unknown search space. Within the *Artificial Intelligence* such problems are typically the subject of the application of, including but not limited to, the use of *Neural Networks* and *Convolutional Neural Networks* [1, 2]. Recent advancements have also been made using *Counterfactual Regret Minimization* showing promising results. [3, 4]. Statistical approaches to classical probability poker problems are also presented in the research literature using traditional probability inferenses, combinatorics, bayes belief networks and partial formulas [5, 6].

Table 1: AI PokerServer Default Settings

	Parameter					
	N.Players	Initial Chips Count	Initial Ante	Ante Raise every	Client Response Time [ms]	Display Sleep Time [ms]
Value	4	200	10	10	4.000	1000

3 Method

3.1 System Overview

Following is presented a *high-level* system overview of the classes and components of the *Caesar* poker agent represented as a simplified *UML Class Diagram* (*Figure 1*).

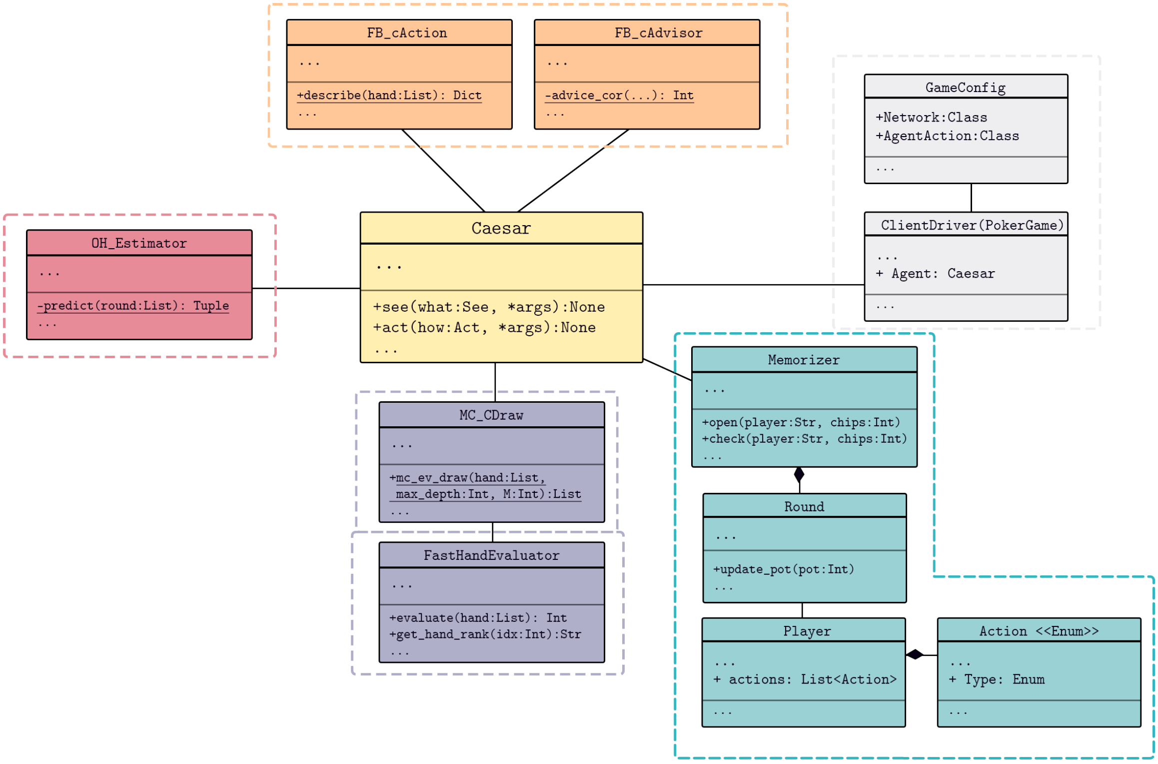


Figure 1: Simplified *UML Class Diagram* of the *Caesar* AI agent and submodules

3.2 Strategy: overview

The agent's overall goal, and subsequently, strategy is to *maximize its winnings*(relative to its chips count) by acting *optimally*. As briefly introduced in *Section 3.1*, the optimality is achieved by subdividing the problem in subproblems and having multiple *modules* taking care of the different phases of the game; each with their own *performance metrics*(i.e. utilities) that are trying to *maximize*, all contributing to the overall goal of the agent. Following are a high-level black box description of each game phase strategy.

3.2.1 Open Phase

In the *open phase* the agent *check* if the odds of losing are greater or equal than its *risk tolerance* range given its current chips count(*Algorithm 1*), otherwise, if the odds are favorable open with a bet size suggested by the *FB_ cAdvisor module*(More in *Section 3.8*).

Algorithm 1 Simplified Open Phase Strategy

Require: $0.35 \leq RT \leq 0.90$ ▷ Risk Tolerance/Min.Playable Odds

$odds \leftarrow$ FB_cAction.describe(self.hand)
if $odds[\text{"}loss\text{"}] \geq RT_0$ and $(c_{bet}/r_{chips} \leq RT_1)$ **then**
  return $AgentAction.CHECK$
end if
if $r_{chips} < min_{pot}$ **then**
  return $AgentAction.CHECK$
end if
$bet \leftarrow$FB_cAdvisor.bet( )
return $(AgentAction.OPEN, bet)$

3.2.2 Call or Raise Phase

During the *Call or Raise Phase*, the agent requests the computation of the *expected value*(*Equation 1*) for each of the allowed actions given their respective probabilities (*Equation 2*). The optimal action is then selected by taking the action that *maximizes* the *expectation*(*Equation 3*). For the first 3 rounds only prior distributions are used, as not enough data has been collected, afterwards the probabilities are augmented with the estimation of predicted highest opponent hand, hence the prior probabilities given the estimation of the *highest opponent hand*(*Equation 4*) (*Algorithm 2*).

$$E[X] = \sum_i x_i P(x_i) \tag{1}$$

$$\begin{gathered} a \in \{Fold, Call, Raise, AllIn\} \\ E_a = P(win) * Reward + P(lose) * Loss \end{gathered} \tag{2}$$

$$a^* = \underset{a}{\operatorname{argmax}}(E_a) \tag{3}$$

$$\begin{aligned} & & a &\in \{Fold, Call, Raise, AllIn\} \\ OH &= \max_{\hat{oh}}, & \text{for } \hat{oh} &\in \{\hat{oh}_1, \hat{oh}_2, \hat{oh}_3, \hat{oh}_4\} \\ E_a &= P(win|OH = \hat{OH}) * Reward + & P(lose&|OH = \hat{OH}) * Loss \end{aligned} \tag{4}$$

Algorithm 2 Simplified Call or Raise Strategy (Expected Value)

Require: $N_{rounds} > 3$

$ohes_{pred}, ohes_{scores} \leftarrow$ OH_ESTIMATOR.PREDICT(memorizer_ref.rounds)

$highest_{ohe} \leftarrow \min(ohes_{pred}.items(), key = lambdax : ohes_{pred}[1])$

$bet_to_place \leftarrow$FB_CADVISOR._CBET(r_chips, min_pot, odds, budget)

$ev_{fold} \leftarrow odds[\text{"}loss\text{"}] * -abs(bet\_so\_far)$

$ev_{call} \leftarrow compute\_cond(odds[\text{"}win\text{"}], highest_{ohe}) * w\_pot + compute\_cond(odds[\text{"}loss\text{"}], highest_{ohe}) * -abs(bet\_so\_far + call\_size)$

$ev_{raise} \leftarrow compute\_cond(odds[\text{"}win\text{"}], highest_{ohe}) * w\_pot + compute\_cond(odds[\text{"}loss\text{"}], highest_{ohe}) * -abs(bet\_so\_far + bet\_to\_place)$

$ev_{allin} \leftarrow compute\_cond(odds[\text{"}win\text{"}], highest_{ohe}) * w\_pot + compute\_cond(odds[\text{"}loss\text{"}], highest_{ohe}) * -abs(bet\_so\_far + remainig\_chips)$

$\max_{ev} \leftarrow max(ev_{fold}, ev_{call}, ev_{raise}, ev_{ev_allin})$ ▷ Maximize EV

return $\begin{cases} 0 & max_{ev} = ev_{fold} \\ 1 & max_{ev} = ev_{allin} \\ 2 & max_{ev} = ev_{call} \\ 3 & max_{ev} = ev_{raise} \end{cases}$

3.2.3 Draw Phase

During the *draw phase* multiple *Monte Carlo* simulations will be run with a depth of 32 to accomodate al 32 possible combinations and the optimal *draw policy* for the current given hand will emerge (More in *Section 3.5*).

Algorithm 3 Simplified Draw Strategy

$draws \leftarrow$ MC_CDRAW.MC_EV_DRAW(hand=hand, M=1000, max_depth=32)[1]

return $(draws)$

3.3 PEAS

The following 5 card poker agent, formerly *Caesar* has been designed to interact with the *client driver (Environment)* (i.e. PokerGame) through the use of two methods: *Caesar.Act()*(actuators), representing the agent actuators that allows to interact with the *Environment* and *Caesar.See()*(Sensors) which allow to *observe, sense and register* events. A summary of the PEAS description is tabulated in *Table 2.* An extract of both method is presented in *Listing 1 and 2.*

Table 2: PEAS Description

	Metrics			
	(P)erformance Measure	(E)nvironment	(A)ctuators	(S)ensors
Description	Expected Value/Hand Strength(Only for Monte-Carlo simulations)	PokerGame/ ClientDriver Events	Caesar.Act	Caesar.See

```python
def see(self, what: See, *args) -> None:
gateway = {
    self.See.NEW_ROUND: self.memorizer.new_round,
    self.See.GAME_OVER: self.memorizer.game_over,
    self.See.PLAYER_CHIPS: self._player_chips_update,
    self.See.ANTE_CHANGED: self.memorizer.update_ante,
    self.See.FORCED_BET: self.memorizer.forced_bet,
    self.See.PLAYER_OPEN: self.memorizer.open,
    self.See.PLAYER_CHECK: self.memorizer.check,
    self.See.PLAYER_RAISE: self.memorizer.raise_to,
    self.See.PLAYER_CALL: self.memorizer.call,
    self.See.PLAYER_FOLD: self.memorizer.fold,
    self.See.PLAYER_ALL_IN: self.memorizer.all_in,
    self.See.PLAYER_DRAW: self.memorizer.draw,
    self.See.PLAYER_HAND: self.memorizer.hand_revealed,
    self.See.ROUND_OVER_UNDISPUTED: self.memorizer.
       round_over,
    self.See.ROUND_OVER_DISPUTED: self.memorizer.
       round_over,
}
if what == self.See.PLAYER_HAND and args[0] == self.name:
    print("**********************")
    print("current hand:", args[1])
if what == self.See.NEW_ROUND and args[0] < 2:
    return
if what == self.See.PLAYER_HAND and args[0] == self.name:
    self.hand = args[1]
return gateway.get(what, "Error: what not found")(*args)
```

Listing 1: *Caesar.see() extract*

```
def act(self, how: Act, *args) -> None:
    print("how", how, args)
    gateway = {
        self.Act.OPEN: self.open,
        self.Act.CALL_OR_RAISE: self.cor,
        self.Act.DRAW: self.draw,
    }
    return gateway.get(how, "Error: how not found")(*args
        )
```

Listing 2: *Caesar.act() extract*

3.4 Fast Hand Evaluator

In order to implement an effective system that would compute *optimal draws* within the pre-imposed tournament's limits (*Table 1*) that would *maximize the expected value given a specific hand distribution*, as briefly discussed in *Section 3.2*, and subsequently in *Section 3.2.2*, has been necessary to implement a 5 card poker hand evaluator that operates in **constant time (O(N))**. The latter *time complexity* has been used as a pre-require during the design phase and being been observed to be an essential requirement for the *MC-CDraw* module to operate *within time constraints* while producing *optimal policies*. In order to meet the latter *system constraints* has been find indispensable to find an affordable, *space-wise*, algorithm that would provide an *unique hands' strength*, given a set of 5 cards, providing *constant lookup time* without the need to arbitrarily and explicitly declare each possible card combination alongside its hand rank; operation that would be unfeasible, given the 2.598.960 possibile unique cards combinations (*Equation 5*). [7, 8, 9]. Since the goal of the evaluator is to provide the hand's strength of any given 5 card set, is possible to reduce the *problem domain space* by grouping together all combinations that yield the same *hand strength* (*i.e. when compared they display the same hand value*) and all those combinations that are proposed in different order but maintain the same *function output*. This problem can be solved by enumerating all the combinations given a valid poker hand rank (*i.e. High card, One Pair, or Two Pair*) [10]. A summary is presented in *Table 3*.

$$^{52}C_5 = \left(\begin{array}{c} 52 \\ 5 \end{array} \right) = \frac{n!}{k!(n-k)!} = 2.59896E + 6 \tag{5}$$

Table 3: Possible 5 Card Hand's combination. Problem domain reduction [10]

Hand Ranking	Unique Combinations	Group-by Combinations
Straight Flush	40	10
Four of a Kind	624	156
Full House	3744	156
Flush	5108	1277
Straight	10.200	10
Three of a Kind	54.912	858
Two Pair	123.552	858
One Pair	1.098.240	2.860
High Card	1.302.540	1.277
Sum	2.598.960	7462

3.4.1 Card Representation

In order to later *maximize the expectation over a set of hands' strengths*, the evaluator has to map for each given set of cards hands $\{C_1, C_2, \cdots, C_5\}$ where $C_1 \cdots C_5 \in ValidCardSet$ to a positive integer such that $1 \leq f(x) \leq 7462$ (card strength expressed in *descending order*). For this purpose has been utilized *prime product encoding*, which, based on the *fundamental theorem of arithmetic*, allows to encode combinations to unique entries when keys are assigned to prime numbers as their product generate unique numbers. This technique has been found to be very common in poker evaluators [7] and widely adopted in the *Computer Science* field. Albeit every prime factor could be used to map the keys, provided a negative scaler when encoding (for large key values), has been decided to use a similar mapping our custom evaluator is inspired from [10] upon which popular evaluator are also based upon such as the widely used library *PHEvalutor*[11]. *Table 4* shows the prime factors used to encode each card.

Table 4: Card Encoding

	Cards												
	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten	Jack	Queen	King	Ace
$\mathbb{P}$	2	3	5	7	11	13	17	19	23	29	31	37	42

Following the *C.Kev's format* [10] each card can be encoded into a *32-bit integer* as depicted in *Figure 2*. Moreover, the field *prime product*, encodes the prime factors from *Table 4* stored in 6 bits (2^6 to store value 42).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
			A	K	Q	J	10	9	8	7	6	5	4	3	2	C	D	H	S	b_3	b_2	b_1	b_0			p_5	p_4	p_3	p_2	p_1	p_0
			Card Value													Suit				B.Value						Prime Factor					

Figure 2: Card bit-representation encoding

An example of the bit representation of the card *Seven of Spade* $\{7s\}$ is shown in *Figure 3*.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
			0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	1	1	1			0	0	1	1	0	1
			Card Value													Suit				B.Value						Prime Factor					

Figure 3: Card bit-representation encoding Example

3.4.2 Hand Evaluation

Provided the bit representation of each card, discussed in the previous section, the main part of the hand evaluation is straightforward and composed of a series of *bit-masking* operations to extract certain part of the encoded card. Despite some parts of the bit representation, might seem *redundant* they are necessary to easy and simplify the *lookup computations*. An evident *trade-off* has to be made to obtain better *time complexity*, where some *space complexity* has to be sacrificed in favors of better *running times*. In this sense, the evaluation complexity has been divided and split in different *sub-problems*; first *Straight flush and Flush* are evaluated, as they are the easiest to compute and they all share by the same suit; this operation can be performed by extracting the suits through bitwise shift masking and then *AND* all cards for checking for equality (*Equation 6*).

$$C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5 \wedge (\texttt{0xf000})_{16} \tag{6}$$

Otherwise we perform a lookup for *Straight flush or high-cards* as shown in *Snippet 3*. Note that after perform a *bitwise OR* between each card and extracting the relevant portion using bit-masking, each lookup output has to be shifted by the relative rank's range. A full list of the partition map to decode and categorize, if necessary, into a specific poker hand is given in *Listing 4* (each value provides lower bound).*Listing 6* depicts the suits bit-masks for each suit for ease of use.

```python
##############
#   EVALUATE   #
##############
# (private)
# card must be encoded
# len(cards) must be 5 for unpack
def _evaluate(self, cards: list) -> any:
    idx = (cards[0] | cards[1] | cards[2] | cards[3] | cards[4]) >> 0x10
    # =====================
    # = S-Flush and Flush ==
    # =====================
    if cards[0] & cards[1] & cards[2] & cards[3] & cards[4] & 0xf000:
        return self.FLUSHES[idx]
    # ========================
    # = Straight and H-Cards ==
    # ========================
    # you might wanna cache this
    s_or_high = self.UNIQUE_RANKS[idx]
    if s_or_high:
        return s_or_high

    # =====================
    # = Exact hash lookup ==
    # =====================
    idx = (cards[0] & 0xff) * (cards[1] & 0xff) * (cards[2] & 0xff) * (cards[3] & 0xff) * (cards[4] & 0xff)
    # print('bitwise', cards[0]&0xff, cards[1]&0xff, cards[2]&0xff, cards[3]&0xff, cards[4]&0xff, idx)
    # print('debug:', idx, self.exact_lookup(idx), len(self.HASH_VALS))
    return self.HASH_VALS[self.exact_lookup(idx)]
```

Listing 3: *Fast Hand Evaluator: _evaluate() extract*

```python
# ============================
# = Hand ranks Paritions Map ==
# ============================
# NAME -> Value
# EQ.Class
STRAIGHT_FLUSH_RANGE = 0
```

```
FOUR_OF_A_KIND_RANGE = 10
FULL_HOUSE_RANGE = 166
FLUSH_RANGE = 322
STRAIGHT_RANGE = 1599
THREE_OF_A_KIND_RANGE= 1609
TWO_PAIR_RANGE = 2467
ONE_PAIR_RANGE = 3325
HIGH_CARD_RANGE = 6185
```

Listing 4: *Fast Hand Evaluator: Hand Ranks Partition Map*

```
# ===============
# = Hand ranks ==
# ===============
# Value -> Name
# EQ.Class
HAND_RANKS = {
    0: "Straight flush",
    10: "Four of a Kind",
    166: "Full House",
    322: "Flush",
    1599: "Straight",
    1609: "Three of a Kind",
    2467: "Two Pairs",
    6185: "High Cards"
}
```

Listing 5: *Fast Hand Evaluator: extract*

```
# ===============
# = Card Suits ==
# ===============
# NAME -> VAlUE
CLUB = 0x8000
DIAMOND = 0x4000
HEART = 0x2000
SPADE = 0x1000
```

Listing 6: *Fast Hand Evaluator: extract*

Lastly if the hand is not a *Straight flush, flush, straight, or high-card* we compute another *lookup* but using as a key the unique product of all the prime factors of each card. Therefore, we obtain the prime factor for each card, by extracting the 6 bits, then multiply the prime factors together (*Equation 7*).

$$\prod_{i=1}^{5} C_i \wedge (\texttt{0xff})_{16} \tag{7}$$

For a more efficient approach (*space-wise*) is also possible to compute an *exact hash* rather than using the built-in hash function, in order to optimize storing space and lower sparsity by storing each non-subsequent entries in continues memory entries. This has been found to be a popular approach adopted by common libraries such as *PHEvaluator* [11]. We adopted a similar approach with a truncated hash in order to work with our variation of 5 card poker [9] and an extract is provided in *Listing 7*.

```python
# ======================
# = Exact hash lookup ==
# ======================
# smart impl inspired by S.Paul
# this is blazing fast
def exact_lookup(self, value):
    i = j = k = 0
    value += 0xe91aaa35
    value ^= value >> 16
    value += value << 8
    value ^= value >> 4
    j  = (value >> 8) & 0x1ff
    i  = (value + (value << 2)) >> 19
    return (i ^ self.HASH_ADJ[j])
```

Listing 7: *Fast Hand Evaluator: exact_lookup() extract*[9]

3.5 Monte Carlo Draw Estimation

In order to *act optimally* during the *draw phase*, and therefore, ultimately, contributing to *maximize the overall agent's utility* (More in *Section 3.2.2*), the agent has to draw the cards that *maximize* its odds of winning. Moreover, it has to discards the cards that when replaced have the highest probability of yielding the **highest hand's strength** given the current set of cards. Given a state S_i that contains the current set of poker cards in your hand $S_i = \{C_1, C_2, C_3, C_4, C_5\}$, we are interested in computing the right policy that maximizes our utilities. As briefly discussed in *Section 2*, during the literature review phase have been found different statistical techniques to approximate the expectation over given distribution and hereafter produce the optimal *policy* with *low computation overhead* through the use of probabilistic inference, combinatorics and partial formula [6]. Hence the main focus of this course is on *Artificial intelligence* has been decided to utilize methodologies in our curriculum and common in the field of AI. Due to the large search space (*Equation 5*) *Monte Carlo* with repeated random sampling has been implemented to simulate and draw inference with the aim of computing the optimal draw policy.

$$\begin{gathered} X_n = \frac{S_n}{n} = \frac{1}{n}\sum_{i=1}^{n} Y_i \\ P\left(\omega \in \Omega : \lim_{n\to\infty} [X_n(\omega)] = \mu\right) = 1 \\ X_n \overset{\text{a.s.}}{\to} \mu \end{gathered} \tag{8}$$

3.5.1 Monte Carlo Discard Simulation

In order to *maximize our performance metric*, the **hand's strengh**, evaluated using the *hand evaluator* discussed in *Section 3.4.2*, we need to simulate all possible 32 discards combinations *Equation 9*. To *minimize* execution times, we pre-compute the discards combinations, as shown in *Listing 8* where "1" denotes the replacement of the card at the i_{th} place.

$$N_{discards} = \sum_{i=1}^{5} {}^{5}C_i = 32 \tag{9}$$

```
# ======================
# = Discards scenarios =
# ======================
# for speed precomputed
# all scenarios
# Discard   0, 1, 2,  3, 4, 5
```

```
# C(n,r)     1,  5,  10,10,  5,  1
discards = np.array(
    [
        # discard 0
        [0, 0, 0, 0, 0],
        # discard 1 (5)
        [1, 0, 0, 0, 0],
        [0, 1, 0, 0, 0],
        [0, 0, 1, 0, 0],
        [0, 0, 0, 1, 0],
        [0, 0, 0, 0, 1],
        # discard 2 (10)
        [1, 1, 0, 0, 0],
        [1, 0, 1, 0, 0],
        [1, 0, 0, 1, 0],
        [1, 0, 0, 0, 1],
        [0, 1, 1, 0, 0],
        [0, 1, 0, 1, 0],
        [0, 1, 0, 0, 1],
        [0, 0, 1, 1, 0],
        [0, 0, 1, 0, 1],
        [0, 0, 0, 1, 1],
        # discard 3 (10)
        [1, 1, 1, 0, 0],
        [1, 1, 0, 1, 0],
        [1, 1, 0, 0, 1],
        [1, 0, 0, 1, 1],
        [1, 0, 1, 0, 1],
        [1, 0, 1, 1, 0],
        [0, 1, 1, 1, 0],
        [0, 1, 0, 1, 1],
        [0, 1, 1, 0, 1],
        [0, 0, 1, 1, 1],
        # discard 4 (5)
        [1, 1, 1, 1, 0],
        [1, 1, 1, 0, 1],
        [1, 1, 0, 1, 1],
        [1, 0, 1, 1, 1],
        [0, 1, 1, 1, 1],
        # discard 5 (1)
        [1, 1, 1, 1, 1],
    ]
)
```

Listing 8: *Pre-computed discard combinations*

Following the *Law of Large Numbers (LLNN) (Equation 8)*, where $X_i \cdots X_n$ are independent random variables, we can obtain an *unbiase MC estimation* of a parameter by running the averages for that parameter, given the sampling is large enough that they will eventually converge (*Equation 10*). Therefore, random sampling from normal distribution is possible to estimate the coefficient β, $\hat{\beta}$, then by taking the mean of all the samples we can approximate the value of the *unbiased estimate* β (*Equation 11*).

$$\hat{\theta}_{MC} = \frac{1}{N}\sum_{i=1}^{N}\hat{\theta} \tag{10}$$

$$\begin{aligned} Y &= \alpha + \beta X_i \\ \hat{\beta}_{MC} &= \frac{1}{N}\sum_{i=1}^{N}\hat{\beta} \end{aligned} \tag{11}$$

The *Listing 9* shows an extract of the *Monte Carlo simulation*, and it runs M simulation with a depth *max_ depth* (*number of discards combinations to simulate*, default is 32). Instead of using a fixed number of simulations is also possible to use a time-based constrained of simulate until convergence is less then a predefined value ε. However, since the algorithm is not after an accurate estimation of each probability distribution, but rather for a *policy*, has been observed that in most scenarios less than 1000 simulations are sufficient to obtain an *optimal policy* as the *strategy converges faster* than the *true* expectence of the estimator. A pseudocode of the function parameters has been attached in *Algorithm 4*.

Algorithm 4 Input and Ouputs of the Montecarlo

Input

hand	Current poker hand
max_depth	Monte Carlo Simulation Depth / draws to estimate
M	Simulation count upper bound

Output

(d_idx, d_cards)	Tuple containing the index of cards to replace and the cards to replace

```python
# ====================================
# = Monte Carlo LRR with R-Sampling =
# ====================================
# add hybrid selection to reduce
# search space
for si in range(M):
    # ========================
    # = Compute replacements =
    # ========================
    c_hand_sample = [
        x if x not in ["10h", "10d", "10s", "10c"] else "
           T" + x[2]
        for x in c_hand
    ]
    # ============================
    # = No discards Special Case =
    # ============================
    # TODO: update this with analytical statistics
    #  no need ot waste cycles here
    if c_ndiscards == 0:
        # compute directly current hand
        mc_samples[idis][si] = (c_hand_sample, idis)
        mc_betas[idis][si] = evaluate_cards(*
           c_hand_sample)
        continue
    c_deck = MC_CDraw._build_new_deck(exclude=c_hand)
    rng = np.random.default_rng()
    sampled = rng.choice(c_deck, c_ndiscards, replace=
       False)
    c_hand_sample.extend(sampled)
    # print("evaluating: ", c_hand_sample)
    # ================================
    # = Monte Carlo Samples and Beta =
    # ================================
    mc_samples[idis][si] = (c_hand_sample, idis)
    mc_betas[idis][si] = evaluate_cards(*c_hand_sample)
# ==============
# = Compute B^  =
# ==============
mc_beta_hats[idis] = np.mean(mc_betas[idis])
```

Listing 9: *Simplified Monte Carlo Simulatition LRR with R-Sampling*

3.6 Memorization

The following poker AI agent (*Caesar*) has a built-in memorization module that allows to record and store all the events passed from the *driver client*. The *Memorizer* module has been designed as a *hot-pluggable* dependency and can be disconnected from the core components of *Caesar*; if turned off, it automatically disables the *OH_Estimator* module (More in *Section 3.7*), as the estimation of the opponents' poker hands requires historical data processing and depends on this module. It provides both *data managmenet capabilities* as well as acting as a basic *Data Access Layer (DAL)* for the logic components of the poker agent. Optionally allows to store the full-log dump as a binary format (*serialized object*) through the use of the library *Pickle* for debug purposes providing *diagnostic capabilities* [12]. Since all debug information are organized and stored in rounds, through the use of a small script (*Listing 10*) is possible to replay the entire match round-by-round. A list of all tracked events provided by the *client driver API* is tabulated in *Table 5*

Table 5: Process and Stored Events

Events	Tracked	Available for Debug/Diagnostic
New Round	Yes	Yes
Pot Tracking	Yes	Yes
Call Tracking	Yes	Yes
Players' Chips' Tracking	Yes	Yes
Forced Bet Tracking	Yes	Yes
Player Open	Yes	Yes
Player Check	Yes	Yes
Player Raise	Yes	Yes
Player Call	Yes	Yes
Player Folds	Yes	Yes
Player All-in	Yes	Yes
Player Draw	Yes	Yes
Player Round-Over	Yes	Yes
Player Game-Over	Yes	Yes
Player Round-Dispute Tracking	Yes	Yes
Player Hand Reveled (showdown)	Yes	Yes

```
rounds = pd.read_pickle(r'dumps/dump_1641118662.044854.
    log')

for round in rounds:
    print(f'ROUND {round.number}:')
```

```
    for opponent in round.opponents:
        print(f'{opponent}:', round.opponents[opponent].
            __dict__)
    print(f'{"-"*100}')
```

Listing 10: *Sample code for match replay*

3.7 Opponents' Hand Estimation

The agent in order to be able to act *optimally*, by providing means of estimation to the *FB_ cAdvisor module*, has to, including but not limited to, estimate the opponents' poker hand strength. This operation, as brifely discuessed in *Section 3.2.2*, is performed by the *OH_Estimator* class which, by utilizing the data provided by the *Memorizer module* (More in *Section 3.6*), aims to infer the strength of each opponent's poker hand. The *hand's strength* is predicted by *curve fitting* a *linear regression model* using the historical data available through the *Memorizer*. The x_j points for each opponents, given his past rounds, are computed using the *Equation 12*; the difference between the *opponent's bet* and the *relative minimum bet to play* is computed and divided by the amount of chips available to the player in that moment; the value can also be scaled in percentile by multiplying it by 100. The Y_j points are instead computed by taking the opponents cards, after every showdown, and computing their *hand's strength* through the *fast hand evaluator* as discussed in *Section 3.4.2* ; each value will therefore be in the range of $1 \geq Y_j \leq 7462$ and represent the target of our predictions. *Algorithm 5 shows the pseudocode of the model fitting using the scikit library*[13].

Algorithm 5 Opponents' Hand Strength Regression Estimation

Require: $N \geq 3$
Ensure: $len(X) \geq 3$
procedure $\_\text{FIT}(N = 3)$
    $X, y \leftarrow assemble\_set(memorizer.data)$
    for $opponent \leftarrow opponents$ **do**
        $model[opponent] \leftarrow LinearRegression().fit(X[opponent], y[opponent])$
    end for
end procedure

$$x_j = \frac{\frac{\Delta bet_{i-1}}{min_{bet_{i-1}}}}{(chips\_player_k)_j} \cdot 100 \qquad \begin{aligned} &\text{for } i = 1, 2, \ldots, N_{actions} \\ &\text{for } j = 1, 2, \ldots, N_{rounds} \\ &\text{for } k = 1, 2, \ldots, N_{players} \end{aligned} \tag{12}$$

3.7.1 Linear Regression, Polynomial Regression, Multivariate Regression

During the testing phase *linear* and *polinomial* regression model have been tested against a *simple reflex agent* with no significant difference between the *linear* and *polinomial* regression; due to time constraints, and to avoid overfitting, has been chosen the simpler *linear model.* The possibility of utilizing *multivariate regression* by augmenting the feature set with also the number of discards that lead to the showdown has also been investigated and compared against the *linear* model with no significant improvement; this can be reconducted to a limitation in our testing setup (i.e. the limited availability to adversarial opponents), or simply due to the fact that a **rational opponent agent** tries *maximize* its utilities and therefore, due to the simple rule set of the game, there are only a limited number of ways to accomplish this; more generally, the bet size, the feature under analysis, when scaled to players' chips is linearly correlated with its hand's strenght when the *agent is acting rationally.*

3.8 Variable budget allocation

The agent's chip's asset, at any point in time during the game, is subdivided into *reserved* and *allocable.* The former is, as the name implies, reserved and cannot be bet, and accounts for *nonoptimal* plays while the latter is dedicated for the agent's to bet upon and is *stretched* and *shrinked* during the game according to the game state. *Figure 3.8* and *Figure 3.8* depicts the budget allocation at different *agent's remaing chips state.*

Figure 4: Budget allocation for $R_{chips} \leq 120$

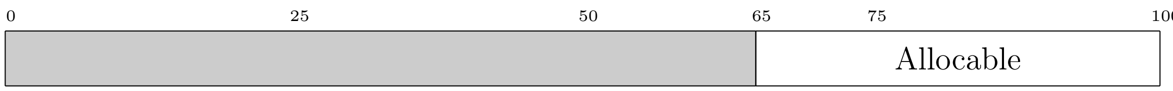
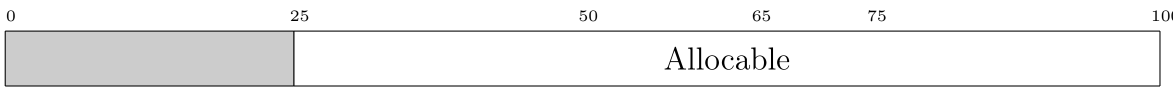


Figure 5: Budget allocation for $R_{chips} \geq 200$



3.9 Proportional Bet Size Mapping

The agent's bet size at each round is computed proportial to the *predicted expected probability* of winning in respect to the available *"allocable"* budget as discussed in *Section 3.8*. *Listing 11* shows relevant portion of the code for computing the mapping between odds and bet size (chips).

```
odds = FB_cAction.describe(self.hand)
rdiff = r_chips - min_raise
rbudget = budget_allocable * rdiff if (b_allocable * rdiff) > min_raise else rdiff
bet = int(np.interp(odds["win"], [0, 1], [0, rbudget]))
```

Listing 11: *Bet size mapping*

4 Results

4.1 Random and Reflex Rational Adversarial Agent

During the pre-tournament testing phase, *Caesar*, the developed 5 card poker AI agent has been tested, and its parameterized values being evaluated and optimed to *maximize* their *performance metrics* and *expected utilities*. The agent has been tested against the provided *random agent* and a *simple reflex agent* with basic prior probabilities in a 4 player game (*Caesar* vs. 3 adversarial agents of the same type) in a *docker container* in 10 batches of 5 games (to promote card variance due to different seeds during game state creation). The *game configuration* have been kept constant across all the testing and are listed in *Table 6*.

Table 6: AI PokerServer Testing Settings

	Parameter					
	N.Players	Initial Chips Count	Initial Ante	Ante Raise every	Client Response Time [ms]	Display Sleep Time [ms]
Value	4	200	10	10	15.000	0

During the preliminary testing phase, demonstrated good performances against the *Random agent*, losing only 2 games out of 50 with a *win-rate* of 96%. Running against a *Reflex Agent*, *Caesar*, won only 46 times out of 50 with a *win-rate* of 92%. A comparison is plot in *Figure 6*.

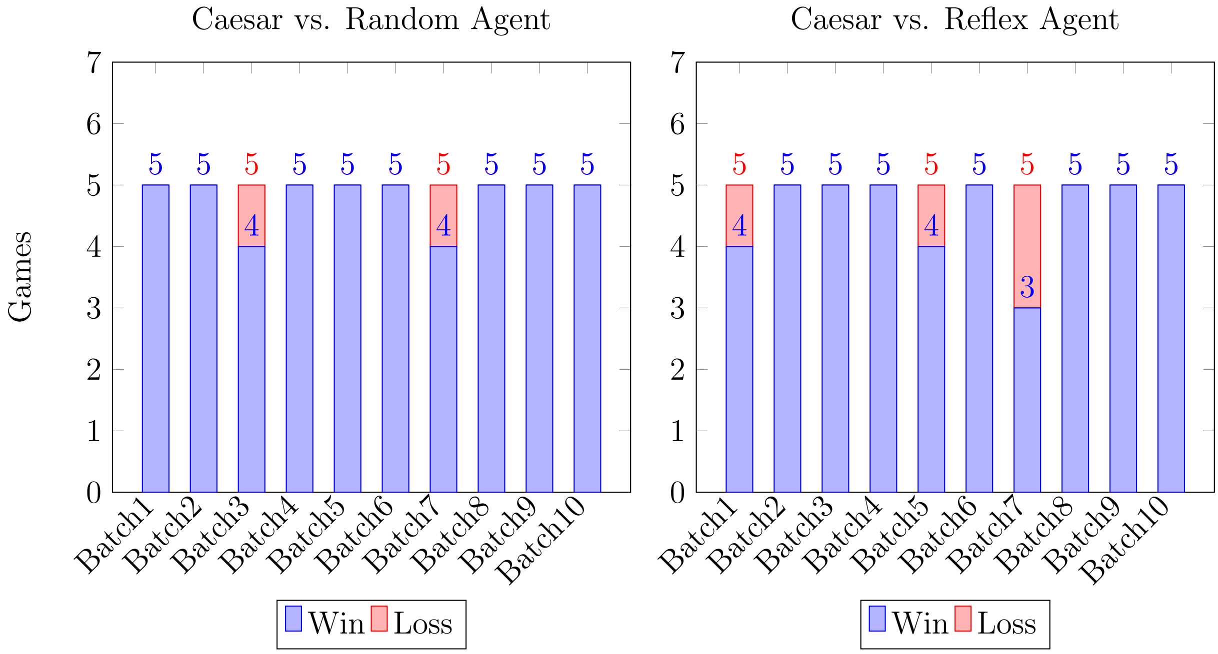


Figure 6: Caesar vs. Different Adversarial Agents Testing

By analyzing the *diagnostic* providedy by the *Memorizer* class, has been possible to unfold an issue in the *strategy* adopted by *Caesar* that would break the approximate optimality under certain cirmustances due to an *non-optimal value* of the paramter that characterizes the *risk tolerance*; this caused the agent, with too low value of RT_0, the agent could encounter *starvation* and being induced into a series of *fold* that would, most of the times, result in the agent losing. This phenomenon was caused mainly, but not limited to, a set of bad cards with associated low probabilities of success; in this context the expected utility of folding outweighted the odds of continue playing, even with a 4 or 5 card draw. This issue has been fixed by *optimizing the RT_0 parameter* **(i.e. increasing the risk tolerance)** when running low on chips ($R_{chips} \leq 120$). Moreover, is straightforward that the agent should take more risks with a low chip's asset, hoping for comeback under optimal play, rather than folding and being eliminated from subsequent rounds due to the high cost of the enforced ante. A before and after comparison of the *optimized risk tolerance* is plot in *Figure 7* where we have an incresae in *win-rate* from 92% to 98% with only one loss over a span of 50 games tested.

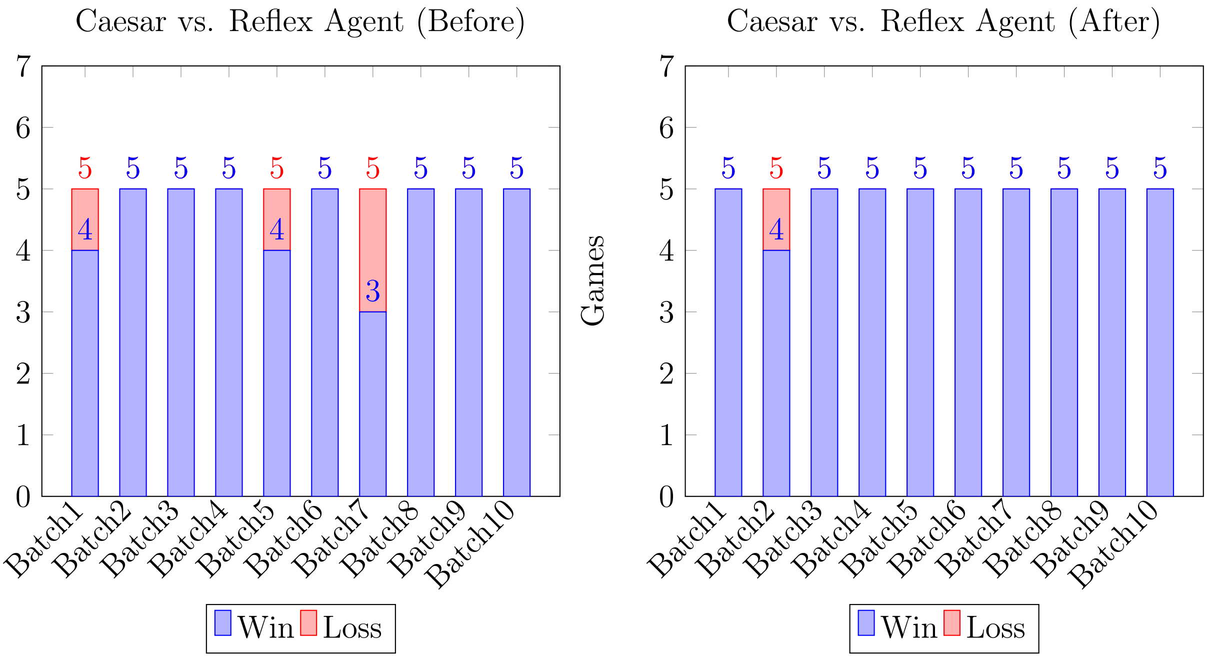


Figure 7: Caesar vs. Reflex Agent - Before and After Fold Starvation Fix

4.2 Tournament Results

The tournament results showed similar results when compared to the tests conducted against a *reflex adversarial agent* showing, due to the game rules and configutation, that the analyzed game has not abundant room for exploitation when trying to *maximize the expected utilities* and especially on longer game session, putting in place techniques (such as bluffing) that go against the *statistical odds* is detrimental. In the **pre-tournament** *Caesar* won 10 matches out of 14 (one matches another player joined twice and the game started without our agent) of which 2 out of 14 the *Monte Carlo* estimation ran out of time, disqualifying *Caesar*. This showed a sampled *win-rate* of 83.3%.

During the **final tournament** *Caesar* showed similar performances, gaining a significant lead in the first 6 matches winning 4 matches out of 6. Consequently, the professor increased the *client response time* of the game server, therefore for the sake of experimentation, the increased *Monte Carlo Simulations* have been increased from 800 to 1600.This combined with the longer rounds caused the *docker container* to run out of memory under specific circumstances; this phenomenon become more apparent with the increase of the round's count as the *Memorizer* heap size would grow. At every subsequent game the *simulation count* has been decreased from 1600 back to 800 in steps of 200 per game. With values greater than 1200 when combined with the dump of the *Memorizer* would make the docker container run out of memory, while for values between 1200 and 800 the docker container would not run ouf of memory but would not reply to the game server within the allowed time period. By inspecting the diagnostic's data, after the tournament, has been observed that this behaviour has been due to *Python* not respecting the *Docker memory constraints* ending up with the system resourse manager killing the process itself [14]. For futuher development, there exist *easy-to-implement* workarounds such as setting specific docker memory flags and pre-allocating the memory.

5 Conclusion

The following paper discussed the implementation of the 5 card poker AI agent, formerly *Caesar*, carried out as a project assignment for the course *DT8042 Artificial Intelligence* at *Halmstad University.* The agent showed promising results during both the internal pre-tournament test phase as well as the two official tournaments, of which **Caesar won, scoring first place**. Through the process, become apparent, as underlined by the proposed material during *literature review* as discussed in *Section 2*, that due to the game configuration, *5 card poker* is better suited for statistical approches [6]. Utilizing other methodologies within the AI field, beside the popular approaches including but not limited to, *Neural Network*, *Deep Neural Network*, and techniques as *Counterfactual Regret Minimization*, such as the one outlined in this paper (and present within the course curriculum), has been shown to be effective but coming with a **high computational overhead**.

References

[1] Suraiya Jabin. Poker hand classification. *Proceeding - IEEE International Conference on Computing, Communication and Automation, ICCCA 2016*, pages 269–273, 1 2017. doi: 10.1109/CCAA.2016.7813761.

[2] Garrett Nicolai and Robert J. Hilderman. No-limit texas hold'em poker agents created with evolutionary neural networks. *CIG2009 - 2009 IEEE Symposium on Computational Intelligence and Games*, pages 125–131, 2009. doi: 10.1109/CIG.2009.5286485.

[3] J Zhang, K Li, B Zhang, M Xu, and C Wang. Parallel counterfactual regret minimization in crowdsourcing imperfect-information expanded game. pages 1444–1451, 2021. doi: 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00195.

[4] Weiming Liu, Bin Li, and Julian Togelius. Model-free neural counterfactual regret minimization with bootstrap learning. *IEEE Transactions on Games*, 2022. ISSN 24751510. doi: 10.1109/TG.2022.3158649.

[5] Bill Chen and Jerrod. Ankenman. *The mathematics of poker*. ConJelCo LLC, 2006. ISBN 9781886070257.

[6] Catalin Barboianu. *DRAW POKER ODDS: The Mathematics of Classical Poker*. 6 2007. ISBN 9738752051.

[7] Zhenzhen Hu, Jing Chen, Wanpeng Zhang, Shaofei Chen, William Yuan, Junren Luo, Jiahui Xu, and Xiang Ji. Odds estimating with opponent hand belief for texas hold'em poker agents. 7 2021.

[8] C. Kev. Cactus kev's poker hand evaluator, 2000. URL `http://suffe.cool/poker/evaluator.html`.

[9] Senzee Paul. Senzee 5 - paul senzee: Some perfect hash, 6 2006. URL `http://senzee.blogspot.com/2006/06/some-perfect-hash.html`.

[10] C. Kev. Enumerating five-card poker hands, 2006. URL `http://suffe.cool/poker/7462.html`.

[11] L. Henry. Pokerhandevaluator/cpp at master · henryrlee/pokerhandevaluator · github. URL `https://github.com/HenryRLee/PokerHandEvaluator/tree/master/cpp`.

[12] Python Software Foundation. pickle — python object serialization — python 3.10.5 documentation. URL `https://docs.python.org/3/library/pickle.html`.

[13] scikit-learn developers. sklearn.linear_model.linearregression — scikit-learn 1.1.1 documentation. URL https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html.

[14] Carlos Becker. Making python respect docker memory limits · carlos becker, 2021. URL https://carlosbecker.com/posts/python-docker-limits/.

A Appendix

A.1 Source Code

The full source code is also available on *GitHub* at the following link: `https://github.com/espressoshock/caesar-5c-poker-agent-ai`