# Inference for Learning Agents

March 10, 2023

## 1 Introduction

In this project, we will explore the evolution of rationalizable set as the number of auctions grows in the quality-weighted first-price auction and its implications for display advertising.
Furthermore, we will conduct a simulation of a truthful v.s. non-truthful mechanism with learning bidders, and explore the difference between optimized inferred and endowed values in both mechanisms to maximize revenue.

## 2 Preliminaries

### 2.1 Inference of Values

To ensure that ads are shown to the most appropriate audience, advertisers often use demographic information about the user to assign a quality score to each ad. This quality score is then used in a first-price auction, where advertisers bid on the opportunity to have their ad displayed. However, to determine the winner of the auction, the quality score is factored into the bid, creating a quality-weighted first-price auction. This auction mechanism aims to ensure that the ad with the highest quality is displayed to the user, while also providing a fair and efficient way for advertisers to bid on ad space.

In each round of every quality weighted first price auction, Given the demographic information of a user, qualities of ads, and bids from advertisers, we will return the winning bidder and the corresponding price paid in a quality-weighted first-price auction. As such, we set our learning rate to be:

$$\alpha = 0.05$$

With this learning rate, we update the next bid of the advertiser based on the outcome of the previous auction using the exponential average weighted algorithm. Also, we considered the auction for three advertisers, with endowed values [0.8, 0.7, 0.6] respectively.

Lastly, we plotted the regret to see whether our algorithm minimizes the regret over time, and with every round of each auction, computing the rationalizable set over time along with the endowed values to observe the convergence.

### 2.2 Online Revenue Maximization

To estimate if we maximized revenue, we ran an algorithm with this logic:
The simulation randomly endows a group of bidders with values, and then runs a non-truthful mechanism with learning bidders for a specified number of rounds. The bid data is then used to infer the bidders' values, and the mechanism's parameters are optimized to maximize revenue for the inferred values. The bidders' learning algorithms are then reset, and the optimized mechanism is run again for the same number of rounds, allowing for a comparison of the revenue obtained with the revenue obtained from the optimized mechanism for endowed values and the revenue obtained from the mechanism with inferred values.

In this part, we will also be using the truthful second price auction, (also known as a Vickrey auction),

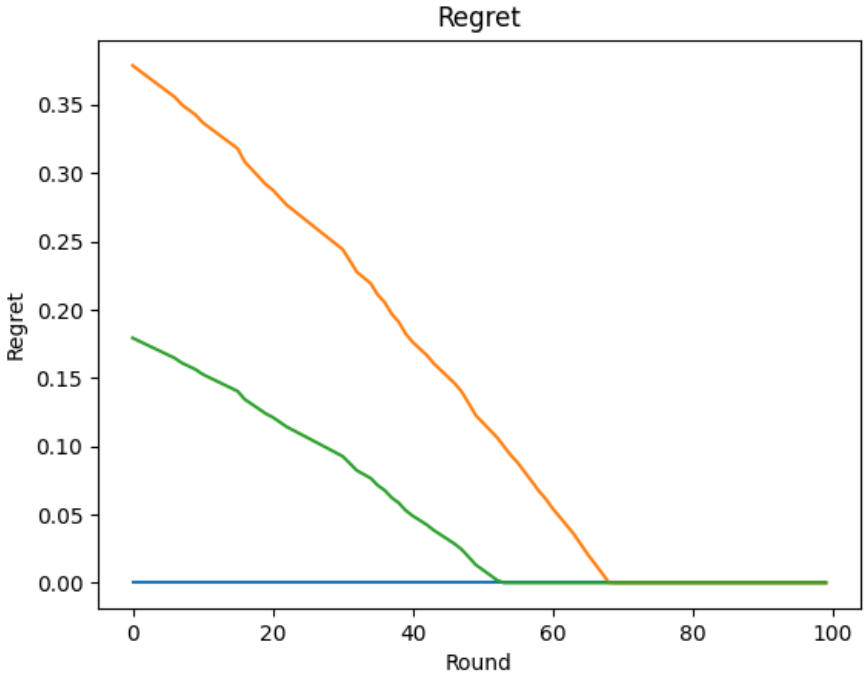
where the highest bidder wins but pays the second-highest bid, to compare with our GSP algorithm (non-truthful algorithm). It will be interesting to observe the difference between truthful and non-truthful algorithms' outcomes.

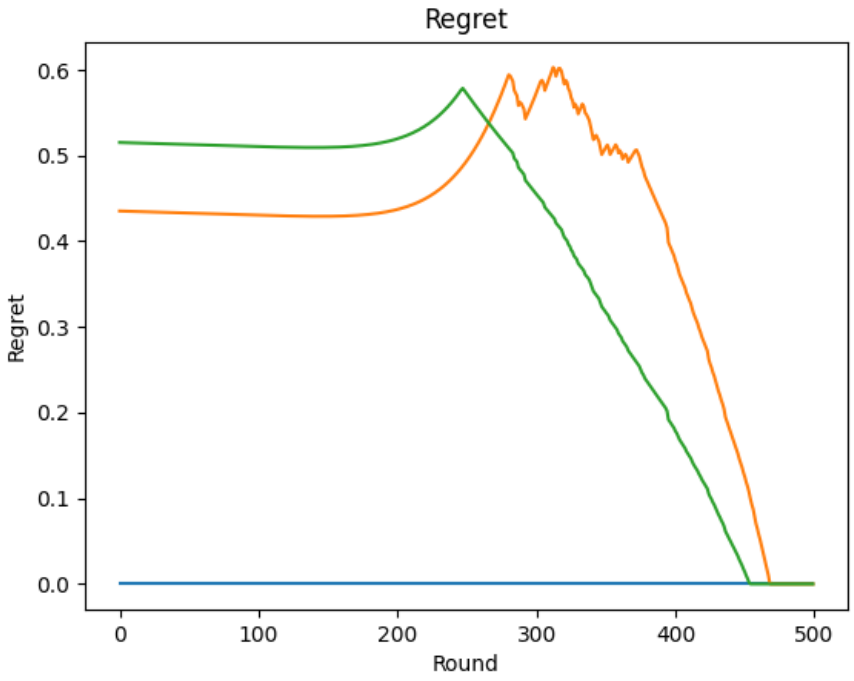
To understand the change over time, we plotted graphs to track the change between the optimized mechanism for endowed values and the revenue obtained from the mechanism with inferred values, which are also really useful for simulating and optimizing non-truthful mechanisms with learning bidders in a variety of applications.

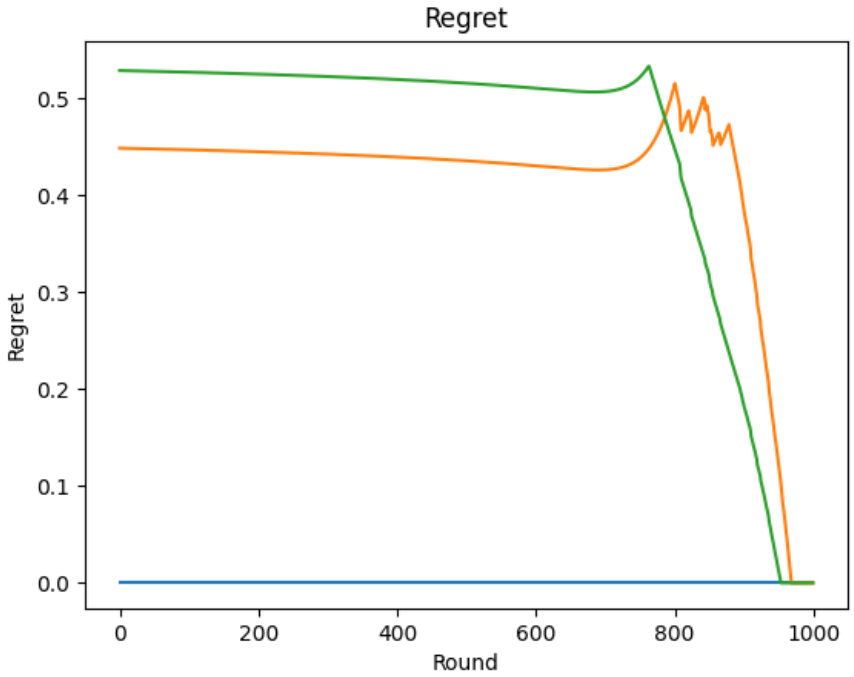
# 3 Results

## 3.1 Inference of Values

To track the regret overtime, we did simulations for rounds = [50, 100, 500, 1000], and these were the results:

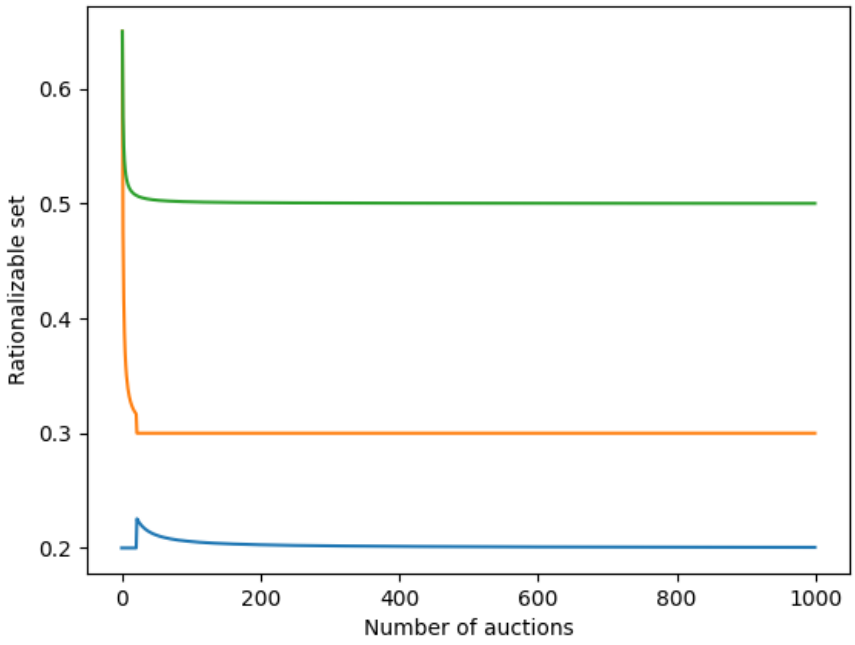




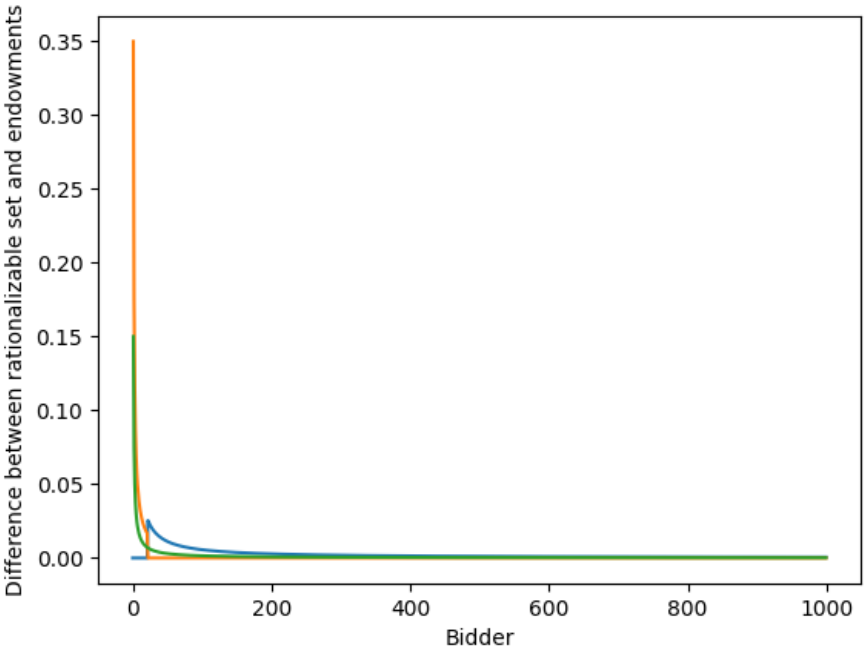




Observing the graph, we can see the regret converges to nearly 0 after rounds.

On the other hand, we can see the rationalizable set converging to the optimal value:




Where we can see the difference between rationalizable set and endowments:



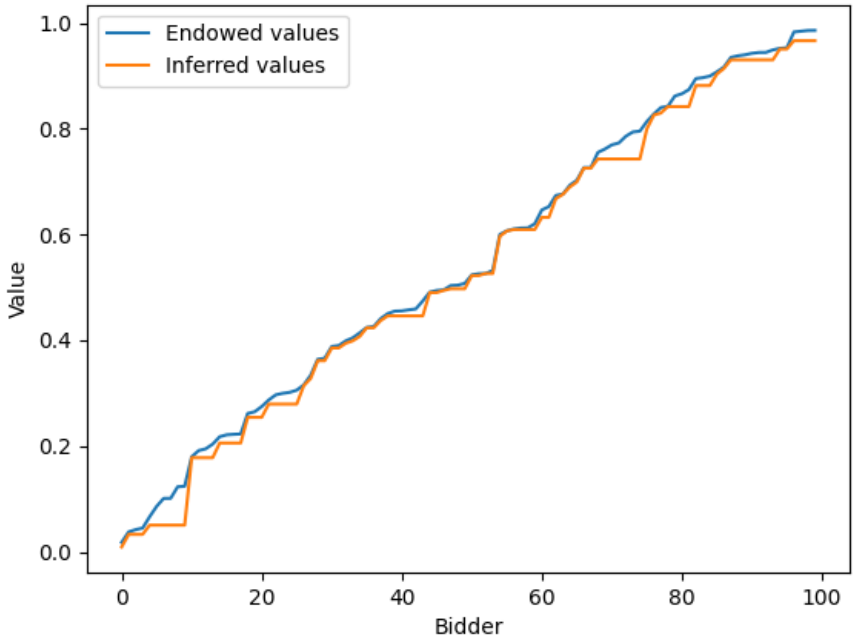

## 3.2 Online Revenue Maximization

### 3.2.1 Truthful mechanism - second price auction

In this part, we generated random values for a group of bidders and ran a second-price auction for a number of rounds. We then use the bid data to infer the bidders' values and optimize the reserve price and learning rate of the mechanism to maximize revenue for the inferred values. This is the result we got after 1000 rounds:

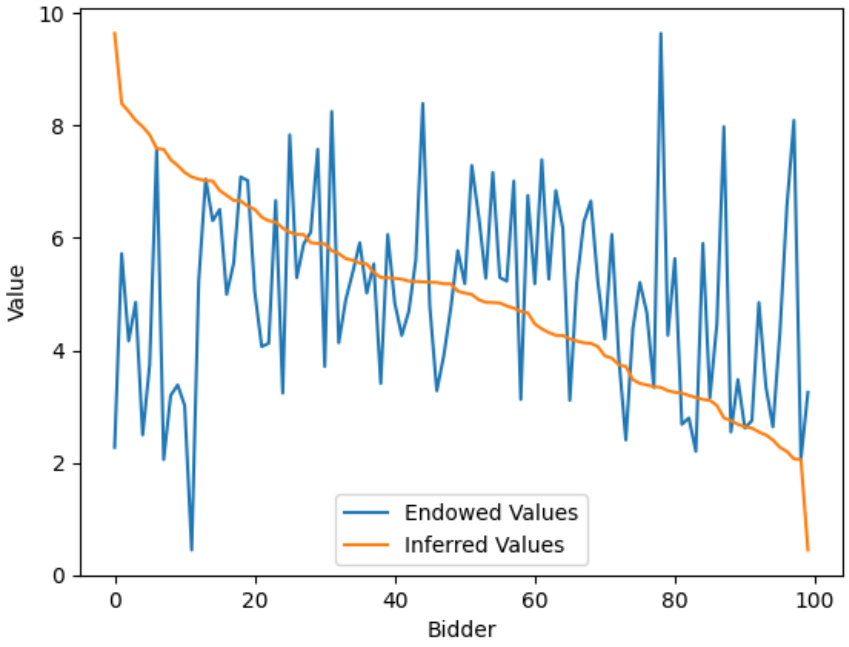
| Revenue obtained by optimizing mechanism (Second-price auction) | Result |
|---|---|
| Inferred values | 2814.8032362067306 |
| Endowed values | 2633.8438592287603 |

Table 1: Table for accumulated revenue obtained overtime.




### 3.2.2 Non-truthful mechanism - GSP auction

In our algorithm, bidders are endowed with values that are drawn from a normal distribution. As the rounds progress, the bids and allocations of the bidders are updated based on their endowed values and the results of previous rounds. However, because the GSP mechanism is non-truthful, bidders may strategically underbid, which can cause fluctuations in the endowed values graph. This is the result:




For example, if a bidder believes that other bidders will submit low bids, they may submit a higher bid in order to win the auction and pay a lower price than their bid. This can lead to fluctuations in the endowed values graph, as the bids and allocations of the bidders are updated based on their endowed values and the results of previous rounds.

# 4 Conclusions

## 4.1 Inference of Values

In conclusion, the quality-weighted first-price auction is a useful mechanism for allocating ad space in display advertising. In our project, we can see that the rationalizable set will converge for different endowed values.

To further discuss point prediction, it may be interesting if we could train a regression model that takes as input the demographic features of the bidders and outputs their bids could be a good idea. We could also impose constraints on the predicted bids to ensure they are reasonable.

## 4.2 Online Revenue Maximization

The difference between the outcomes is truthfulness. The endowed values graph can fluctuate a lot because of the non-truthful nature of the GSP mechanism. In the GSP mechanism, bidders submit bids, and the highest bidders win the slots in descending order of their bids. However, each bidder is charged the bid of the bidder below them, rather than their own bid. This incentivizes bidders to submit bids that are lower than their true value, in order to avoid being charged too much.

Overall, the fluctuations in the endowed values graph are a result of the non-truthful nature of the GSP mechanism and the strategic behavior of the bidders.

# 5 Appendix

```
import random
```

```
import matplotlib.pyplot as plt
import math
import matplotlib.pyplot as plt

class Advertiser:
    def __init__(self, value):
        self.value = value
        self.bid = 0
        self.total_bid = 0
        self.num_wins = 0
        self.rationalizable_set = []
        self.regret_set = []

    def update_bid(self, winner_index, price_paid, epsilon, alpha, t):
        """Update the bid of the advertiser based on the outcome of the previous auction
        if winner_index == self.index:
            self.num_wins += 1
        self.total_bid += self.bid
        self.bid = max(0, self.bid + alpha * (self.value - price_paid) / (epsilon + price

        # Compute the rationalizable set
        max_bid = max([a.bid for a in advertisers if a != self])
        for i in range(t):
            eps = self.get_epsilon(i + 1)
            for v in [0.1, 0.3, 0.5, 0.7, 0.9]:
                bids = [a.bid if a != self else v for a in advertisers]
                regret = max(0, v * (max_bid - self.bid) - eps)
                if regret == 0:
                    self.rationalizable_set.append((v, eps))

    def get_epsilon(self, t):
        """Compute the value of epsilon for the current round."""
        return 1 / (t ** 0.5)


def quality_weighted_first_price_auction(demographic_info, qualities, bids):
    """Given the demographic information of a user, qualities of ads, and bids from adve
    return the winning bidder and the corresponding price paid in a quality-weighted firs
    """
    # Filter the ad qualities and bids based on the demographic information of the user
    ad_qualities = [qualities[i] for i in demographic_info]
    ad_bids = [bids[i] for i in demographic_info]

    # Compute the quality-weighted bids
    weighted_bids = [q * b for q, b in zip(ad_qualities, ad_bids)]

    # Find the index of the highest quality-weighted bid
    winner_index = demographic_info[weighted_bids.index(max(weighted_bids))]

    # Return the winning bidder and the corresponding price paid
    return winner_index, ad_bids[winner_index]


def quality_weighted_first_price_auction_round(advertisers, t, alpha):
    """Run a single round of the quality-weighted first-price auction with regret minimi
    # Generate the qualities of the ads for the current round
```

```
    qualities = [random.uniform(0, 1) for _ in range(len(advertisers))]

    # Compute the bids of the advertisers for the current round
    bids = [advertiser.bid for advertiser in advertisers]

    # Run the auction
    winner_index, price_paid = quality_weighted_first_price_auction(range(len(advertiser

    # Update the bids of the advertisers based on the outcome of the auction
    for advertiser in advertisers:
        advertiser.rationalizable_set = []
        advertiser.update_bid(winner_index, price_paid, advertiser.get_epsilon(t), alpha

    # Return the winner and the price paid in the auction
    return winner_index, price_paid

# Let's say there are three users
# Suppose there are 3 advertisers with endowed values
advertisers = [Advertiser(0.8), Advertiser(0.6), Advertiser(0.7)]

# Set the learning rate and the initial bid to 0
alpha = 0.05
for advertiser in advertisers:
    advertiser.bid = 0
    advertiser.index = advertisers.index(advertiser)

# Run 1000 rounds of the auction
t = 1
for _ in range(100):
    winner_index, price_paid = quality_weighted_first_price_auction_round(advertisers, t
    print("Round", t, "- The winner is advertiser", winner_index, "with a bid of", price

    # Print the regrets of the advertisers
    total_regret = 0
    for advertiser in advertisers:
        v = advertiser.value
        eps = advertiser.get_epsilon(t)
        bid = advertiser.bid
        max_bid = max([a.bid for a in advertisers if a != advertiser])
        regret = max(0, v * (max_bid - bid) - eps)
        advertiser.regret_set.append(regret)
        total_regret += regret
        print("Advertiser", advertisers.index(advertiser), "- Regret:", regret)

    # Compute the average regret and print it
    avg_regret = total_regret / len(advertisers)
    print("Average regret:", avg_regret)

    # Increment the round counter
    t += 1

def compute_point_prediction(rationalizable_sets):
    """Compute the point prediction from the intersection of the rationalizable sets."""
    weighted_values = []
    total_weight = 0
    for rationalizable_set in rationalizable_sets:
```

```
        if len(rationalizable_set) > 0:
            min_epsilon = min([r[1] for r in rationalizable_set])
            for r in rationalizable_set:
                if r[1] <= min_epsilon + 0.01:
                    weight = math.exp(-r[1] ** 2 / 2)
                    weighted_values.append((r[0], weight))
                    total_weight += weight
    if total_weight == 0:
        return None
    else:
        v_prime = sum([w[0] * w[1] / total_weight for w in weighted_values])
        return v_prime

# Print the final bids and total bids of the advertisers
for advertiser in advertisers:
    print("Advertiser", advertisers.index(advertiser), "- Final bid:", advertiser.bid, "-
    # print("Advertiser", advertisers.index(advertiser), "- Rationalizable Set:", advert
    # compute point prediction
    # v_prime = compute_point_prediction(advertiser.rationalizable_set)
    # if v_prime is None:
    #     print("No point prediction can be made.")
    # else:
    #     print(f"Point prediction: {v_prime:.3f}")

#plot the rationalizable set of each advertiser
# for advertiser in advertisers:
#     # for rationalizable_set in advertiser.rationalizable_set:
#     plt.plot(advertiser.rationalizable_set, label = "advertiser"+str(advertisers.index
# plt.title("Rationalizable Set")
# plt.xlabel("Value")
# plt.ylabel("Epsilon")
# plt.show()
# plt.close()

#plot the regret of each advertiser
for advertiser in advertisers:
    #plot the regret set of each advertiser
    print(advertiser.regret_set)
    plt.plot(advertiser.regret_set, label = "advertiser"+str(advertisers.index(advertise
plt.title("Regret")
plt.xlabel("Round")
plt.ylabel("Regret")
plt.show()

import numpy as np
import matplotlib.pyplot as plt

# Function to generate random values for bidders
def generate_values(num_bidders):
    return np.random.uniform(low=0, high=1, size=num_bidders)

# Function to run a second-price auction and return the winning bid and revenue
def run_auction(bids):
    sorted_bids = np.sort(bids)[::-1] # sort bids in decreasing order
    winning_bid = sorted_bids[0]
    second_highest_bid = sorted_bids[1]
```

```
    revenue = second_highest_bid
    return winning_bid, revenue

## Define a function to optimize the mechanism for a given set of bids and values
def optimize_mechanism(bids, values):
    sorted_bids = np.sort(bids)[::-1]
    max_revenue = 0
    for reserve_price in sorted_bids:
        num_bids_above_reserve = np.sum(bids > reserve_price)
        revenue = reserve_price * num_bids_above_reserve
        if revenue > max_revenue:
            max_revenue = revenue
            optimal_reserve_price = reserve_price
    return max_revenue

# def optimize_mechanism(bids, values):
#     reserve_price = 0
#     learning_rate = 0.1
#     for i in range(100):
#         # Compute the optimal reserve price for the given bids and values
#         sorted_values = np.sort(values)[::-1]
#         cutoff_index = np.argmax(sorted_values <= reserve_price)
#         if cutoff_index == 0:
#             optimal_reserve_price = 0
#         else:
#             optimal_reserve_price = sorted_values[cutoff_index - 1]

#         # Update the reserve price and learning rate
#         reserve_price += learning_rate * (optimal_reserve_price - reserve_price)
#         learning_rate *= 0.99

#     return reserve_price

# Endow bidders with values
num_bidders = 100
true_values = generate_values(num_bidders)

# Run the mechanism for n rounds
n = 10000
bids = np.zeros(num_bidders)
for i in range(n):
    for j in range(num_bidders):
        # Generate a bid for the jth bidder based on their value
        bid = np.random.normal(loc=true_values[j], scale=0.1)
        if bid < 0:
            bid = 0
        bids[j] = bid

    # Run the auction and update the bids
    winning_bid, revenue = run_auction(bids)
    for j in range(num_bidders):
        if bids[j] == winning_bid:
            bids[j] = 0

# Infer the bidders' values from the bid data
inferred_values = np.zeros(num_bidders)
```

```
for j in range(num_bidders):
    inferred_values[j] = np.max(bids[bids < true_values[j]])

# Optimize the reserve price and learning rate of the mechanism for the inferred values
optimized_reserve_price = optimize_mechanism(bids, inferred_values)

# Plot the changes in endowed values and inferred values
plt.plot(np.sort(true_values), label='Endowed values')
plt.plot(np.sort(inferred_values), label='Inferred values')
plt.xlabel('Bidder')
plt.ylabel('Value')
plt.legend()
plt.show()

# Reset the learning algorithms of the bidders
learning_rates = np.ones(num_bidders)

# Run the new mechanism on these learning bidders for n rounds
n = 10000
bids = np.zeros(num_bidders)
revenue_sum = 0
for i in range(n):
    for j in range(num_bidders):
        # Generate a bid for the jth bidder based on their learned value
        bid = np.random.normal(loc=inferred_values[j], scale=0.1)
        if bid < 0:
            bid = 0
        bids[j] = bid

    # Run the auction and update the bids
    winning_bid, revenue = run_auction(bids)
    revenue_sum += revenue
    for j in range(num_bidders):
        if bids[j] == winning_bid:
            bids[j] = 0

# Compute the average revenue per round
revenue_per_round = revenue_sum / n

# Compare the revenue obtained in step 5 with the revenue obtained from step 4 and with
revenue_optimized_inferred_values = (num_bidders - 1) * optimized_reserve_price
revenue_optimized_endowed_values = (num_bidders - 1) * optimize_mechanism(bids, true_valu

print('Average revenue per round (optimized for inferred values):', revenue_per_round)
print('Revenue obtained by optimizing mechanism for inferred values:', revenue_optimized
print('Revenue obtained by optimizing mechanism for endowed values:', revenue_optimized_e

import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize_scalar

# Define the GSP non-truthful mechanism
def gsp(bids, ranks):
    revenue = 0
    allocation = np.zeros_like(bids)
    sorted_indices = np.argsort(bids)[::-1]
```

```
    for i in range(len(bids)):
        j = sorted_indices[i]
        if bids[j] >= ranks[i]:
            allocation[j] = 1
            revenue += ranks[i]
        else:
            allocation[j] = 0
    return allocation, revenue

# Define the value distribution
def generate_values(num_bidders):
    return np.random.normal(loc=5, scale=2, size=num_bidders)

# Define the learning algorithm
def learning_algorithm(num_bidders, num_rounds, mechanism):
    values = generate_values(num_bidders)
    bids_history = []
    revenue_history = []
    for t in range(num_rounds):
        # Endow the bidders with values
        bids = values.copy()
        # Run the mechanism
        allocation, revenue = mechanism(bids, values)
        # Record the bids and revenue
        bids_history.append(bids)
        revenue_history.append(revenue)
        # Update the values based on the allocation and revenue
        values += allocation * (values - bids)
    return np.array(bids_history), np.array(revenue_history), values

# Define the optimization function
def optimize(mechanism, bids_history, values):
    def objective(x):
        ranks = np.sort(values)[::-1] * x
        total_revenue = 0
        for i in range(len(bids_history)):
            bids = bids_history[i]
            allocation, revenue = mechanism(bids, ranks)
            total_revenue += revenue
        return -total_revenue / len(bids_history)
    res = minimize_scalar(objective, bounds=(0, 1), method='bounded')
    return res.x * np.sort(values)[::-1]

# Run the learning algorithm with GSP mechanism for 1000 rounds
num_bidders = 100
num_rounds = 1000
bids_history, revenue_history, values = learning_algorithm(num_bidders, num_rounds, gsp)

# Optimize the mechanism for the inferred values
ranks = optimize(gsp, bids_history, values)

# Plot the changes in endowed values and inferred values
plt.plot(values, label='Endowed Values')
plt.plot(ranks, label='Inferred Values')
plt.xlabel('Bidder Index')
plt.ylabel('Value')
```

```
plt.legend()
plt.show()
```