Evaluation of AI Strategies for Blackjack

Team 5

111950020 李欣穎　109550072 何嘉娓　109550202 白詩愷　110550092 林啟堯

The Rules of Blackjack

Objective of the Game

Each participant tries to beat the dealer by getting a count as close to 21 as possible, without going over 21.

Card Values/scoring

Each individual player if their ace is worth 1 or 11. Face cards are 10 and any other card is its original value.

The Play

Players hit (take a card) or stand (keep cards) to get near 21. Going over 21 is a bust, losing the bet. A soft hand has an ace, which can be 1 or 11; if hitting would bust with an 11, it becomes 1 instead.

Problem Statement

Blackjack is one of the most classic card games in the casino. Since this game has remained popular in casinos for such a long time, it is evident that the dealer usually has a higher winning probability than the player. This observation sparks our curiosity: **Are there any strategies that can help players perform better in this game?**

Introduction

We compare the win rates of agents using **random selection, Basic Strategy, Expectimax, Random Forest Classifier, NeuroEvolution of Augmenting Topologies (NEAT)**. We aim to answer:

1. **How does the player's win rate vary under different agents?**
2. **Does the different number of decks affect the win rate?**
3. **Are the casino's rules inherently designed to favor the dealer?**

Table of Contents

01

Review Model

Justin Bodnar Model

Goal of the Model

The goal of Bodnar's model is to train network that mimics a **basic blackjack strategy** that has:

- Input:
  Information regarding the current game state (e.g., player's hand value)
- Output:
  Decision to *hit* or *stay*.

Model Architecture

The model proposed uses Monte Carlo simulations of Blackjack games to create training data.

The model runs on a neural network:

- Level 1: Player and hand value
- Level 2: Player hand + dealer's face-up card
- Level 3: All cards seen

Mason Lee Model

Goal of the Model

The goal of this project is to train an AI agent to play Blackjack by evolving its decision-making strategy using **NEAT** (NeuroEvolution of Augmenting Topologies).

- Input:
  Current game state (player's hand, dealer's visible card, usable ace info, etc.)
- Output:
  AI decides whether to hit, stand, or double

Model Architecture

Evolve neural networks based on fitness (win rate, performance):

1. Initialize game
2. Player decision making
3. Dealer logic
4. Determine result

02

Related Work

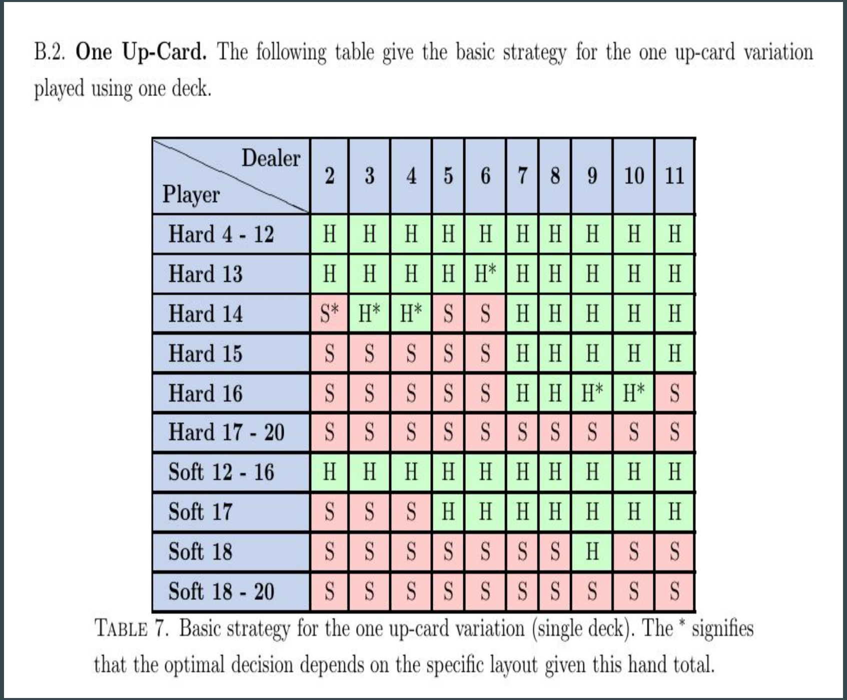
Paper

Paper :

https://arxiv.org/abs/2407.08755v1

We read a paper of strategy of Blackjack.

It provides basic strategy rules for Blackjack, used as reference for Basic Strategy Agent logic.



B.2. **One Up-Card.** The following table give the basic strategy for the one up-card variation played using one deck.

Player \ Dealer	2	3	4	5	6	7	8	9	10	11
Hard 4 - 12	H	H	H	H	H	H	H	H	H	H
Hard 13	H	H	H	H	H*	H	H	H	H	H
Hard 14	S*	H*	H*	S	S	H	H	H	H	H
Hard 15	S	S	S	S	S	H	H	H	H	H
Hard 16	S	S	S	S	S	H	H	H*	H*	S
Hard 17 - 20	S	S	S	S	S	S	S	S	S	S
Soft 12 - 16	H	H	H	H	H	H	H	H	H	H
Soft 17	S	S	S	H	H	H	H	H	H	H
Soft 18	S	S	S	S	S	S	S	H	S	S
Soft 18 - 20	S	S	S	S	S	S	S	S	S	S

TABLE 7. Basic strategy for the one up-card variation (single deck). The * signifies that the optimal decision depends on the specific layout given this hand total.

03

Dataset and Platform

Dataset

Kaggle Blackjack Dataset:

https://www.kaggle.com/datasets/dennisho/blackjack-hands?resource=download

size of the dataset: 50000000(rows)X12(columns)

We only use the first 500000 rows in our work

Vector Playing Cards Dataset:

https://code.google.com/archive/p/vector-playing-cards/

size of the dataset: 52 PNG card images.

Column of Dataset

Dataset Structure

shoe_id	Unique ID for each shoe (deck batch)
cards_remaining	Number of cards left in the shoe
dealer_up	Dealer's face-up card (1=A, 10=10/J/Q/K)
initial_hand	Player's initial hand (2 cards)
dealer_final	Dealer's final hand after the game
dealer_final_value	Final hand value for the dealer
player_final	Player's final hand(s) (supports multiple hands after splitting)
player_final_value	Final value(s) for each player hand
actions_taken	Sequence of actions (H=Hit, S=Stand, D=Double, P=Split)
run_count	Running count (Hi-Lo system)
true_count	True count (normalized by decks remaining)
win	Game outcome (1=win, 0=draw, -1=loss, 1.5=Blackjack win, 2=splitting win)

Data Process

Parsing Complex Columns:

Columns such as **initial_hand** and **actions_taken** are stored as strings representing lists.

These strings are converted to Python objects using **ast.literal_eval()**

For **actions_taken**, only the first sublist is used for training.

Feature Extraction:

For each player action, the following features are extracted:

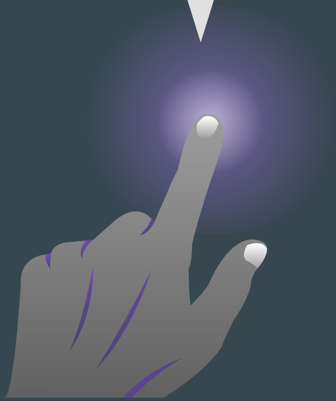
Feature	Description
player_sum	Sum of player's current hand points
is_soft	Whether the hand is a soft hand (contains 11)
num_cards	Number of cards in the player's current hand
dealer_up	Dealer's face-up card
action	Encoded action: 0=Hit, 1=Stand
win	Binary label: 1=Win, 0=Loss

04

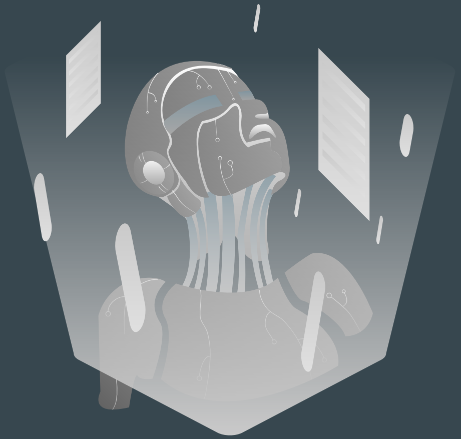
Baseline

AI Strategies

Random and Basic Strategy



Random Forest Classifier

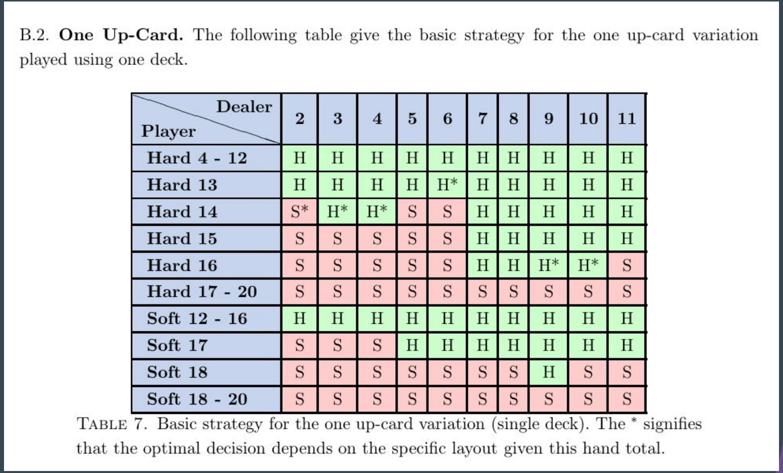


NeuroEvolution of Augmenting Topologies (NEAT)

Random and Basic Strategy

The Random Agent makes **decisions purely based on randomness**. At each decision point, the agent randomly chooses between two actions: **Hit/Stand**

The Basic Strategy Agent strictly follows the static Blackjack **decision table** on the right.



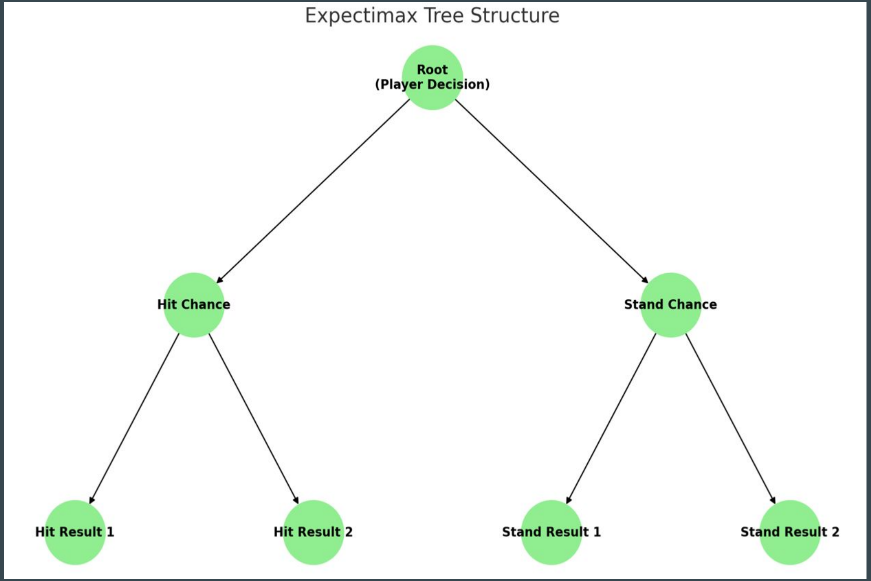
B.2. **One Up-Card.** The following table give the basic strategy for the one up-card variation played using one deck.

Dealer / Player	2	3	4	5	6	7	8	9	10	11
Hard 4 - 12	H	H	H	H	H	H	H	H	H	H
Hard 13	H	H	H	H	H*	H	H	H	H	H
Hard 14	S*	H*	H*	S	S	H	H	H	H	H
Hard 15	S	S	S	S	S	H	H	H	H	H
Hard 16	S	S	S	S	S	H	H	H*	H*	S
Hard 17 - 20	S	S	S	S	S	S	S	S	S	S
Soft 12 - 16	H	H	H	H	H	H	H	H	H	H
Soft 17	S	S	S	H	H	H	H	H	H	H
Soft 18	S	S	S	S	S	S	S	H	S	S
Soft 18 - 20	S	S	S	S	S	S	S	S	S	S

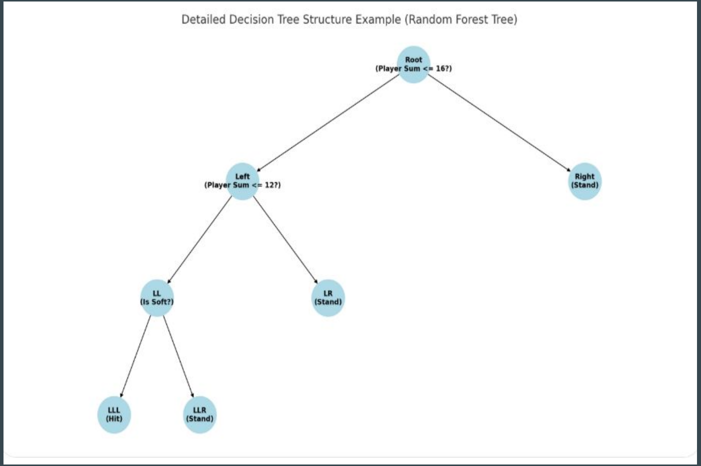
TABLE 7. Basic strategy for the one up-card variation (single deck). The * signifies that the optimal decision depends on the specific layout given this hand total.

Expectimax

1. **Build the Game Tree**
   - Max nodes for player's choices (Hit/Stand)
   - Chance nodes for random events (card draws)
2. **Evaluate Leaf Nodes**
   - Calculate terminal state values (win, lose, draw scores).
3. **Backpropagate Values**
   - Max nodes: Choose the **maximum value**.
   - Chance nodes: Compute the **expected value** (weighted average of outcomes).
4. **Choose Optimal Action**
   - At the root, pick the action with the highest expected value.



Random Forest Classifier



1. Bootstrap Sampling

   Randomly sample the training data with replacement to create a new dataset for each tree.

2. Feature Subset Selection

   At each node split, randomly select a subset of features.

3. Tree Construction

   Grow each decision tree to the maximum depth or until a stopping condition is met

4. Ensemble Prediction:

   For a new input, get predictions from all trees and use majority voting for classification.

5. Final Output:

   Return the majority class (classification) or average prediction (regression).

NeuroEvolution of Augmenting Topologies (NEAT)

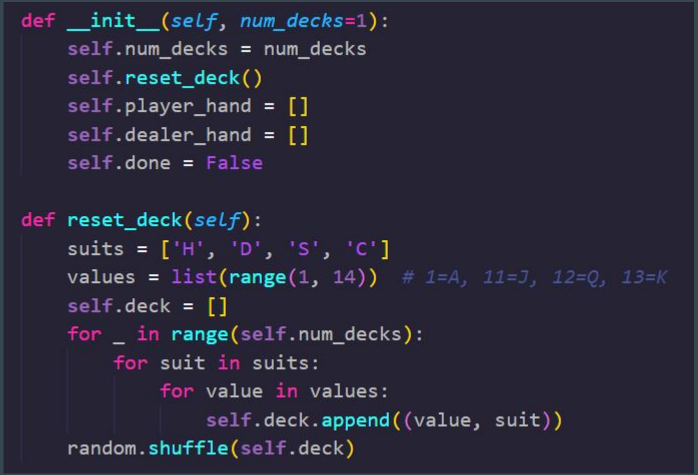
Step	**Action**
Initialize	Create random, simple networks
Fitness Eval	Run network → get fitness score
Speciation	Group by genetic distance (innovation #)
Fitness Sharing	Divide fitness within species
Select Parents	Based on adjusted fitness
Crossover	Align by innovation #, combine genes
Mutation	Adjust weights, add nodes/connections
New Generation	Offspring + elite genomes

05

Main Approach

Environment

- **Deck Generation** and **Shuffling** (reset_deck):
  Generates multiple decks of standard 52-card playing cards (1=A, 11=J, 12=Q, 13=K) and shuffles them randomly.
- **Game Initialization** (reset):
  At the start of each round, deals two cards to the player and one upcard to the dealer, and resets the game state.



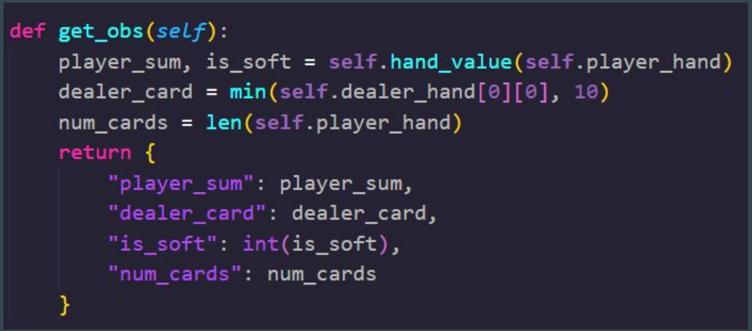
```
def __init__(self, num_decks=1):
    self.num_decks = num_decks
    self.reset_deck()
    self.player_hand = []
    self.dealer_hand = []
    self.done = False

def reset_deck(self):
    suits = ['H', 'D', 'S', 'C']
    values = list(range(1, 14))  # 1=A, 11=J, 12=Q, 13=K
    self.deck = []
    for _ in range(self.num_decks):
        for suit in suits:
            for value in values:
                self.deck.append((value, suit))
    random.shuffle(self.deck)
```

Environment

- **Game State Representation (get_obs):**

  Outputs player state features at each step, including:

  - Player total sum (player_sum)
  - Soft hand status (is_soft)
  - Dealer upcard (dealer_card)
  - Number of player cards (num_cards)



```
def get_obs(self):
    player_sum, is_soft = self.hand_value(self.player_hand)
    dealer_card = min(self.dealer_hand[0][0], 10)
    num_cards = len(self.player_hand)
    return {
        "player_sum": player_sum,
        "dealer_card": dealer_card,
        "is_soft": int(is_soft),
        "num_cards": num_cards
    }
```

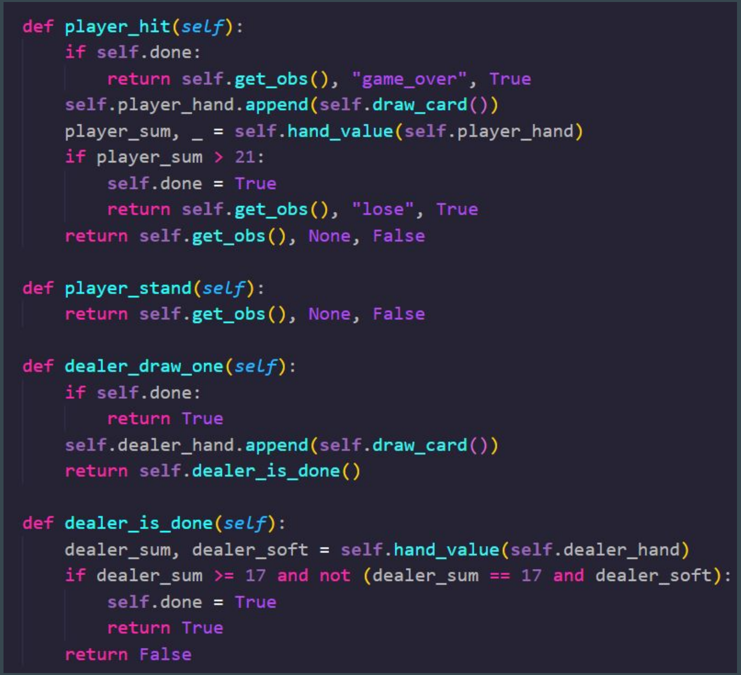
Environment

- **Player Action Handling**

  Players can choose "hit" or "stand"; the system updates the game state accordingly:

  - If "hit", draw a card and update the state.
  - If "stand", the dealer plays automatically following standard rules (stop at 17 or higher unless it's a soft 17).
- **Game Result Evaluation (get_game_result)**

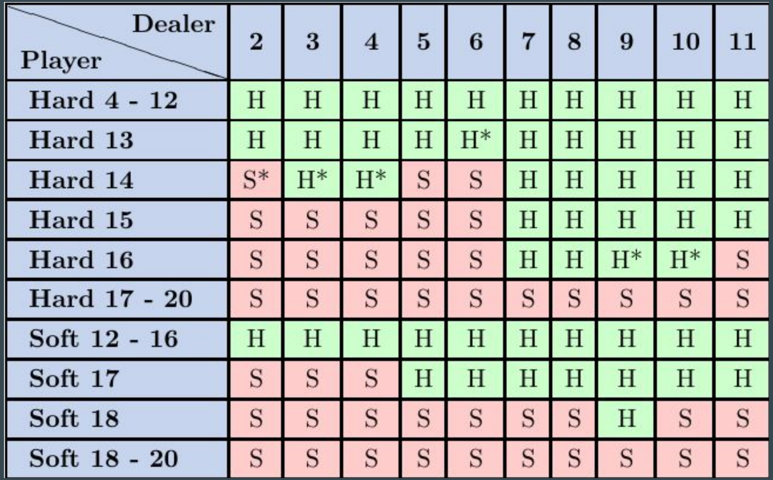
  Compares player and dealer final hands to determine the outcome: win / lose / draw.



```python
def player_hit(self):
    if self.done:
        return self.get_obs(), "game_over", True
    self.player_hand.append(self.draw_card())
    player_sum, _ = self.hand_value(self.player_hand)
    if player_sum > 21:
        self.done = True
        return self.get_obs(), "lose", True
    return self.get_obs(), None, False

def player_stand(self):
    return self.get_obs(), None, False

def dealer_draw_one(self):
    if self.done:
        return True
    self.dealer_hand.append(self.draw_card())
    return self.dealer_is_done()

def dealer_is_done(self):
    dealer_sum, dealer_soft = self.hand_value(self.dealer_hand)
    if dealer_sum >= 17 and not (dealer_sum == 17 and dealer_soft):
        self.done = True
        return True
    return False
```

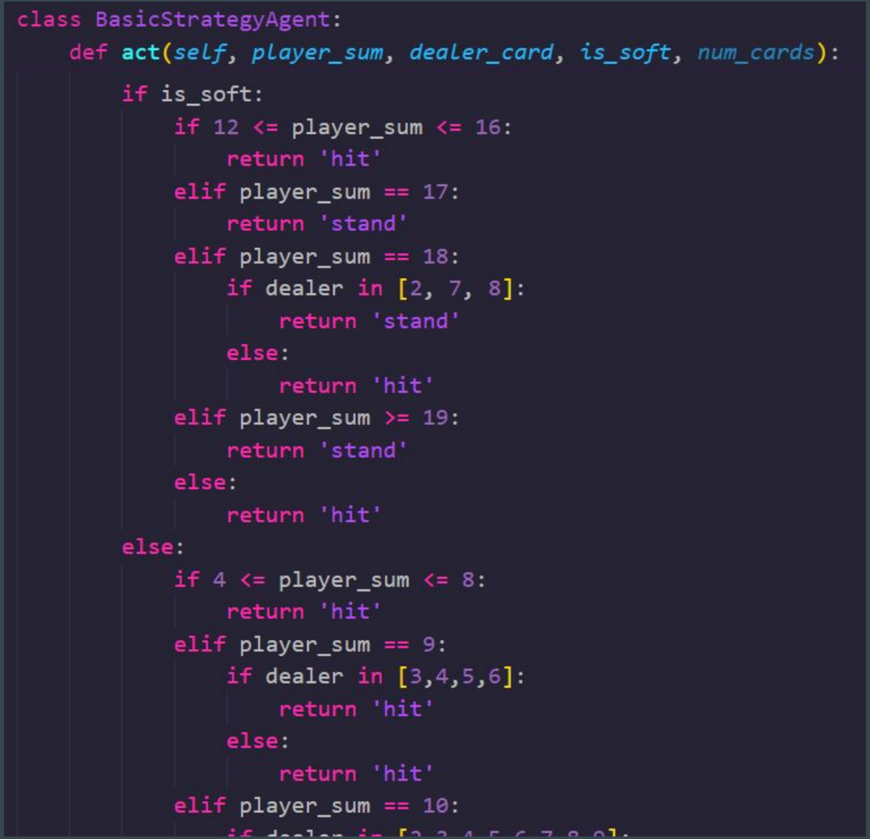
Random and Basic Strategy Agents

1. Random Agent
   - At each decision step, randomly selects Hit or Stand
   - No learning/logic
2. Basic Strategy Agent
   - Follows Blackjack basic strategy table
   - Purely rule-based/no learning



Dealer Player	2	3	4	5	6	7	8	9	10	11
Hard 4 - 12	H	H	H	H	H	H	H	H	H	H
Hard 13	H	H	H	H	H*	H	H	H	H	H
Hard 14	S*	H*	H*	S	S	H	H	H	H	H
Hard 15	S	S	S	S	S	H	H	H	H	H
Hard 16	S	S	S	S	S	H	H	H*	H*	S
Hard 17 - 20	S	S	S	S	S	S	S	S	S	S
Soft 12 - 16	H	H	H	H	H	H	H	H	H	H
Soft 17	S	S	S	H	H	H	H	H	H	H
Soft 18	S	S	S	S	S	S	S	H	S	S
Soft 18 - 20	S	S	S	S	S	S	S	S	S	S

Implement the Basic Strategy Agents



```python
class BasicStrategyAgent:
    def act(self, player_sum, dealer_card, is_soft, num_cards):
        if is_soft:
            if 12 <= player_sum <= 16:
                return 'hit'
            elif player_sum == 17:
                return 'stand'
            elif player_sum == 18:
                if dealer in [2, 7, 8]:
                    return 'stand'
                else:
                    return 'hit'
            elif player_sum >= 19:
                return 'stand'
            else:
                return 'hit'
        else:
            if 4 <= player_sum <= 8:
                return 'hit'
            elif player_sum == 9:
                if dealer in [3,4,5,6]:
                    return 'hit'
                else:
                    return 'hit'
            elif player_sum == 10:
                if dealer in [2,3,4,5,6,7,8,9]:
```

Expectimax Agent

Objective

Compare expected outcomes of 'Hit' and 'Stand' to decide optimal move.

Approach

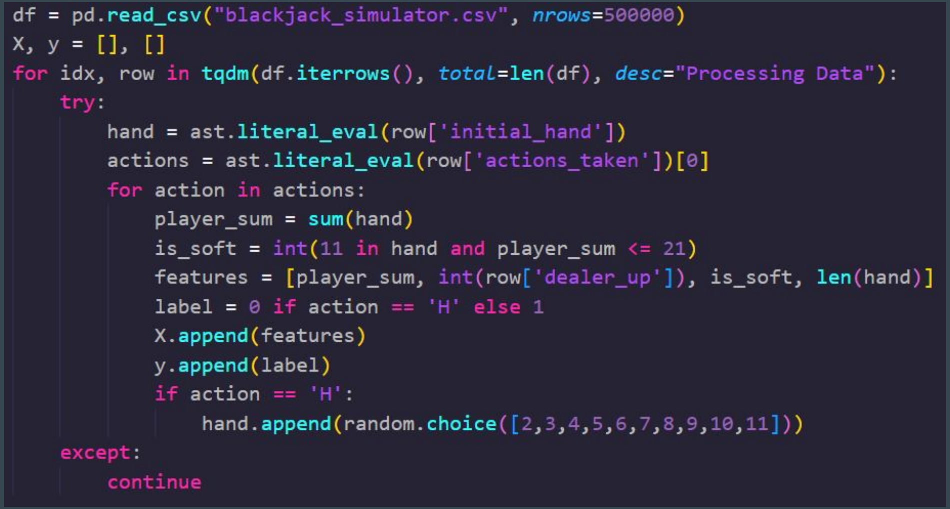
- 1st Loop (Stand):
  - Loop through all dealer possible values (2–11).
  - If dealer < 17, assume dealer draws to 17.
  - Compute difference:
    Agent Value - Dealer Value
  - Accumulate differences to get Stand EV.
- 2nd Loop (Hit):
  - Agent draws all possible cards (excluding dealer's card).
  - Add drawn card to agent's hand.
  - Re-run 1st loop logic for each new agent value.
  - Accumulate differences to get Hit EV.

Final Decision

- Compare averages:
  Hit EV vs Stand EV
- Choose the action with the higher expected value.

Random Forest Classifier Agent

- **Training Process (train_rfc_agent.py)**
  - Data Source: blackjack_simulator.csv
  - Features: player_sum, dealer_card, is_soft, num_cards
  - Label: 0=Hit, 1=Stand



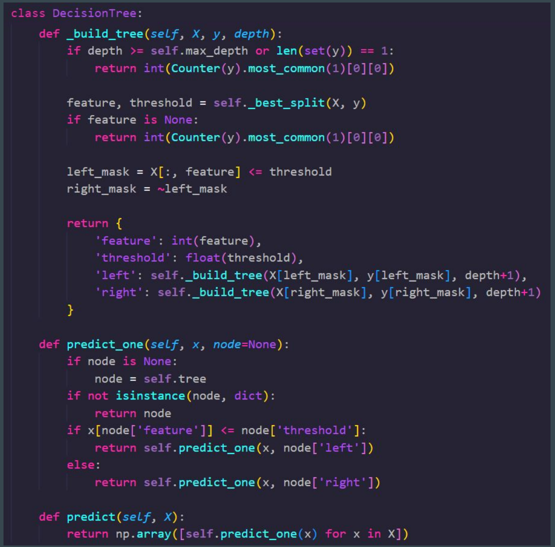
```
df = pd.read_csv("blackjack_simulator.csv", nrows=500000)
X, y = [], []
for idx, row in tqdm(df.iterrows(), total=len(df), desc="Processing Data"):
    try:
        hand = ast.literal_eval(row['initial_hand'])
        actions = ast.literal_eval(row['actions_taken'])[0]
        for action in actions:
            player_sum = sum(hand)
            is_soft = int(11 in hand and player_sum <= 21)
            features = [player_sum, int(row['dealer_up']), is_soft, len(hand)]
            label = 0 if action == 'H' else 1
            X.append(features)
            y.append(label)
            if action == 'H':
                hand.append(random.choice([2,3,4,5,6,7,8,9,10,11]))
    except:
        continue
```

Random Forest Classifier Agent

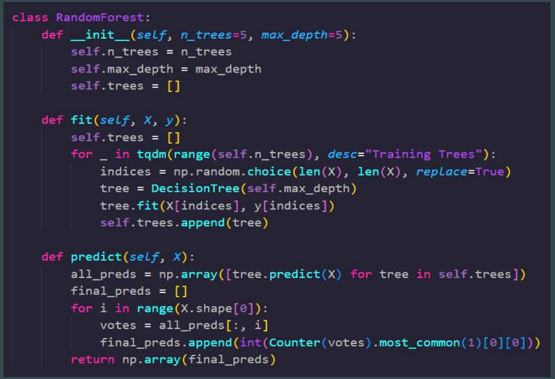
Training Process (train_rfc_agent.py)

Random Forest Training Logic

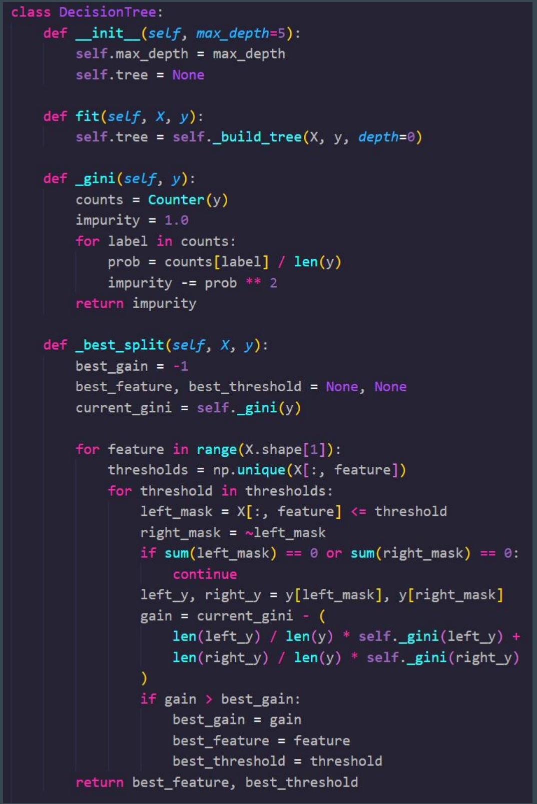
- Train multiple decision trees (default: 5 trees, max depth 5) using Bagging for diversity.
- Each tree is trained on a bootstrap sample (random subset with replacement).
- Decision trees are built recursively:
  1. Use Gini impurity to find the best split.
  2. Create left/right branches until max depth or pure leaf.



```python
class DecisionTree:
    def _build_tree(self, X, y, depth):
        if depth >= self.max_depth or len(set(y)) == 1:
            return int(Counter(y).most_common(1)[0][0])

        feature, threshold = self._best_split(X, y)
        if feature is None:
            return int(Counter(y).most_common(1)[0][0])

        left_mask = X[:, feature] <= threshold
        right_mask = ~left_mask

        return {
            'feature': int(feature),
            'threshold': float(threshold),
            'left': self._build_tree(X[left_mask], y[left_mask], depth+1),
            'right': self._build_tree(X[right_mask], y[right_mask], depth+1)
        }

    def predict_one(self, x, node=None):
        if node is None:
            node = self.tree
        if not isinstance(node, dict):
            return node
        if x[node['feature']] <= node['threshold']:
            return self.predict_one(x, node['left'])
        else:
            return self.predict_one(x, node['right'])

    def predict(self, X):
        return np.array([self.predict_one(x) for x in X])
```



```python
class RandomForest:
    def __init__(self, n_trees=5, max_depth=5):
        self.n_trees = n_trees
        self.max_depth = max_depth
        self.trees = []

    def fit(self, X, y):
        self.trees = []
        for _ in tqdm(range(self.n_trees), desc="Training Trees"):
            indices = np.random.choice(len(X), len(X), replace=True)
            tree = DecisionTree(self.max_depth)
            tree.fit(X[indices], y[indices])
            self.trees.append(tree)

    def predict(self, X):
        all_preds = np.array([tree.predict(X) for tree in self.trees])
        final_preds = []
        for i in range(X.shape[0]):
            votes = all_preds[:, i]
            final_preds.append(int(Counter(votes).most_common(1)[0][0]))
        return np.array(final_preds)
```



```python
class DecisionTree:
    def __init__(self, max_depth=5):
        self.max_depth = max_depth
        self.tree = None

    def fit(self, X, y):
        self.tree = self._build_tree(X, y, depth=0)

    def _gini(self, y):
        counts = Counter(y)
        impurity = 1.0
        for label in counts:
            prob = counts[label] / len(y)
            impurity -= prob ** 2
        return impurity

    def _best_split(self, X, y):
        best_gain = -1
        best_feature, best_threshold = None, None
        current_gini = self._gini(y)

        for feature in range(X.shape[1]):
            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                left_mask = X[:, feature] <= threshold
                right_mask = ~left_mask
                if sum(left_mask) == 0 or sum(right_mask) == 0:
                    continue
                left_y, right_y = y[left_mask], y[right_mask]
                gain = current_gini - (
                    len(left_y) / len(y) * self._gini(left_y) +
                    len(right_y) / len(y) * self._gini(right_y)
                )
                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature
                    best_threshold = threshold
        return best_feature, best_threshold
```

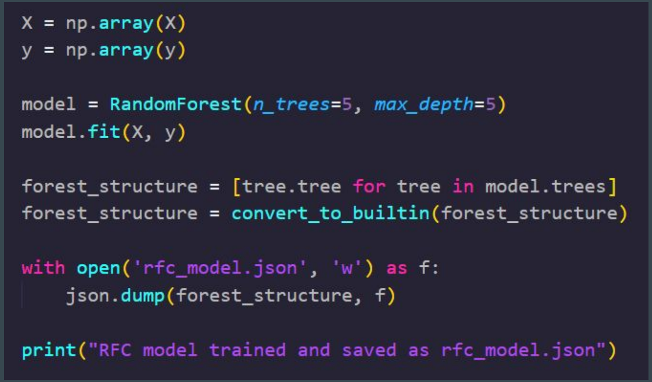
Random Forest Classifier Agent

Training Process (train_rfc_agent.py)

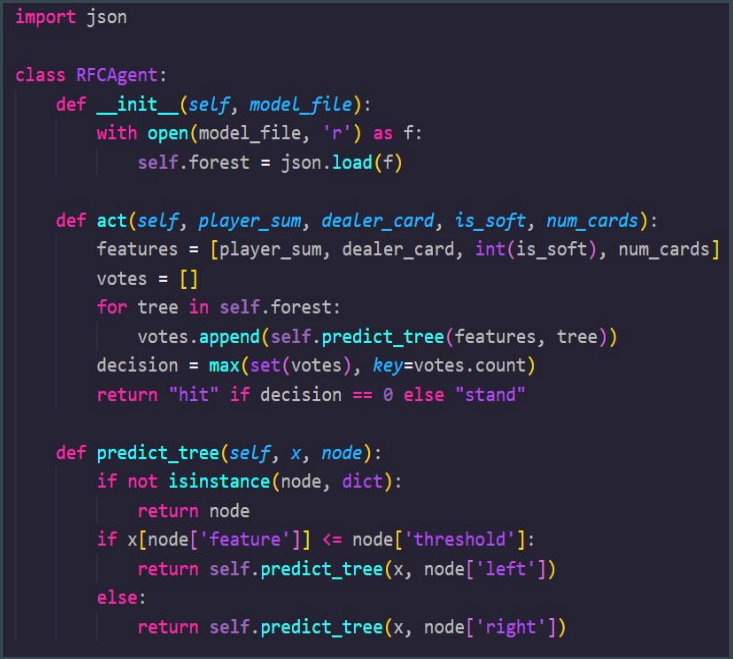
- Output: Model saved as JSON (rfc_model.json) with feature, threshold, left, right structure

Inference Logic (rfc_agent.py)

- Loads rfc_model.json
- For each state, traverse trees and collect votes
- Final action by majority voting: Hit/Stand



```
X = np.array(X)
y = np.array(y)

model = RandomForest(n_trees=5, max_depth=5)
model.fit(X, y)

forest_structure = [tree.tree for tree in model.trees]
forest_structure = convert_to_builtin(forest_structure)

with open('rfc_model.json', 'w') as f:
    json.dump(forest_structure, f)

print("RFC model trained and saved as rfc_model.json")
```



```
import json

class RFCAgent:
    def __init__(self, model_file):
        with open(model_file, 'r') as f:
            self.forest = json.load(f)

    def act(self, player_sum, dealer_card, is_soft, num_cards):
        features = [player_sum, dealer_card, int(is_soft), num_cards]
        votes = []
        for tree in self.forest:
            votes.append(self.predict_tree(features, tree))
        decision = max(set(votes), key=votes.count)
        return "hit" if decision == 0 else "stand"

    def predict_tree(self, x, node):
        if not isinstance(node, dict):
            return node
        if x[node['feature']] <= node['threshold']:
            return self.predict_tree(x, node['left'])
        else:
            return self.predict_tree(x, node['right'])
```

NEAT Agent

- NEAT (NeuroEvolution of Augmenting Topologies) evolves both network topology and weights for the Blackjack AI.
- Each **genome = a neural network**:
  - Node genes (key, bias, layer: input/output/hidden)
  - Connection genes (src, dst, weight, enabled, innovation)
- **Mutation operations**: Change weights, change biases, add connections, add nodes
- **Crossover**: **Combine two parents** based on innovation numbers
- **Speciation**: Group genomes by similarity to **preserve diversity**
- **Fitness**: Total reward from 1000 Blackjack games (input features: player sum, dealer card, is soft, num cards)
- Output: 1 node → "hit" (if > 0.5) or "stand“
- Final model: Saved as JSON (node biases, layers, connection weights) for inference.

NEAT Agent

Node / Connection Structures

```python
class Node:
    def __init__(self, key, bias=None, layer=None):
        self.key = key
        self.bias = random.uniform(-1, 1) if bias is None else bias
        self.layer = layer

class Conn:
    def __init__(self, src, dst, weight=None, enabled=True, tag=None):
        self.src = src
        self.dst = dst
        self.weight = random.uniform(-1, 1) if weight is None else weight
        self.enabled = enabled
        self.tag = tag
```

Mutation Logic

```python
def mutate(self):
    if random.random() < 0.8: self.mutate_weights()
    if random.random() < 0.8: self.mutate_biases()
    if random.random() < 0.03: self.add_conn()
    if random.random() < 0.05: self.add_node()
```

Inference Logic

```python
class NEATAgent:
    def __init__(self, model_path="neat_model.json"):
        with open(model_path, encoding="utf-8") as f: data = json.load(f)
        self.net = Network(data)
    def act(self, player_sum, dealer_card, is_soft, num_cards):
        x = [player_sum/21, dealer_card/10, int(is_soft), num_cards/5]
        return "hit" if self.net.activate(x) > 0.5 else "stand"
```

Crossover

```python
@staticmethod
def crossover(p1, p2, cfg):
    if p2.fitness > p1.fitness: p1, p2 = p2, p1
    child = Genome(None, cfg)
    child.nodes = {k: Node(k, n.bias, n.layer) for k, n in p1.nodes.items()}
    child.conns = {}
    for k, c in p1.conns.items():
        other = p2.conns.get(k, c)
        chosen = other if random.random() < 0.5 else c
        child.conns[k] = Conn(chosen.src, chosen.dst, chosen.weight, chosen.enabled, chosen.tag)
    return child
```

Evaluate

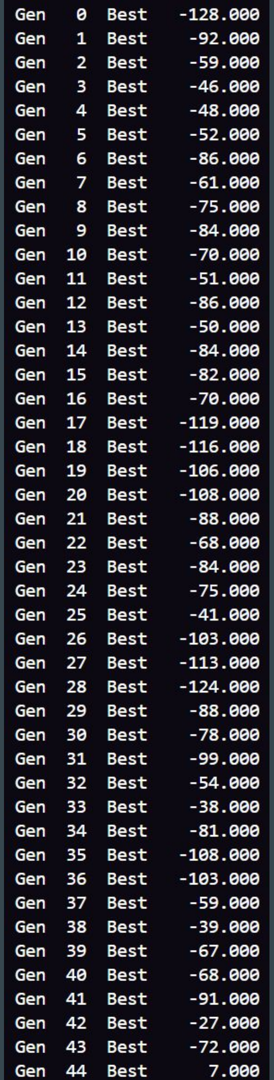
```python
def evaluate(self):
    env = BlackjackEnv()
    for g in self.genomes.values():
        net = Network(g)
        total = 0
        for _ in range(1000):
            state, done = env.reset(), False
            while not done:
                x = [state["player_sum"]/21, state["dealer_card"]/10, int(state["is_soft"]), len(env.player_hand)/5]
                action = "hit" if net.activate(x)[0] > 0.5 else "stand"
                state, reward, done = env.step(action)
                total += reward
        g.fitness = total
```

06

Evaluation Metrics

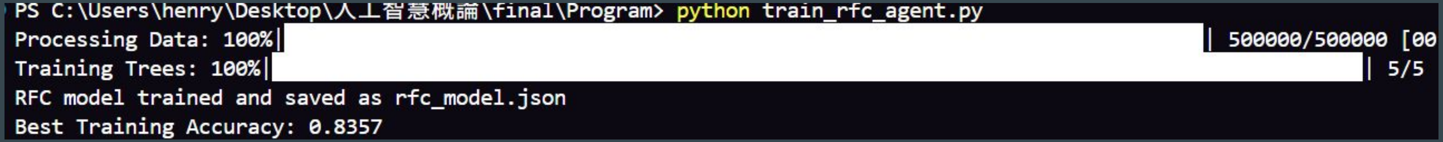
Evaluation Metrics

NEAT



```
Gen   0  Best   -128.000
Gen   1  Best    -92.000
Gen   2  Best    -59.000
Gen   3  Best    -46.000
Gen   4  Best    -48.000
Gen   5  Best    -52.000
Gen   6  Best    -86.000
Gen   7  Best    -61.000
Gen   8  Best    -75.000
Gen   9  Best    -84.000
Gen  10  Best    -70.000
Gen  11  Best    -51.000
Gen  12  Best    -86.000
Gen  13  Best    -50.000
Gen  14  Best    -84.000
Gen  15  Best    -82.000
Gen  16  Best    -70.000
Gen  17  Best   -119.000
Gen  18  Best   -116.000
Gen  19  Best   -106.000
Gen  20  Best   -108.000
Gen  21  Best    -88.000
Gen  22  Best    -68.000
Gen  23  Best    -84.000
Gen  24  Best    -75.000
Gen  25  Best    -41.000
Gen  26  Best   -103.000
Gen  27  Best   -113.000
Gen  28  Best   -124.000
Gen  29  Best    -88.000
Gen  30  Best    -78.000
Gen  31  Best    -99.000
Gen  32  Best    -54.000
Gen  33  Best    -38.000
Gen  34  Best    -81.000
Gen  35  Best   -108.000
Gen  36  Best   -103.000
Gen  37  Best    -59.000
Gen  38  Best    -39.000
Gen  39  Best    -67.000
Gen  40  Best    -68.000
Gen  41  Best    -91.000
Gen  42  Best    -27.000
Gen  43  Best    -72.000
Gen  44  Best      7.000
```

RFC



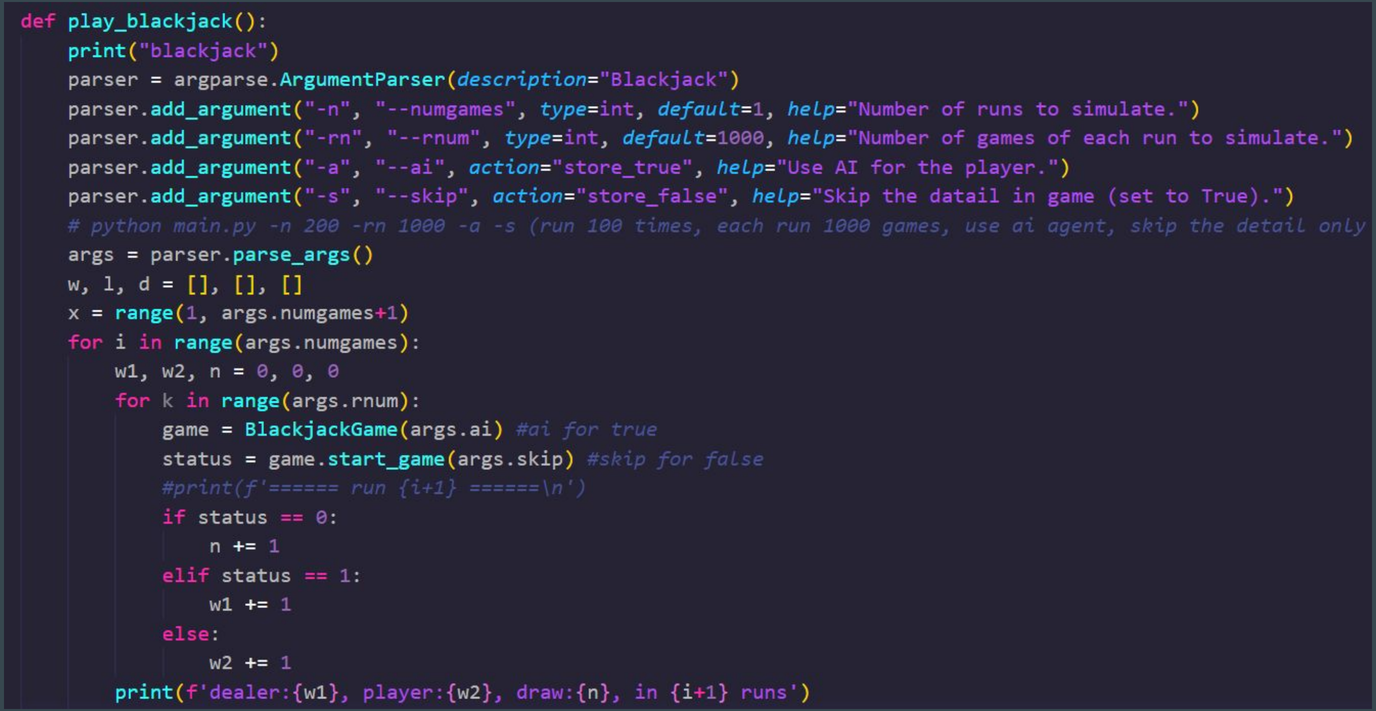
```
PS C:\Users\henry\Desktop\人工智慧概論\final\Program> python train_rfc_agent.py
Processing Data: 100%|                                                  | 500000/500000 [00
Training Trees: 100%|                                                             | 5/5
RFC model trained and saved as rfc_model.json
Best Training Accuracy: 0.8357
```

07

Result and Analysis

Compare Win Rate of All Agents (Expectimax)

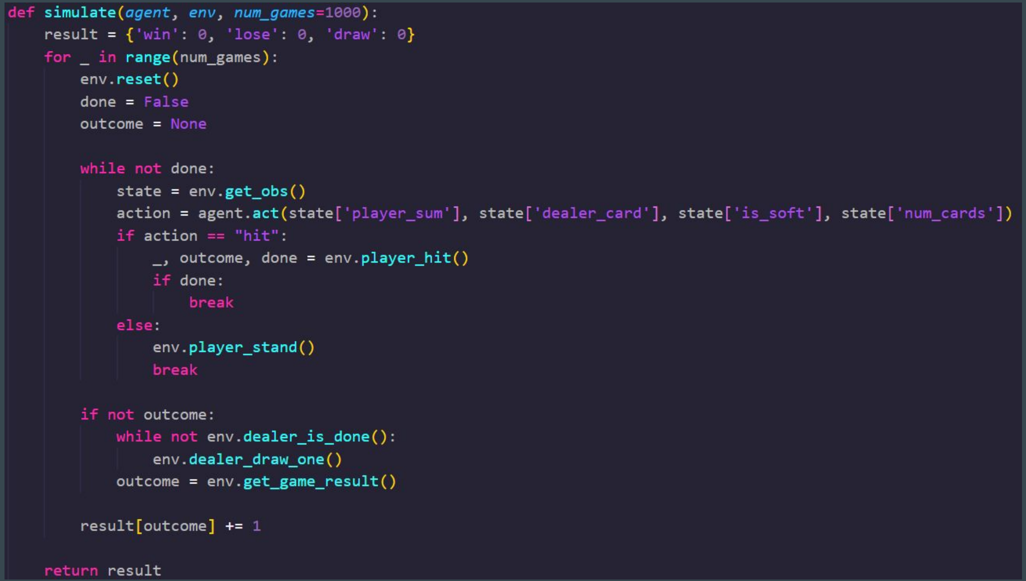
Each agent is tested over 1000 trials, each trial simulating 1000 games.
Metrics collected: Win rate



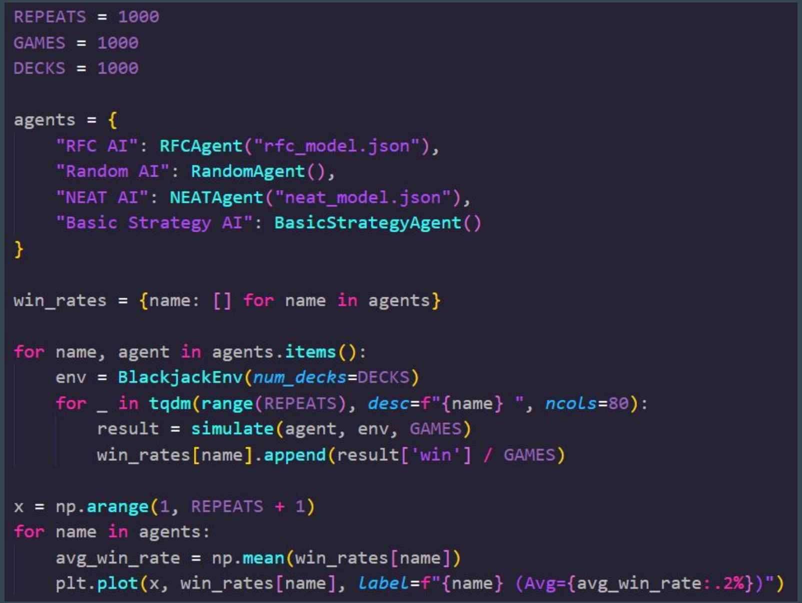
```python
def play_blackjack():
    print("blackjack")
    parser = argparse.ArgumentParser(description="Blackjack")
    parser.add_argument("-n", "--numgames", type=int, default=1, help="Number of runs to simulate.")
    parser.add_argument("-rn", "--rnum", type=int, default=1000, help="Number of games of each run to simulate.")
    parser.add_argument("-a", "--ai", action="store_true", help="Use AI for the player.")
    parser.add_argument("-s", "--skip", action="store_false", help="Skip the datail in game (set to True).")
    # python main.py -n 200 -rn 1000 -a -s (run 100 times, each run 1000 games, use ai agent, skip the detail only
    args = parser.parse_args()
    w, l, d = [], [], []
    x = range(1, args.numgames+1)
    for i in range(args.numgames):
        w1, w2, n = 0, 0, 0
        for k in range(args.rnum):
            game = BlackjackGame(args.ai) #ai for true
            status = game.start_game(args.skip) #skip for false
            #print(f'====== run {i+1} ======\n')
            if status == 0:
                n += 1
            elif status == 1:
                w1 += 1
            else:
                w2 += 1
        print(f'dealer:{w1}, player:{w2}, draw:{n}, in {i+1} runs')
```

Compare Win Rate of All Agents (RFC/Random/NEAT/Basic Strategy)

Each agent is tested over 1000 trials, each trial simulating 1000 games.
Metrics collected: Win rate



```python
def simulate(agent, env, num_games=1000):
    result = {'win': 0, 'lose': 0, 'draw': 0}
    for _ in range(num_games):
        env.reset()
        done = False
        outcome = None

        while not done:
            state = env.get_obs()
            action = agent.act(state['player_sum'], state['dealer_card'], state['is_soft'], state['num_cards'])
            if action == "hit":
                _, outcome, done = env.player_hit()
                if done:
                    break
            else:
                env.player_stand()
                break

        if not outcome:
            while not env.dealer_is_done():
                env.dealer_draw_one()
            outcome = env.get_game_result()

        result[outcome] += 1

    return result
```



```python
REPEATS = 1000
GAMES = 1000
DECKS = 1000

agents = {
    "RFC AI": RFCAgent("rfc_model.json"),
    "Random AI": RandomAgent(),
    "NEAT AI": NEATAgent("neat_model.json"),
    "Basic Strategy AI": BasicStrategyAgent()
}

win_rates = {name: [] for name in agents}

for name, agent in agents.items():
    env = BlackjackEnv(num_decks=DECKS)
    for _ in tqdm(range(REPEATS), desc=f"{name} ", ncols=80):
        result = simulate(agent, env, GAMES)
        win_rates[name].append(result['win'] / GAMES)

x = np.arange(1, REPEATS + 1)
for name in agents:
    avg_win_rate = np.mean(win_rates[name])
    plt.plot(x, win_rates[name], label=f"{name} (Avg={avg_win_rate:.2%})")
```

Results-Expectimax Agent

Win Rate

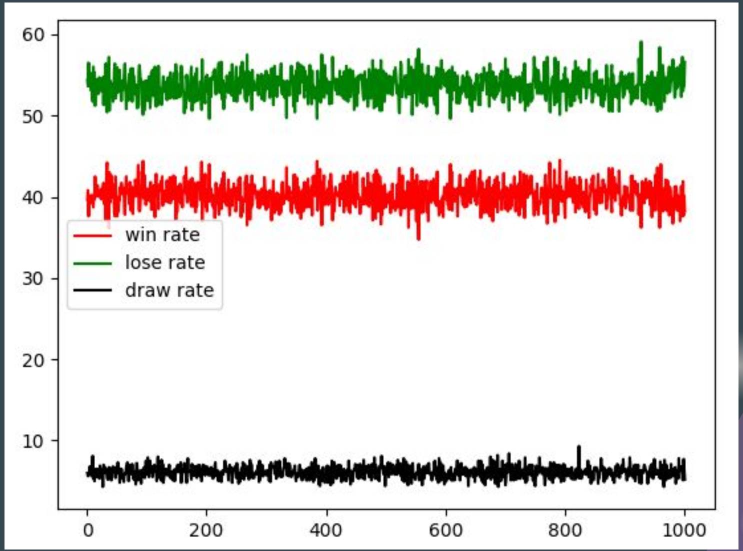
Win rate of approximately 40%
Not a bad outcome considering Blackjack's high uncertainty and the agent's lack of advanced features (e.g., card counting)

Stability

Variance < 1.5% across multiple runs
Indicates a consistently performing agent, not overly affected by random fluctuations

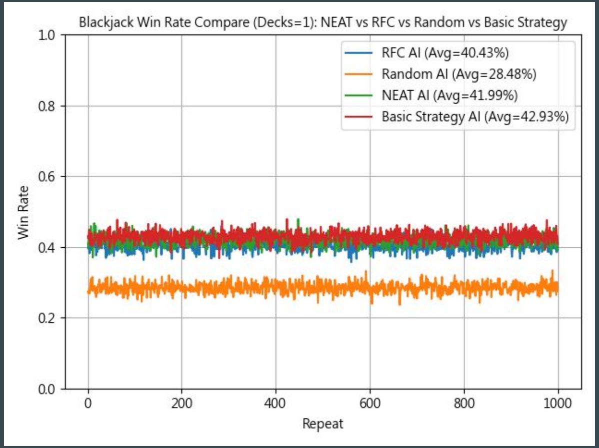
Interpretation

Expectimax performs well despite the game’s probabilistic nature
Suggests strong decision-making logic in terms of evaluating *Hit* vs *Stand*

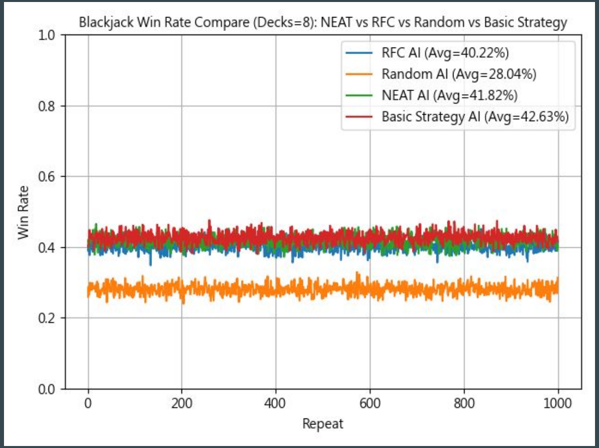


Results-RFC/Random/NEAT/Basic Strategy

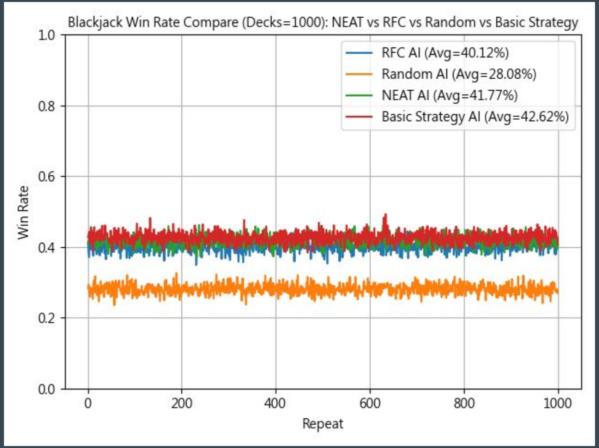
Deck = 1



Deck = 8



Deck = 1000



Discussion / Analysis

1. NEAT performed the best because it **learns without the dataset**

2. However, agents that **relied on the dataset** such as the Random Forest Classifier performs a bit worse due to the fact that **the dataset is unable to duplicate and represent all possible situations** that could happen

3. There is a less than 50% chance of winning for all agents because in the game of Blackjack, the dealer always will always have the upper hand: the player goes first, thus the player will always “bust” (exceed 21) first

The Better Model

By win rate:

NEAT >Basic strategy > Expectimax > RFC > Random

Conclusion

1. How does the player's win rate vary under different agents?

NEAT has the **best performance**, the win rate very close to Basic_Strategy Agent

2. Does the different number of decks affect the win rate?

No, win rate isn't affected by different numbers of agents.

3. Are the casino's rules inherently designed to favor the dealer?

Yes, based on the basic_strategy agent, the win rate of the player is about 43%, and the win rate of our implements are also roughly 40%; therefore, we conclude the rules of Blackjack are inherently designed to favor the dealer.

Limitation of our work

1. **No Casino Rule Variations**: Our environment assumes fixed rules and does not support actions like Split, Double, or Surrender, limiting real-world simulation.

2. **Dataset Constraints**: The blackjack_simulator.csv is a static dataset and does not include advanced strategies like card counting or memory-based logic.

Apply the model/method to practical use GUI Implementation

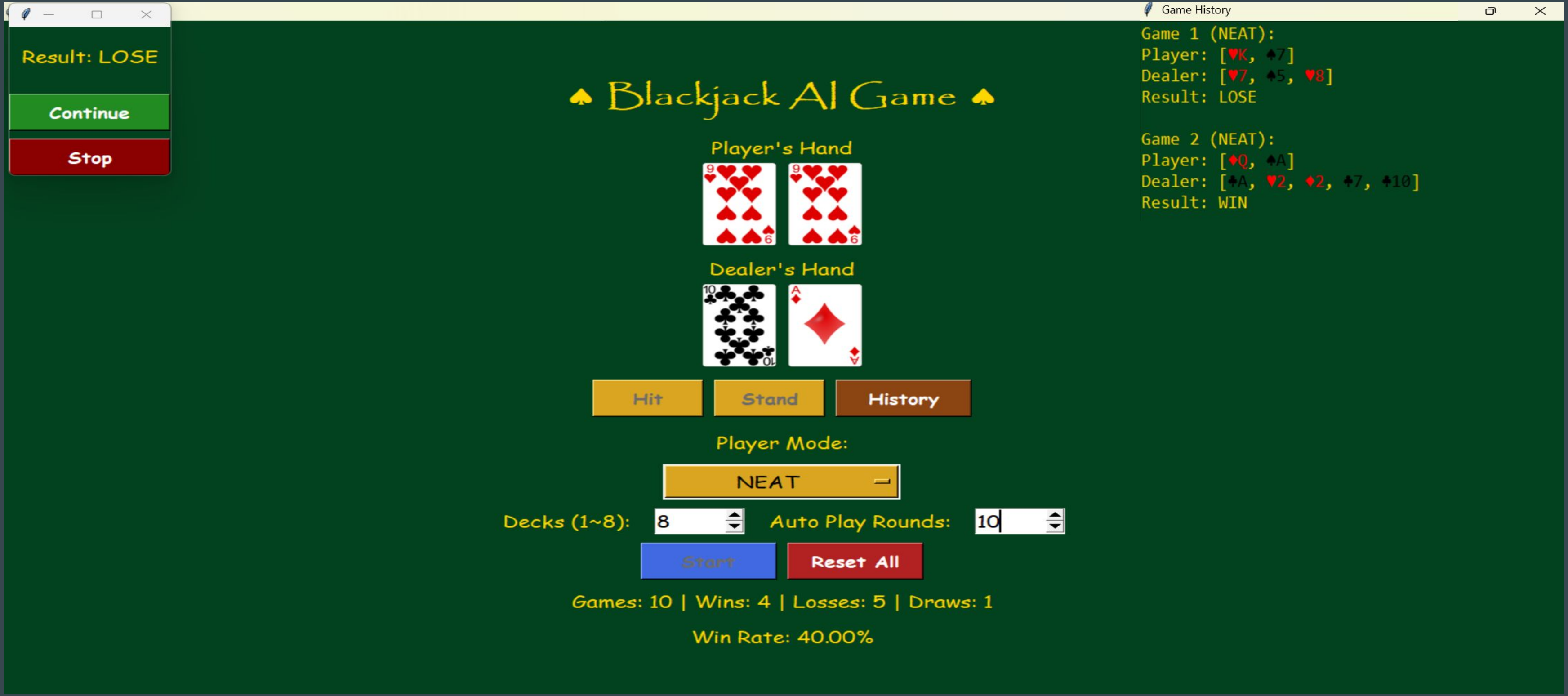
GUI Features

Component	Description
Mode Selection	Dropdown menu: Manual, Random, Basic, RFC, NEAT
Manual Play Buttons	Hit, Stand
AI Simulation Control	Auto-run games in batch, update results without dialog boxes
Card Display	Visual display of cards using **Vector Playing Cards** images
Game Status Display	Shows player/dealer hands, scores, wins, losses, draws
Result Dialog (Manual)	After each manual game, popup shows result and allows new game

Integration Logic:

1. GUI interacts with BlackjackEnv to get game state and render visuals.
2. When AI mode is selected, the GUI runs simulations automatically.
3. The GUI updates statistics and history in real-time.
4. Manual mode supports full gameplay with user decisions.

Apply the model/method to practical use
GUI Implementation



Decks (1~8): 8 Auto Play Rounds: 10

Games: 10 | Wins: 4 | Losses: 5 | Draws: 1

Win Rate: 40.00%

```
Game 1 (NEAT):
Player: [♥K, ♠7]
Dealer: [♥7, ♠5, ♥8]
Result: LOSE

Game 2 (NEAT):
Player: [♦Q, ♠A]
Dealer: [♣A, ♥2, ♦2, ♣7, ♣10]
Result: WIN
```

WORK DISTRIBUTION:

1. Review of Models:
   a. Bodnar: 111950020 李欣穎
   b. Lee: 110550092 林啟堯
2. Model Implementation:
   a. Expectimax Implementation: 109550072 何嘉娓
   b. Random Forest/Neat/GUI Implementation: 110550092 林啟堯
3. Presentation:
   a. Presentation Preliminary Draft: 109550202 白詩愷
   b. Presentation: 110550092 林啟堯、111950020 李欣穎
   c. Oral Presentation: 111950020 李欣穎

Thank You!