Poker Perceptron Bot and Analysis

COMP 380
Spring Semester 2024
Thomas McKeown, Daniel Daugbjerg, Kent Morris, Casey Klutznick
Dr. Jiang
May 15, 2024

Assertion:

The following program generates and uses poker data in order to predict the best option at each stage of a round of Texas Hold'em using multiple perceptrons. Simple forms of games are simulated in order to create data files which are then inputted into the perceptrons to create training and testing sets. The perceptrons are also directly used to play a live game of poker and attempt to win hands and fold when necessary.

This project was completed, functional to the requirements of the instructions, on time by 11:59 pm on 5/15/24.

Background:

The association of board games and artificial intelligence has been a popular topic for many years now and continues to be a good measure of machine learning capabilities. Our interest lies in the game, Texas Hold 'em, because of the fact that it is a game where luck is a very large factor, therefore it is not easily 'solved.' We need to use machine learning in order to create some sort of intuition and decide whether or not to fold given a set of cards. Those who have already built machines to play poker and provided documentation generally use classification machines to learn the specific patterns that give the best chance of winning a hand. Also, research indicates that using separate machines for each round of a poker hand allows for more precise decision-making and overall better results.

Our Approach:

Data Generation:

In order to generate simple and effective data that our perceptrons would be able to use in order to properly predict winning and losing hands, we created the python files, trainingbot.py and generate_training_data.py.

Trainingbot.py is a file containing the python class, DummyBot, which simulates playing a game of poker. It randomly generates cards as tuples with an integer representing the number and the suit of the card, also ensuring that no duplicates are generated. After generating cards for each round, including those for the simulated player and its opponents, the class is also able to find who the winner of the game is.

Using trainingbot.py, generate_training_data.py writes to four csv files in order to create data for each stage of the game (pre-flop, flop, turn, and river). This code simulates a game and at each stage, writes the current cards seen by the simulated player. The last value written to each csv file line is 1 for if the simulated player won and 0 if they lost. Since the game simulations are so simple and one simulation creates data for all four files, it is easy to simulate thousands of data records in the blink of an eye.

Some factors that had to be considered after seeing the outcome of training and testing the perceptrons was that the data generated was heavily influenced by the number of players in the simulation and the randomization of a given dataset could give a variety of different training results due to the amount of significant combinations in Texas Hold 'em. We found that 4 players tended to give the best balance of good hands correlating to winning games and bad hands correlating to losing games. This is because having 5 players on a table is just enough competition so that a good hand that stands out is likely to win. Having less players leads to more wins with all hands, and having more players leads to more losses all hands. Also, in order to train a perceptron to have good variance in decision-making, we had to randomly generate and partition a dataset to have just the right amount of variance in hands and a good split of wins and losses.

Training and Testing Perceptrons:

The perceptrons used for this project were imported from sklearn, which provides hyperparameter modifications such as alpha, max iterations before convergence, and initial weights. In order to input data into these perceptrons, we used the Pandas framework in order to read the generated csv files into dataframes, which were then split into different sections for training and testing using train_test_split from sklearn as well. These values were then easily inputted into the perceptron.

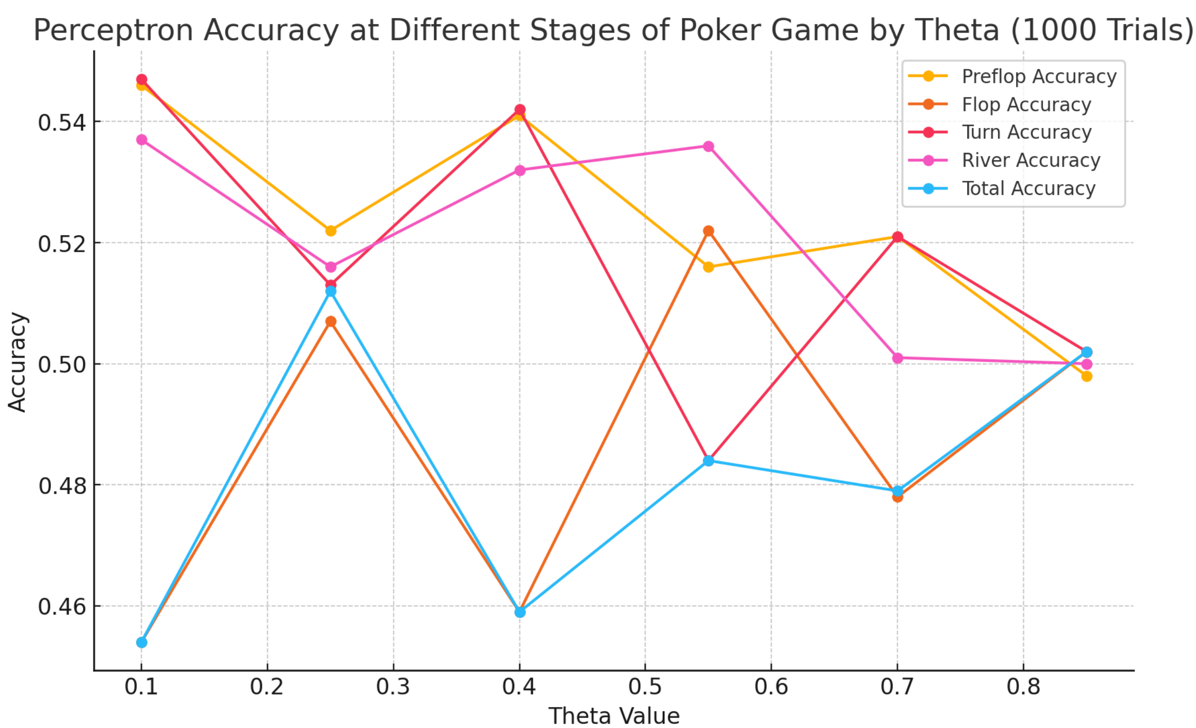
In order to adjust the hyperparameters for the perceptrons, we adjusted different hyperparameters and ran them with different datasets to find the best average performance. We found that lowering the learning rate and continuing training when no changes in weights were detected tends to produce the best observed results overall, but has variance in performance just like any other setting of hyperparameters. Our specific parameter setup has the learning rate at 0.25, 100,000 maximum epochs, and continues even after no change is detected in the weights.

Analysis:

At the start of this project the perceptrons were hovering around 60% accuracy, and now the perceptrons for each stage have an approximate accuracy of ~50-55% for the pre-flop, turn, and river, where the hyperparameters for the imported perceptron from sklearn were as described above. There were two main changes made that enhanced our bot. First, with the 60% starting accuracy, we had implemented binary activations for our perceptrons to begin with, 0 and 1. While this was mediocre, we encountered an issue where the bot would fold every time on pre-flop, after implementing bipolar activations for the perceptrons, this problem was resolved and we encountered better and more balanced generalizations. This is because the -1 and 1 allow the ability to check more of the degree of the hand, i.e. how good or bad the hand, not plainly good or bad. This bipolar introduction also was seen to enhance the balance of our bot as we saw more variation in the decisions and games being played. Secondly, the introduction of more opponents. Adding opponents caused the bot to lose more and gave it the ability to interpret better hands. These two paired together increased the bots poker ability, while seemingly decreasing the accuracy, but nevertheless making it have higher quality generalizations.

In our analysis we will be looking at the accuracy of each perceptron at each stage of the game and determining if given the output of the game, i.e. player won, opponent won, or there was a tie, whether or not our bot chose correctly. This is why it was so important to enhance our bot with the changes described above. For starters the pre-flop stage is where the bot can only see its own cards, so for it to determine whether or not to flop or play, just like regular poker, can be hard. This is why we see an average lower accuracy for the pre-flop. We created a separate file to test the accuracies of the perceptrons and ran it on different theta activations. This separate file, play_accuracy.py, evaluates the performance of our bot using the trained perceptron models at different stages of the game. It generates a simulated deck, deals cards for pre-flop, flop, turn, and river stages, and uses the perceptrons to make decisions based on these stages. The script runs a specified number of trials, input by the user, to test the accuracy of decisions at each stage against the final outcomes. It calculates and displays the accuracy of the bots' decisions for pre-flop, flop, turn, river, and overall game decisions, providing insights into the effectiveness of the perceptron models in predicting successful poker strategies. A table is shown below that features different thetas and their corresponding accuracies for the separate stages.

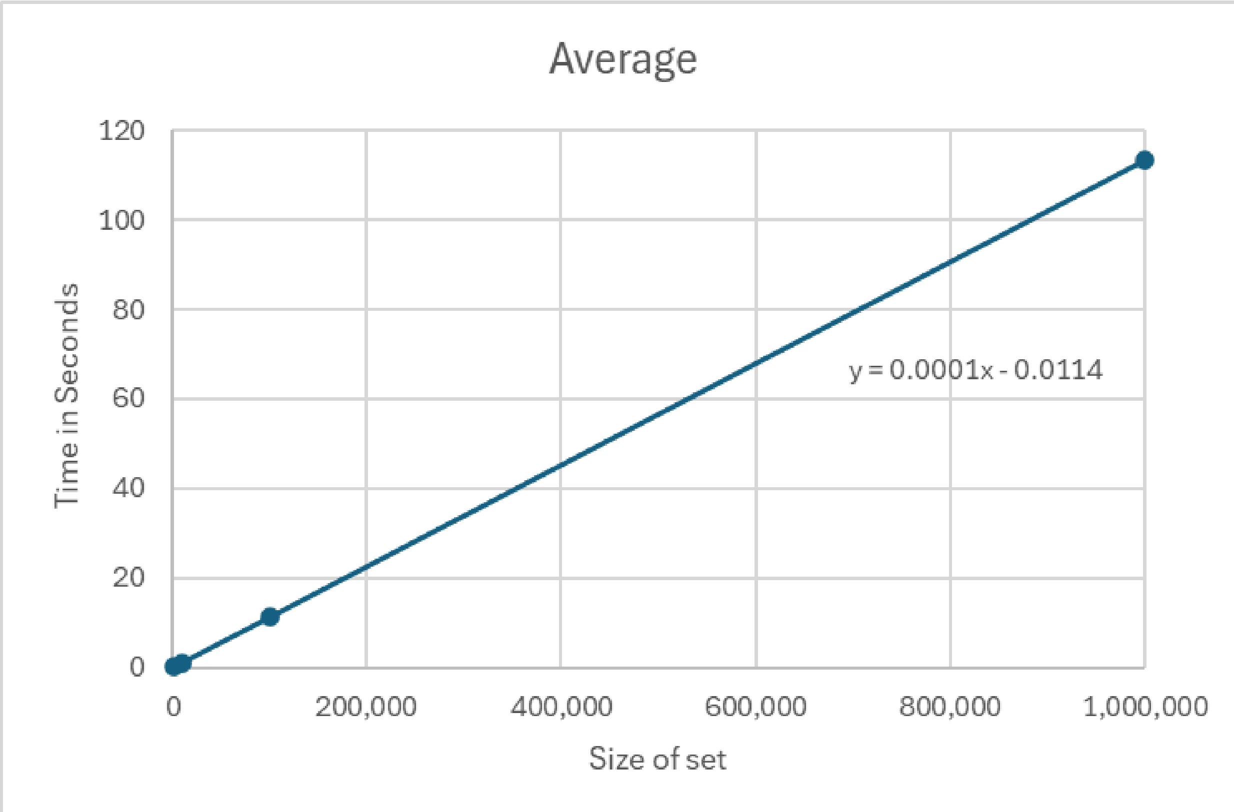
Theta	Preflop Accuracy	Flop Accuracy	Turn Accuracy	River Accuracy	Total Accuracy
0.1	0.546	0.454	0.547	0.537	0.454
0.25	0.522	0.507	0.513	0.516	0.512
0.4	0.541	0.459	0.542	0.532	0.459
0.55	0.516	0.522	0.484	0.536	0.484
0.7	0.521	0.478	0.521	0.501	0.479
0.85	0.498	0.502	0.502	0.5	0.502



There is a clear indication that the theta value influences the accuracy, but not uniformly across all stages of the game. For instance, while the preflop and turn stages perform best at lower theta values, the flop stage seems to peak slightly higher. For total accuracy, which considers all stages, does not necessarily peak where individual stage accuracies are highest, indicating a complex interaction between the stages. There can also clearly be seen a theta for which the output of the Accuracy is most similar, the .25 for theta. This is the theta we landed on for the best generalizations.

One way to potentially increase the accuracy of the bot would be to increase the number of samples in the training data. Our training program is capable of generating large datasets relatively quickly, but there is randomness involved in every generation which results in many various unseen scenarios causing inaccuracy for the bot. There are 2,598,960 hands possible in poker, but due to randomness, it is unlikely that they would all appear, even in a training sample of 10,000,000. Ideally a large set that covers all possible scenarios would exist and be easy to generate, but time restricts this possibility. A test on how much time it takes to generate data of different sizes is shown below. Each generation was created on the same computer under similar conditions and averaged over 5 trials for accuracy. Each trial’s length is in seconds and rounded to three decimal places. Timing was done using python’s time module. Timing begins after the user’s final input is entered and stops once all the data files have successfully been closed.

Set Size	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**
1,000	0.143	0.132	0.131	0.145	0.130	0.1362
10,000	1.148	1.144	1.135	1.140	1.150	1.1434
100,000	11.210	11.409	11.173	11.221	11.297	11.262
1,000,000	114.742	113.224	113.162	113.856	111.724	113.342



Each time the data set size is multiplied by ten, the amount of time it takes to generate the set is also multiplied by ten. Using the equation created by the trendline of this data set, the estimated amount of time to generate a set of size 10,000,000 would be around 1,000 seconds, or about 17 minutes. Creating a set of around a billion samples would take over 24 hours to create based on this trendline. To ensure the accuracy of this model, a test on generating 10,000,000 samples was employed with a timer to check if this set could be created within 20 minutes. The sample took 19 minutes to generate, suggesting that larger samples may not follow the linear path established by the dataset above and will likely be longer than predicted. This presents an issue where generating data and training more accurate bots requires either more computing power or significantly more time. While the generation of a set in the millions is somewhat reasonable time wise, larger sets do not always guarantee better results due to the randomness of the game and generation.

Conclusion:

This project was a great learning experience and way to lay out a basic formula in order to implement a learning machine into a real-world application. It allowed us to flesh out the full process of creating data, reading data, and predicting data through classification learning machines. Also, since this project was somewhat simplified in order to complete in the given amount of time, this project can serve as a blueprint for creating an even smarter and more efficient learning algorithm in order to properly predict poker hand outcomes with greater accuracy.

References:

https://medium.com/@tor_92315/machine-learning-and-card-games-6b210f8ec322

https://www.deepstack.ai/

Appendix A:

- Import sklearn, tkinter, pickle (all imports in code)
- Run generate_training_data.py to create your own data files (name data files “preflop.csv”, “flop.csv”, “turn.csv”, and “river.csv”)
- Run perceptron.py to save trained perceptrons with the previously generated data
- Run pokerGUI.py or play_accuracy.py in order to use/test the trained perceptrons on given poker inputs

Appendix B:

trainingbot.py

```
#python file for creating poker dummy bot in order to create training input data
import random
from enum import Enum

class Hand(Enum):
   HIGH_CARD = 0
   PAIR = 1
   TWO_PAIR = 2
   THREE_KIND = 3
   STRAIGHT = 4
   FLUSH = 5
   FULL_HOUSE = 6
   FOUR_KIND = 7
   STRAIGHT_FLUSH = 8
   ROYAL_FLUSH = 9

   # Define less-than comparison
   def __lt__(self, other):
       if isinstance(other, Hand):
           return self.value < other.value
       return NotImplemented

   # Define less-than-or-equal-to comparison
   def __le__(self, other):
       if isinstance(other, Hand):
           return self.value <= other.value
       return NotImplemented

   # Define greater-than comparison
   def __gt__(self, other):
       if isinstance(other, Hand):
           return self.value > other.value
       return NotImplemented

   # Define greater-than-or-equal-to comparison
   def __ge__(self, other):
       if isinstance(other, Hand):
           return self.value >= other.value
       return NotImplemented


class DummyBot:

   def __init__(self):
       self.hand = [] #dummy hand
       self.table = [] #cards on table
       self.suits = [set() for _ in range(12)] #tracks any repeated card generations
       self.opponents = [] #opponent hand


   #generate dummy's hand and flop
   def generate_hand_and_table(self):
       #hand
       for i in range(2):
           card = random.randint(2,14) #14 technically counts as 1 and 14 (ace)
           suit = random.randint(1,4)
           while suit in self.suits[card-3]: #check for existing card
               suit = ((suit)%4)+1
           self.suits[card-3].add(suit)
           self.hand.append((card,suit))

       #flop
```

```
        for i in range(3):
            card = random.randint(2,14)
            suit = random.randint(1,4)
            while len(self.suits[card-3]) == 4: #make sure not more than 4 of card number/face
                card = random.randint(2,14)
            while suit in self.suits[card-3]: #check for existing card
                suit = ((suit)%4)+1
            self.suits[card-3].add(suit)
            self.table.append((card,suit))


    #generate opponent for chance of losing/winning
    def generate_opponent(self):
        self.opponents.append([])
        for i in range(2):
            card = random.randint(2,14) #14 technically counts as 1 and 14 (ace)
            suit = random.randint(1,4)
            while len(self.suits[card-3]) == 4: #make sure not more than 4 of card number/face
                card = random.randint(2,14)
            while suit in self.suits[card-3]: #check for existing card
                suit = ((suit)%4)+1
            self.suits[card-3].add(suit)
            self.opponents[-1].append((card,suit))

    #generate another table card each round
    def generate_cards(self):
        card = random.randint(2,14)
        suit = random.randint(1,4)
        while len(self.suits[card-3]) == 4: #make sure not more than 4 of card number/face
            card = random.randint(2,14)
        while suit in self.suits[card-3]: #check for existing card
            suit = ((suit)%4)+1
        self.suits[card-3].add(suit)
        self.table.append((card,suit))

    def decide_winner(self):
        if len(self.hand + self.table) < 5 and len(self.opponents) < 1:
            print("not enough cards")
            return
        else:
            #create own hand
            bot_hand = self.check_hand(self.hand + self.table)
            #create opponent hands
            op_hands = []
            for o in self.opponents:
                op_hands.append(self.check_hand(o + self.table))

            res = 1
            #check if opponent has better hand
            for hand in op_hands:
                if hand[0] > bot_hand[0]:
                    return -1
                elif hand[0] == bot_hand[0]:
                    if hand[0] != Hand.STRAIGHT and hand[0] != Hand.FLUSH and hand[0] !=
Hand.STRAIGHT_FLUSH and hand[0] != Hand.ROYAL_FLUSH:
                        if hand[2] < bot_hand[2]:
                            res = 1
                        elif hand[2] > bot_hand[2]:
                            return -1
                        elif hand[1] > bot_hand[1]:
                            return -1
                    elif hand[1] > bot_hand[1]:
                        return -1
                    elif hand[1] == bot_hand[1]:
                        res = 1
            #if not, bot wins
            return res
```

```
def check_hand(self, hand):
    card_nums = [x[0] for x in hand]
    card_nums += [x[0] for x in self.table]
    card_nums.sort()
    card_suits = [x[1] for x in hand]
    card_suits += [x[1] for x in self.table]

    high = card_nums[-1]
    combo = Hand.HIGH_CARD
    #check for duplicate combos
    new_val = False
    combo_high = 0
    for i in range(1,len(card_nums)):
        if card_nums[i] == card_nums[i-1] and new_val == False:
            if combo == Hand.HIGH_CARD:
                combo = Hand.PAIR
                combo_high = card_nums[i]
            elif combo == Hand.PAIR:
                combo = Hand.THREE_KIND
                combo_high = card_nums[i]
            elif combo == Hand.THREE_KIND:
                combo = Hand.FOUR_KIND
                combo_high = card_nums[i]
        elif card_nums[i] == card_nums[i-1] and new_val:
            if combo == Hand.HIGH_CARD:
                combo = Hand.PAIR
                combo_high = card_nums[i]
            elif combo == Hand.PAIR:
                combo = Hand.TWO_PAIR
                combo_high = max(combo_high, card_nums[i])
            elif combo == Hand.THREE_KIND:
                combo = Hand.FULL_HOUSE
        elif card_nums[i] != card_nums[i-1] and combo != Hand.HIGH_CARD:
            new_val = True

    #check for same suit
    suit = card_suits[0]
    flush = True
    for i in range(1,len(card_suits)):
        if card_suits[i] != suit:
            flush = False

    #check for straight
    straight = False
    if combo == Hand.HIGH_CARD:
        straight = True
        for i in range(1,len(card_nums)):
            if card_nums[i] != card_nums[i-1]+1 and not (card_nums[i-1] == 14 and card_nums[i] == 2):
                straight = False

    if straight and flush:
        if card_nums[-1] == 14:
            combo = Hand.ROYAL_FLUSH
        else:
            combo = Hand.STRAIGHT_FLUSH
    elif combo != Hand.FOUR_KIND and combo != Hand.FULL_HOUSE:
        if flush:
            combo = Hand.FLUSH
        elif straight:
            combo = Hand.STRAIGHT

    return (combo, high, combo_high)

def get_hand(self):
    return self.hand
```

```
    def get_table(self):
        return self.table

    def get_opponents(self):
        return self.opponents
```

generate_training_data.py

```
from trainingbot import DummyBot
def main():
    user_in = input("How many records would you like to generate? ")
    record_num = int(user_in.strip().split()[0])
    pre_flop_f = input("What would you like the name of the pre-flop data file to be? ")
    flop = input("The flop data file? ")
    turn = input("The turn data file? ")
    river = input("The river data file? ")
    try:
        pre_flop = open(pre_flop_f, 'w')
        flop = open(flop, 'w')
        turn = open(turn, 'w')
        river = open(river, 'w')
        for _ in range(record_num):
            dummy = DummyBot()
            #simulate game to create data record
            dummy.generate_hand_and_table()
            hand = dummy.get_hand()
            table = dummy.get_table()
            for card in hand:
                pre_flop.write(str(card[0]) + "," + str(card[1]) + ",")
                flop.write(str(card[0]) + "," + str(card[1]) + ",")
                turn.write(str(card[0]) + "," + str(card[1]) + ",")
                river.write(str(card[0]) + "," + str(card[1]) + ",")
            for card in table:
                flop.write(str(card[0]) + "," + str(card[1]) + ",")
                turn.write(str(card[0]) + "," + str(card[1]) + ",")
                river.write(str(card[0]) + "," + str(card[1]) + ",")

            #4 opponents
            dummy.generate_opponent()
            dummy.generate_opponent()
            dummy.generate_opponent()
            dummy.generate_opponent()
            #dummy.generate_opponent()

            dummy.generate_cards()
            card = dummy.get_table()[-1]
            turn.write(str(card[0]) + "," + str(card[1]) + ",")
            river.write(str(card[0]) + "," + str(card[1]) + ",")

            dummy.generate_cards()
            card = dummy.get_table()[-1]
            river.write(str(card[0]) + "," + str(card[1]) + ",")

            winner = dummy.decide_winner()
            pre_flop.write(str(winner) + "\n")
            flop.write(str(winner) + "\n")
            turn.write(str(winner) + "\n")
            river.write(str(winner) + "\n")

        pre_flop.close()
        flop.close()
        turn.close()
        river.close()
    except Exception as e:
        print(e)
```

```
if __name__ == "__main__":
    main()
```

perceptron.py

```
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
import pandas as pd
import pickle
from sklearn import metrics

def main():
    perceptrons = [Perceptron(max_iter=100000, eta0=0.25, early_stopping=False, random_state=50), #preflop
    Perceptron(max_iter=100000, eta0=0.25, early_stopping=False, random_state=30), #flop
    Perceptron(max_iter=100000, eta0=0.25, early_stopping=False, random_state=50), #turn
    Perceptron(max_iter=100000, eta0=0.25, early_stopping=False, random_state=50)] #river
    #train machines automatically with assumed data files
    dfs = [pd.read_csv("preflop.csv"),
    pd.read_csv("flop.csv"),
    pd.read_csv("turn.csv"),
    pd.read_csv("river.csv")]

    x_y_data = []

    for round in range(len(dfs)):
        X = dfs[round].iloc[:, :-1].values #takes all values but last column
        y = dfs[round].iloc[:, -1].values  #takes value of last column
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
        perceptrons[round].fit(X_train, y_train)
        x_y_data.append((X_train, X_test, y_train, y_test))

    for round in range(len(dfs)):
        y_pred = perceptrons[round].predict(x_y_data[round][1]) #test with X_test
        accuracy = metrics.accuracy_score(x_y_data[round][3], y_pred) #compare prediction to y_test
        print(f'Accuracy: {accuracy:.2f}')

    save_perceptrons(perceptrons)


def save_perceptrons(perceptrons, filename="perceptronObjects"):
    with open(filename, 'wb') as f:
        pickle.dump(perceptrons, f)
        print(f"Perceptrons saved to {filename}")

if __name__ == "__main__":
    main()
```

pokerGUI.py

```
import tkinter as tk
from tkinter import messagebox
import random
from trainingbot import DummyBot
from perceptron import Perceptron
import pandas as pd
import numpy as np
import itertools
import os
import pickle
import random

# Function to create a complete deck
def create_deck():
    card_ranks = list(range(2, 15))  # 2 to Ace (2 to 14)
    card_suits = list(range(1, 5))   # 1 to 4 (Spades, Hearts, Diamonds, Clubs)

    # All combinations of ranks and suits to form a full deck
    deck = [(rank, suit) for rank in card_ranks for suit in card_suits]
```

```
    return deck

# Function to generate a 2-card pre-flop hand
def generate_preflop_hand(deck):
    # Draw two unique cards from the deck
    card1 = deck.pop()
    card2 = deck.pop()

    # Represent the hand as [rank1, suit1, rank2, suit2]
    hand = [(card1[0], card1[1]), (card2[0], card2[1])]

    return hand

# Function to generate the flop (3 community cards)
def generate_flop(deck):
    # Draw three unique cards from the deck for the flop
    flop = tuple(deck.pop() for _ in range(3))

    return flop

# Function to generate the river (1 community card)
def generate_river(deck):
    # Draw one card from the deck for the river
    river = deck.pop()

    # Return the card in the format [rank, suit]
    return (river[0], river[1])


# Function to generate the turn (1 community card)
def generate_turn(deck):
    # Draw one card from the deck for the turn
    turn = deck.pop()

    # Return the card in the format [rank, suit]
    return (turn[0], turn[1])


def convert_to_tuples(flat_list):
    if len(flat_list) % 2 != 0:
        raise ValueError("The input list must contain an even number of elements")

    # Create tuples from pairs of elements (rank, suit)
    return [(flat_list[i], flat_list[i + 1]) for i in range(0, len(flat_list), 2)]

# Function to evaluate poker hands and determine the winner
def compare_poker_hands(player_hand, opponent_hand, table):
    # Convert flat lists to tuples
    table_tuples = convert_to_tuples(table)

    # Combine each hand with the table to create all 5-card combinations
    player_full_hand = player_hand + table_tuples
    opponent_full_hand = opponent_hand + table_tuples
    print(player_full_hand)
    print(opponent_full_hand)
    # Function to get all possible 5-card combinations from a 7-card hand
    def get_combinations(full_hand):
        if len(full_hand) != 7:
            raise ValueError("Each full hand must contain 7 cards")

        return list(itertools.combinations(full_hand, 5))

    # Function to evaluate a poker hand and return a numeric value representing its strength
    def evaluate_poker_hand(hand):
    # Extract ranks and suits
        ranks = sorted([card[0] for card in hand])
        suits = [card[1] for card in hand]
```

```
        # Define poker hand values
        if is_royal_flush(ranks, suits):
            return (10, [])  # Royal Flush
        elif is_straight_flush(ranks, suits):
            return (9, [ranks[-1]])  # Straight Flush, kicker is the highest card
        elif is_four_of_a_kind(ranks):
            # Find the rank that has four of a kind
            four_rank = next(rank for rank in ranks if ranks.count(rank) == 4)
            kicker = next(rank for rank in ranks if rank != four_rank)
            return (8, [four_rank, kicker])  # Four of a Kind with kicker
        elif is_full_house(ranks):
            # Identify the triplet and pair
            triple_rank = next(rank for rank in ranks if ranks.count(rank) == 3)
            pair_rank = next(rank for rank in ranks if ranks.count(rank) == 2)
            return (7, [triple_rank, pair_rank])  # Full House
        elif is_flush(suits):
            # Return the sorted card ranks for flush (highest is the main value)
            return (6, ranks[::-1])  # Flush with kickers
        elif is_straight(ranks):
            # Straight is primarily determined by the highest card
            return (5, [ranks[-1]])  # Straight with highest card as kicker
        elif is_three_of_a_kind(ranks):
            # Identify the triplet and kickers
            triple_rank = next(rank for rank in ranks if ranks.count(rank) == 3)
            kickers = sorted([rank for rank in ranks if rank != triple_rank], reverse=True)
            return (4, [triple_rank] + kickers)  # Three of a Kind with kickers
        elif is_two_pair(ranks):
            # Identify both pairs and the highest kicker
            pairs = [rank for rank in set(ranks) if ranks.count(rank) == 2]
            kicker = max([rank for rank in ranks if ranks.count(rank) == 1])
            return (3, pairs + [kicker])  # Two Pair with kickers
        elif is_one_pair(ranks):
            # Identify the pair and the three highest kickers
            pair_rank = next(rank for rank in set(ranks) if ranks.count(rank) == 2)
            kickers = sorted([rank for rank in ranks if rank != pair_rank], reverse=True)
            return (2, [pair_rank] + kickers)  # One Pair with kickers
        else:
            # For High Card, the order of all cards is the key
            return (1, ranks[::-1])  # High Card, reversed to sort by highest


    # Functions to determine specific poker hands
    def is_royal_flush(ranks, suits):
        return set(ranks) == {10, 11, 12, 13, 14} and len(set(suits)) == 1

    def is_straight_flush(ranks, suits):
        return is_straight(ranks) and is_flush(suits)

    def is_four_of_a_kind(ranks):
        return any(ranks.count(rank) == 4 for rank in ranks)

    def is_full_house(ranks):
        return len(set(ranks)) == 2 and any(ranks.count(rank) == 3  for rank in ranks)

    def is_flush(suits):
        return len(set(suits)) == 1

    def is_straight(ranks):
        return all(ranks[i] + 1 == ranks[i + 1] for i in range(4))

    def is_three_of_a_kind(ranks):
        return any(ranks.count(rank) == 3 for rank in ranks)

    def is_two_pair(ranks):
        return len(set(ranks)) == 3
```

```
    def is_one_pair(ranks):
        return len(set(ranks)) == 4

    # Get all 5-card combinations for each hand
    player_combinations = get_combinations(player_full_hand)
    opponent_combinations = get_combinations(opponent_full_hand)

    # Evaluate the best 5-card combination for each hand
    player_best_hand = max(player_combinations, key=evaluate_poker_hand)
    opponent_best_hand = max(opponent_combinations, key=evaluate_poker_hand)

    # Compare the evaluated values of the best hands
    player_best_value = evaluate_poker_hand(player_best_hand)
    opponent_best_value = evaluate_poker_hand(opponent_best_hand)

    # Determine the winner, considering the main hand value and kickers
    if player_best_value > opponent_best_value:
        return "Player wins!"
    elif player_best_value < opponent_best_value:
        return "Opponent wins!"
    else:
        # If main values are equal, compare the kickers
        player_kickers = player_best_value[1]
        opponent_kickers = opponent_best_value[1]

        if player_kickers > opponent_kickers:
            return "Player wins (kickers)!"
        elif player_kickers < opponent_kickers:
            return "Opponent wins (kickers)!"
        else:
            return "It's a tie!"


# Utility function to map card numbers to names
def get_card_name(card):
    card_dict = {
        11: "J",
        12: "Q",
        13: "K",
        14: "A",
    }
    card_number = card[0]
    card_suit = card[1]
    suit_dict = {
        1: "♠",
        2: "♥",
        3: "♦",
        4: "♣",
    }
    return f"{card_dict.get(card_number, card_number)}{suit_dict[card_suit]}"

# Function to load perceptrons from a file
def load_perceptrons(filename="perceptronObjects"):
    try:
        with open(filename, 'rb') as f:
            perceptrons = pickle.load(f)
        return perceptrons
    except FileNotFoundError:
        messagebox.showerror("Error", f"Perceptrons file '{filename}' not found. Please train perceptrons
first.")
        raise SystemExit(1)  # Exit application gracefully
    except Exception as e:
        messagebox.showerror("Error", f"Error loading perceptrons: {str(e)}")
        raise SystemExit(1)  # Exit application gracefully

# Enum-like values to track game stages
PRE_FLOP = 0
```

```
FLOP = 1
TURN = 2
RIVER = 3
WINNER = 4
PLAYAGAIN = 5

# Basic GUI setup for Poker Texas Hold'em
class PokerGUI(tk.Tk):
   def __init__(self):
       super().__init__()
       self.title("Poker Texas Hold'em")
       self.geometry("400x400")
        # Create an outer frame to simulate the red border
       border_width = 10  # Adjust the border width as desired
       self.outer_frame = tk.Frame(self, bg='red', padx=border_width, pady=border_width)  # Red border
       self.outer_frame.pack(fill=tk.BOTH, expand=True)  # Fill the window

       # Create an inner frame for the content
       self.inner_frame = tk.Frame(self.outer_frame, bg='green')  # Your original background color
       self.inner_frame.pack(fill=tk.BOTH, expand=True)  # Fill the inner frame

       self.perceptrons = load_perceptrons()  # Load perceptrons

       self.create_widgets()
       self.game_stage = PRE_FLOP  # Initialize game stage
       self.deck = []
       self.table = []

   def create_widgets(self):
       # Create poker game widgets with color customization
       self.player_hand_label = tk.Label(self.inner_frame, text="Player Hand:", bg='green', fg='white')  #
Green with white text
       self.flop_label = tk.Label(self.inner_frame, text="Flop:", bg='green', fg='white')  # Green with
white text
       self.turn_label = tk.Label(self.inner_frame, text="Turn:", bg='green', fg='white')  # Green with
white text
       self.river_label = tk.Label(self.inner_frame, text="River:", bg='green', fg='white')  # Green with
white text
       self.opponent_hand_label = tk.Label(self.inner_frame, text="Opponent Hand:", bg='green',
fg='firebrick1')  # Green with white text

       self.deal_button = tk.Button(self.inner_frame, text="Deal", command=self.deal, bg='red',
fg='black')  # Red for deal button

       self.result_label = tk.Label(self.inner_frame, text="", bg='green', fg='white')  # Green with white
text

       # Pack widgets with padding
       self.player_hand_label.pack(pady=5)
       self.flop_label.pack(pady=5)
       self.turn_label.pack(pady=5)
       self.river_label.pack(pady=5)
       self.opponent_hand_label.pack(pady=5)
       self.deal_button.pack(pady=10)
       self.result_label.pack(pady=5)

   def clear_labels(self):
       # Clear all card-related labels
       self.player_hand_label.config(text="Player Hand:")
       self.flop_label.config(text="Flop:")
       self.turn_label.config(text="Turn:")
       self.river_label.config(text="River:")
       self.opponent_hand_label.config(text="Opponent Hand:")
       self.result_label.config(text="")

       self.player_hand_label.pack(pady=5)
       self.flop_label.pack(pady=5)
```

```
        self.turn_label.pack(pady=5)
        self.river_label.pack(pady=5)
        self.opponent_hand_label.pack(pady=5)
    def deal(self):
        if self.game_stage == PRE_FLOP:
            # Deal player and opponent hands
            self.player_table = []
            self.opponent_table = []
            self.player_hand = []
            self.opponent_hand = []
            self.table = []
            self.deck = create_deck()
            random.shuffle(self.deck)

            # Get player's hand
            self.player_hand = generate_preflop_hand(self.deck)
            player_hand_str = ", ".join([get_card_name(card) for card in self.player_hand])
            self.player_hand_label.config(text=f"Player Hand: {player_hand_str}")

            self.player_table.extend([card for sublist in self.player_hand for card in sublist])
            hand = np.array(self.player_table).reshape(1, -1)

            # Get opponent's hand
            self.opponent_hand = generate_preflop_hand(self.deck)
            opponent_hand_str = ", ".join([get_card_name(card) for card in self.opponent_hand])
            self.opponent_hand_label.config(text=f"Opponent Hand: {opponent_hand_str}")

            hand = np.array(self.player_table).reshape(1, -1)

            try:
                preflop_decision = self.perceptrons[0].predict(hand)
                if preflop_decision == 1:
                    self.result_label.config(text="POKER BOT says: Play on Pre-Flop")
                else:
                    self.result_label.config(text="POKER BOT says: Fold on Pre-Flop")
            except Exception as e:
                messagebox.showerror("Error", f"Perceptron prediction error on pre-flop: {str(e)}")
                return

            # Change Deal button text to "Deal Flop"
            self.deal_button.config(text="Deal Flop")
            self.game_stage = FLOP  # Move to next stage

        elif self.game_stage == FLOP:
            # Deal the flop and run perceptron decision
            flop = generate_flop(self.deck)  # The first three community cards
            self.table.extend([card for sublist in flop for card in sublist])
            flop_str = ", ".join([get_card_name(card) for card in flop])
            self.flop_label.config(text=f"Flop: {flop_str}")

            # Use perceptron to decide next step based on flop data

            self.player_table.extend([card for sublist in flop for card in sublist])
            hand = np.array(self.player_table).reshape(1, -1)

            try:
                flop_decision = self.perceptrons[1].predict(hand)
                if flop_decision == 1:
                    self.result_label.config(text="POKER BOT says: Play on Flop")
                else:
                    self.result_label.config(text="POKER BOT says: Fold on Flop")
            except Exception as e:
                messagebox.showerror("Error", f"Perceptron prediction error on flop: {str(e)}")
                return

            # Change Deal button text to "Deal Turn"
            self.deal_button.config(text="Deal Turn")
```

```
                self.game_stage = TURN

            elif self.game_stage == TURN:
                # Deal the turn and run perceptron prediction
                  # Add a card for the turn
                turn = generate_turn(self.deck)
                turn_str = get_card_name(turn)
                self.turn_label.config(text=f"Turn: {turn_str}")
                self.table.extend([card for sublist in [turn] for card in sublist])
                self.player_table.extend([card for sublist in [turn] for card in sublist])
                hand = np.array(self.player_table).reshape(1, -1)

                try:
                    turn_decision = self.perceptrons[2].predict(hand)
                    if turn_decision == 1:
                        self.result_label.config(text="POKER BOT says: Play on Turn")
                    else:
                        self.result_label.config(text="POKER BOT says: Fold on Turn")
                except Exception as e:
                    messagebox.showerror("Error", f"Perceptron prediction error on turn: {str(e)}")
                    return

                # Change Deal button text to "Deal River"
                self.deal_button.config(text="Deal River")
                self.game_stage = RIVER

            elif self.game_stage == RIVER:
                # Deal the river and run perceptron prediction
                river = generate_river(self.deck)
                river_str = get_card_name(river)
                self.river_label.config(text=f"River: {river_str}")

                self.table.extend([card for sublist in [river] for card in sublist])
                self.player_table.extend([card for sublist in [river] for card in sublist])
                hand = np.array(self.player_table).reshape(1, -1)

                try:
                    river_decision = self.perceptrons[3].predict(hand)
                    if river_decision == 1:
                        self.result_label.config(text="POKER BOT says: Play on River")
                    else:
                        self.result_label.config(text="POKER BOT says: Fold on River")
                except Exception as e:
                    messagebox.showerror("Error", f"Perceptron prediction error on river: {str(e)}")
                    return

                self.deal_button.config(text="Decide Winner")
                self.game_stage = WINNER

                # Determine the winner
            elif self.game_stage == WINNER:
                print(self.player_hand)
                print(self.opponent_hand)
                print(self.table)
                winner = compare_poker_hands(self.player_hand,self.opponent_hand,self.table)

                self.result_label.config(text=winner)

                self.deal_button.config(text="Play Again")
                self.game_stage = PLAYAGAIN
            elif self.game_stage == PLAYAGAIN:
                # Change Deal button text to "Start New Game"
                self.clear_labels()
                self.game_stage = PRE_FLOP

# Create and run the GUI application
if __name__ == "__main__":
```

```
    app = PokerGUI()
    app.mainloop()
```

play_accuracy.py

```
import itertools
from perceptron import Perceptron
import pandas as pd
import numpy as np
import itertools
import os
import pickle
import random

def create_deck():
   card_ranks = list(range(2, 15))  # 2 to Ace (2 to 14)
   card_suits = list(range(1, 5))   # 1 to 4 (Spades, Hearts, Diamonds, Clubs)

   # All combinations of ranks and suits to form a full deck
   deck = [(rank, suit) for rank in card_ranks for suit in card_suits]
   return deck

# Function to generate a 2-card pre-flop hand
def generate_preflop_hand(deck):
   # Draw two unique cards from the deck
   card1 = deck.pop()
   card2 = deck.pop()

   # Represent the hand as [rank1, suit1, rank2, suit2]
   hand = [(card1[0], card1[1]), (card2[0], card2[1])]

   return hand

# Function to generate the flop (3 community cards)
def generate_flop(deck):
   # Draw three unique cards from the deck for the flop
   flop = tuple(deck.pop() for _ in range(3))

   return flop

# Function to generate the river (1 community card)
def generate_river(deck):
   # Draw one card from the deck for the river
   river = deck.pop()

   # Return the card in the format [rank, suit]
   return (river[0], river[1])


# Function to generate the turn (1 community card)
def generate_turn(deck):
   # Draw one card from the deck for the turn
   turn = deck.pop()

   # Return the card in the format [rank, suit]
   return (turn[0], turn[1])


def convert_to_tuples(flat_list):
   if len(flat_list) % 2 != 0:
       raise ValueError("The input list must contain an even number of elements")

   # Create tuples from pairs of elements (rank, suit)
   return [(flat_list[i], flat_list[i + 1]) for i in range(0, len(flat_list), 2)]

# Function to evaluate poker hands and determine the winner
def compare_poker_hands(player_hand, opponent_hand, table):
```

```
# Convert flat lists to tuples
table_tuples = convert_to_tuples(table)

# Combine each hand with the table to create all 5-card combinations
player_full_hand = player_hand + table_tuples
opponent_full_hand = opponent_hand + table_tuples
# Function to get all possible 5-card combinations from a 7-card hand
def get_combinations(full_hand):
    if len(full_hand) != 7:
        raise ValueError("Each full hand must contain 7 cards")

    return list(itertools.combinations(full_hand, 5))

# Function to evaluate a poker hand and return a numeric value representing its strength
def evaluate_poker_hand(hand):
# Extract ranks and suits
    ranks = sorted([card[0] for card in hand])
    suits = [card[1] for card in hand]

    # Define poker hand values
    if is_royal_flush(ranks, suits):
        return (10, [])  # Royal Flush
    elif is_straight_flush(ranks, suits):
        return (9, [ranks[-1]])  # Straight Flush, kicker is the highest card
    elif is_four_of_a_kind(ranks):
        # Find the rank that has four of a kind
        four_rank = next(rank for rank in ranks if ranks.count(rank) == 4)
        kicker = next(rank for rank in ranks if rank != four_rank)
        return (8, [four_rank, kicker])  # Four of a Kind with kicker
    elif is_full_house(ranks):
        # Identify the triplet and pair
        triple_rank = next(rank for rank in ranks if ranks.count(rank) == 3)
        pair_rank = next(rank for rank in ranks if ranks.count(rank) == 2)
        return (7, [triple_rank, pair_rank])  # Full House
    elif is_flush(suits):
        # Return the sorted card ranks for flush (highest is the main value)
        return (6, ranks[::-1])  # Flush with kickers
    elif is_straight(ranks):
        # Straight is primarily determined by the highest card
        return (5, [ranks[-1]])  # Straight with highest card as kicker
    elif is_three_of_a_kind(ranks):
        # Identify the triplet and kickers
        triple_rank = next(rank for rank in ranks if ranks.count(rank) == 3)
        kickers = sorted([rank for rank in ranks if rank != triple_rank], reverse=True)
        return (4, [triple_rank] + kickers)  # Three of a Kind with kickers
    elif is_two_pair(ranks):
        # Identify both pairs and the highest kicker
        pairs = [rank for rank in set(ranks) if ranks.count(rank) == 2]
        kicker = max([rank for rank in ranks if ranks.count(rank) == 1])
        return (3, pairs + [kicker])  # Two Pair with kickers
    elif is_one_pair(ranks):
        # Identify the pair and the three highest kickers
        pair_rank = next(rank for rank in set(ranks) if ranks.count(rank) == 2)
        kickers = sorted([rank for rank in ranks if rank != pair_rank], reverse=True)
        return (2, [pair_rank] + kickers)  # One Pair with kickers
    else:
        # For High Card, the order of all cards is the key
        return (1, ranks[::-1])  # High Card, reversed to sort by highest


# Functions to determine specific poker hands
def is_royal_flush(ranks, suits):
    return set(ranks) == {10, 11, 12, 13, 14} and len(set(suits)) == 1

def is_straight_flush(ranks, suits):
    return is_straight(ranks) and is_flush(suits)
```

```
    def is_four_of_a_kind(ranks):
        return any(ranks.count(rank) == 4 for rank in ranks)

    def is_full_house(ranks):
        return len(set(ranks)) == 2 and any(ranks.count(rank) == 3  for rank in ranks)

    def is_flush(suits):
        return len(set(suits)) == 1

    def is_straight(ranks):
        return all(ranks[i] + 1 == ranks[i + 1] for i in range(4))

    def is_three_of_a_kind(ranks):
        return any(ranks.count(rank) == 3 for rank in ranks)

    def is_two_pair(ranks):
        return len(set(ranks)) == 3

    def is_one_pair(ranks):
        return len(set(ranks)) == 4

    # Get all 5-card combinations for each hand
    player_combinations = get_combinations(player_full_hand)
    opponent_combinations = get_combinations(opponent_full_hand)

    # Evaluate the best 5-card combination for each hand
    player_best_hand = max(player_combinations, key=evaluate_poker_hand)
    opponent_best_hand = max(opponent_combinations, key=evaluate_poker_hand)

    # Compare the evaluated values of the best hands
    player_best_value = evaluate_poker_hand(player_best_hand)
    opponent_best_value = evaluate_poker_hand(opponent_best_hand)

    # Determine the winner, considering the main hand value and kickers
    if player_best_value > opponent_best_value:
        return 1
    elif player_best_value < opponent_best_value:
        return 0
    else:
        # If main values are equal, compare the kickers
        player_kickers = player_best_value[1]
        opponent_kickers = opponent_best_value[1]

        if player_kickers > opponent_kickers:
            return 1
        elif player_kickers < opponent_kickers:
            return 0
        else:
            return 1


# Utility function to map card numbers to names
def get_card_name(card):
    card_dict = {
        11: "J",
        12: "Q",
        13: "K",
        14: "A",
    }
    card_number = card[0]
    card_suit = card[1]
    suit_dict = {
        1: "♠",
        2: "♥",
        3: "♦",
        4: "♣",
    }
```

```
   return f"{card_dict.get(card_number, card_number)}{suit_dict[card_suit]}"
def load_perceptrons(filename="perceptronObjects"):
   try:
       with open(filename, 'rb') as f:
           perceptrons = pickle.load(f)
       return perceptrons
   except FileNotFoundError:
       print("Error", f"Perceptrons file '{filename}' not found. Please train perceptrons first.")
       raise SystemExit(1)  # Exit application gracefully
   except Exception as e:
       print("Error", f"Error loading perceptrons: {str(e)}")
       raise SystemExit(1)  # Exit application gracefully


def main():
   perceptrons = load_perceptrons()
   trials = int(input("How many trials would you like to test:"))
   preflopS = 0
   flopS = 0
   turnS = 0
   riverS = 0
   totalS = 0
   noFoldS = 0
   for i in range(trials):
      # Deal player and opponent hands
       player_table = []
       opponent_table = []
       player_hand = []
       opponent_hand = []
       table = []
       deck = create_deck()
       random.shuffle(deck)

       # Get player's hand
       player_hand = generate_preflop_hand(deck)

       player_table.extend([card for sublist in player_hand for card in sublist])

       # Get opponent's hand
       opponent_hand = generate_preflop_hand(deck)

       hand = np.array(player_table).reshape(1, -1)


       preflop_decision = perceptrons[0].predict(hand)

       # Deal the flop and run perceptron decision
       flop = generate_flop(deck)  # The first three community cards
       table.extend([card for sublist in flop for card in sublist])

       # Use perceptron to decide next step based on flop data
       player_table.extend([card for sublist in flop for card in sublist])
       hand = np.array(player_table).reshape(1, -1)


       flop_decision = perceptrons[1].predict(hand)

       turn = generate_turn(deck)


       table.extend([card for sublist in [turn] for card in sublist])
       player_table.extend([card for sublist in [turn] for card in sublist])
       hand = np.array(player_table).reshape(1, -1)

       turn_decision = perceptrons[2].predict(hand)

       river = generate_river(deck)
```

```
        table.extend([card for sublist in [river] for card in sublist])
        player_table.extend([card for sublist in [river] for card in sublist])
        hand = np.array(player_table).reshape(1, -1)


        river_decision = perceptrons[3].predict(hand)

        # Determine the winner
        winner = compare_poker_hands(player_hand,opponent_hand,table)

        if winner == 1 and preflop_decision == 1 or winner != 1 and preflop_decision != 1:
            preflopS += 1
        if winner == 1 and flop_decision == 1 or winner != 1 and flop_decision != 1:
            flopS += 1
        if winner == 1 and turn_decision == 1 or winner != 1 and turn_decision != 1:
            turnS += 1
        if winner == 1 and river_decision == 1 or winner != 1 and river_decision != 1:
            riverS += 1
        if (winner == 1 and  river_decision == 1 and turn_decision == 1 and flop_decision == 1 and preflop_decision == 1) or (winner !=1 and (river_decision != 1 or winner !=1) and (turn_decision != 1) or (winner !=1 and flop_decision != 1)  or (winner !=1 and preflop_decision != 1)):
            totalS += 1
        if (winner == 1 and  river_decision == 1 and turn_decision == 1 and flop_decision == 1 and preflop_decision == 1):
            noFoldS += 1

    preflopA = preflopS / trials
    flopA = flopS / trials
    turnA = turnS / trials
    riverA = riverS / trials
    totalA = totalS / trials
    totalNoFoldA = noFoldS / trials
    print(f'The preflop Accuracy is {preflopA}')
    print(f'The flop Accuracy is {flopA}')
    print(f'The turn Accuracy is {turnA}')
    print(f'The river Accuracy is {riverA}')
    print(f'The total Accuracy is {totalA}')
    print(f'The total Accuracy with no Fold {totalNoFoldA}')

main()
```

Appendix C:

Member	**Contributions**
Daniel Daugbjerg	- Data Generation Code and Report - Perceptron and data importing Code and Report - Assertion - Appendix A - Background - Conclusion
Kent Morris	- Poker Bot GUI Implementation - Perceptron Accuracy Testing - Accuracy Analysis
Casey Klutznick	- Analysis of results - Accuracy testing - Perceptron training
Thomas Mckeown	- Perceptron parameter tweaking and testing - Time analysis