Online Revenue Maximization

February 17, 2023

1 Introduction

In this paper, we will implement and analyze learning algorithms for a second-price auction. In a second-price auction, the highest bidder wins the auction, but only pays the second-highest bid. The seller sets a reserve price, and if no bids are higher than the reserve price, the auction will fail, and the seller will earn zero revenue. The first goal of the learning algorithm is to learn an optimal reserve price that maximizes the seller's expected revenue. On the other hand, we will further simulate and analyze the performance of the Generalized Second Price (GSP) auction. The GSP auction is a widely used mechanism for ad auctions, where advertisers bid on advertising slots, and the auction mechanism determines which advertisements to show to users and what to charge advertisers. The second goal of the project is to simulate the online learning algorithm for the GSP auction and to compare the actual performance of the auction with the theoretical optimal performance.

2 Preliminaries

In order to determine whether the learning algorithm converges to the optimal revenue in the second price auction, the learning algorithm is determined in three different algorithms:

1. Exponential Learning Algorithm determining reserve price
2. Gradient Descent Algorithm determining reserve price

2.1 Second-price Auction

2.1.1 Exponential Weighted Average Algorithm

To implement an exponentially weighted average algorithm to determine the reserve price, we updated the reserved price based on the value of each bidder in each round. In our algorithm, we have α and τ, and calculating the learned reserve price using the formula:

$$reservedPrice = (1 - \alpha)reservedPrice + \alpha(value - (1/\tau))$$

We will calculate the expected revenue for each round and plot the evolution of the expected revenue over time. On the other hand, the optimal reserve price plots the expected revenue based on the optimal reserve price for comparison.

2.1.2 Gradient Descent Algorithm

To consider a more comprehensive algorithm that calculates the updated learned reserved price not just exponentially, we decided to calculate it with gradient descent.
Suppose we have an equation:

$$y = mx + b$$

Then m and b will be its parameters, during the recalculating process, there will be a small change in their values. Let that small change be denoted by δ. The value of parameters will be updated as m=m-δm and b=b-δb, respectively. In our Gradient Descent Algorithm, we implement this strategy

by computing the derivative of the revenue function with respect to the reserve price and then updating the reserve price using the calculated derivative.

$$derivative = \sum_{i}^{n} V_i - ReservedPrice_i$$

$$reservedPrice = reservedPrice - ReservedPrice * derivative$$

As we use the gradient descent algorithm to update the reserve price and compute the expected revenue over time, we are able to analyze whether the result converges to the optimal revenue.
The performance of the learning algorithm is then compared to the optimal reserve price for different learning rates.

2.2 Generalized Second Price Auction

In this section, we will be focusing on Position Auctions. We will implement an online learning algorithm for the Generalized Second Price (GSP) auction for this scenario. We will be analyzing be our online GSP function, and comparing it to the GSP auction.

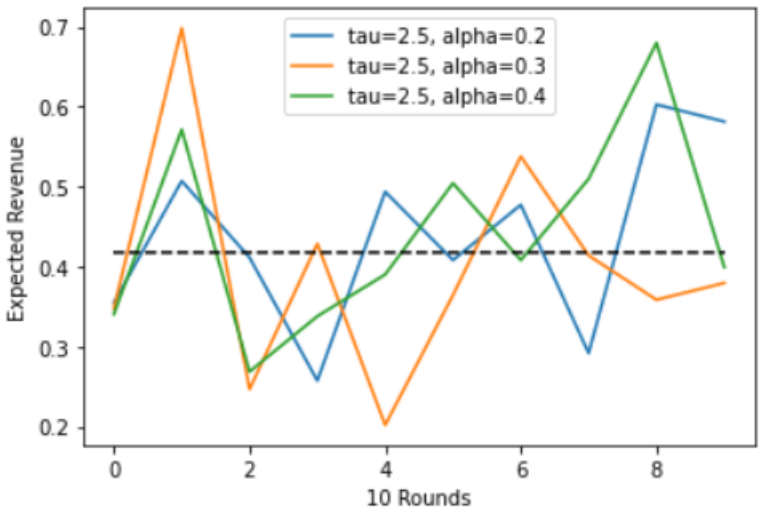
The GSP auction is an auction mechanism used to sell online advertising positions. Our implementation is the online GSP auction, we will compare our online learning algorithm with the GSP auction. Lastly, we will compare the revenue we generated in each round and the theoretical optimal revenue.

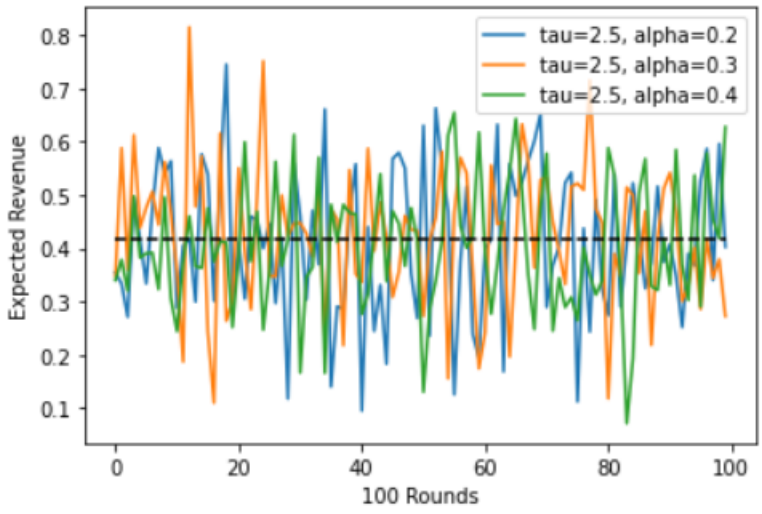
3 Results

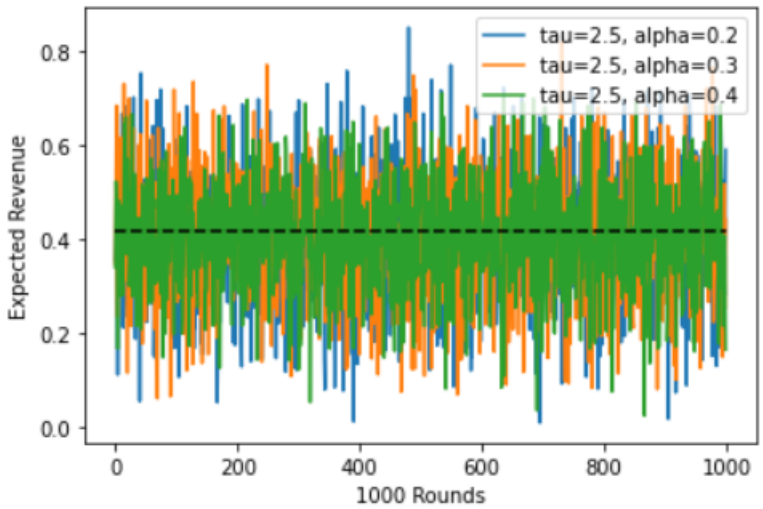
3.1 Second-price Auction

3.1.1 Exponential Weighted Average Algorithm

We tested various α and τ across our exponential average weights learning algorithms as learning rates against each other and found that within this scenario.



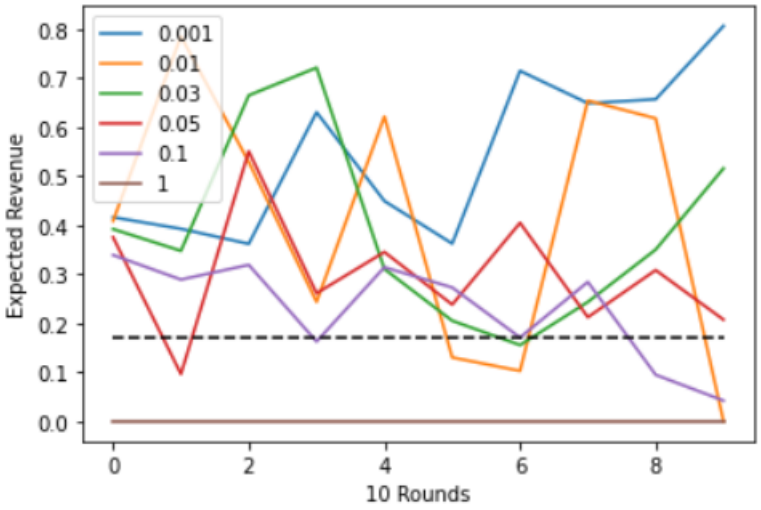


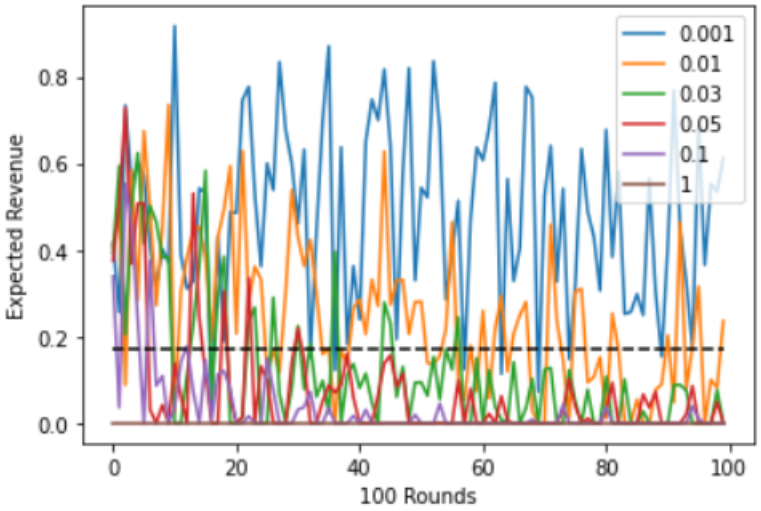


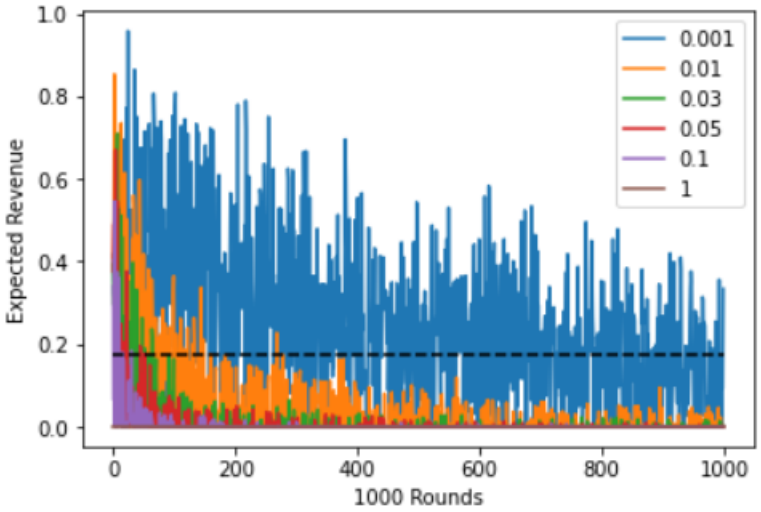
Over rounds of simulations, we can see the result of increasing rounds slowly converging to the optimal revenue.

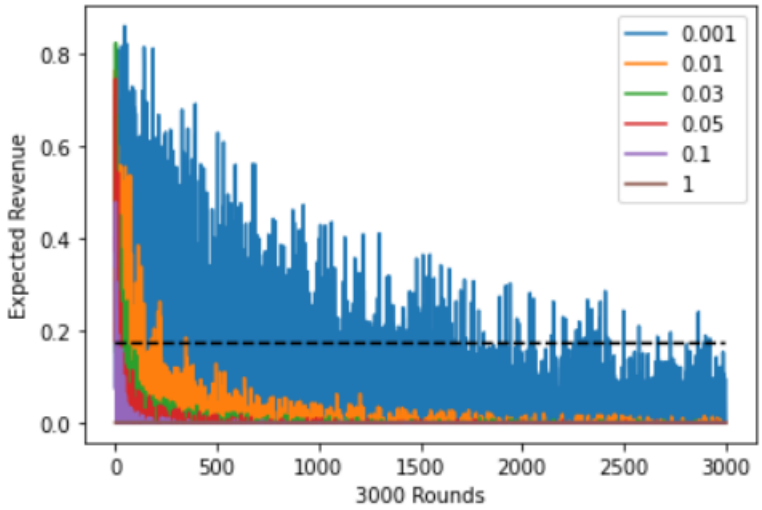
3.1.2 Gradient Descent Algorithm

Given the exponentially weighted average was rather slow, we decided to test upon using gradient descent, and this was the result:


0.001
0.01
0.03
0.05
0.1
1
Expected Revenue
10 Rounds


0.001
0.01
0.03
0.05
0.1
1
Expected Revenue
100 Rounds

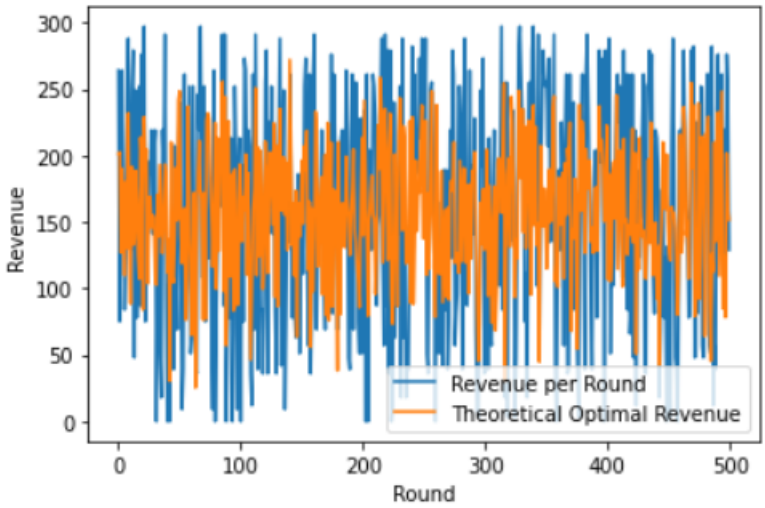

0.001
0.01
0.03
0.05
0.1
1
Expected Revenue
1000 Rounds



As we know that gradient descent minimizes a value by a small fraction, the best rounds that we obtained converging to the expected revenue would be 1000 for our learning algorithm.

3.2 Generalized Second Price Auction

On the other hand, to test our online GSP auction, we plotted out output compared the gsp auction:



Average difference between online revenue and theoretical optimal revenue: 34.264746666

In fact, we see our online revenue algorithm doing quite well over 500 rounds, converging to the GSP auction expected.

4 Conclusion

4.1 Second-price Auction

Overall, we could see that the Exponential Weighted Average and Gradient Descend Algorithm converge to the theoretical optimal revenue over 500 rounds. Although, it is a bit difficult to say which learning algorithm is generally better since they could perform differently across various situations, but over multiple trials, the exponential weights seems to always perform a bit worse at the beginning which could possibly be alleviated with a higher learning rate. Tuning the τ value to be 2.5 performed the best in our exponential learning algorithm.

As for Gradient Descent, having to fine-tune the algorithm causes the learning rate to decrease, which

slows down the algorithm obtaining the revenue converging to the optimal revenue. Other algorithms can be adapted in the descent algorithm to increase the speed of the auction which is an interesting part to learn for our project.

4.2 GSP Auction

Based on our algorithm, we can conclude that the online GSP auction with the proposed algorithm performs reasonably well in terms of generating revenue. The simulation shows that the revenue generated per round is comparable to the theoretical optimal revenue. However, the average difference between the revenue generated and the theoretical optimal revenue is around 64.26, which indicates that there is room for improvement in the algorithm. For instance, our current algorithm does not balance exploration and exploitation. It always bids based on the empirical virtual values, which can be suboptimal if the true values are different. It will be interesting to see whether our algorithm improves if we occasionally bid randomly to explore the true values, while still exploiting the empirical virtual values most of the time.

5 Appendix

```
import numpy as np
import matplotlib.pyplot as plt
import random

# set values of bidder values
def value_distribution(n):
#   print( [random.uniform(0, 1) for i in range(n)])
  return [random.uniform(0, 1) for i in range(n)]

# calculate virtual welfare given current reserve price
def virtual_welfare(reserve_price, value_distribution):
    return sum([value - reserve_price for value in value_distribution if value > reserve

# calculate expected revenue given current reserve price
def expected_revenue(reserve_price, value_distribution):
    n = len(value_distribution)
    vw = sum([value - reserve_price for value in value_distribution if value > reserve_p
    return (1/n) * vw

# implementation of exponentially weighted average algorithm
def exponential_weighted_average(reserve_price, value, tau, alpha):
    return (1 - alpha) * reserve_price + alpha * (value - (1 / tau))

# calculate optimal reserve price for the distribution
def optimal_reserve_price(value_distribution):
    return sum(value_distribution) / len(value_distribution)

# compare performance of learning algorithm to optimal reserve price
def compare_performance(vd, tau, alpha, num_round):
    # initialize reserve price
    reserve_price = 0

    # calculate optimal reserve price
    # optimal_reserve = optimal_reserve_price(value_distribution)
    optimal_reserve = 0.5

    # calculate optimal expected revenue
    # optimal_expected_revenue = expected_revenue(optimal_reserve, value_distribution)
```

```
    optimal_expected_revenue = 5/12

    # initialize list to store expected revenue over time
    expected_revenue_list = []

    # calculate expected revenue over time
    for i in range(num_round):
        # update reserve price using exponentially weighted average algorithm
        arr = vd
        vd.sort(reverse=True)
        reserve_price = exponential_weighted_average(reserve_price, vd[1], tau, alpha)
        # print("reserve_price", reserve_price)
        expected_revenue_list.append(expected_revenue(reserve_price, arr))
        vd =value_distribution(num_bidders)
        # print("value_distribution", vd)
    # print("count", count)
    return expected_revenue_list, optimal_expected_revenue

# number of bidders
num_bidders = 3
num_rounds = [10, 100, 1000, 3000]


vd =value_distribution(num_bidders)
# set the range of values for tau and alpha
taus = [2, 2.5, 3, 3.5]
alphas = [0.2, 0.3, 0.4]

# initialize dictionary to store expected revenue for each tau and alpha
revenues = {}

# compare performance of learning algorithm to optimal reserve price for each tau and alp
for num_round in num_rounds:
    for tau in taus:
        for alpha in alphas:
            expected_revenue_list, optimal_expected_revenue = compare_performance(vd, tau
            plt.plot(expected_revenue_list, label=f"tau={tau}, alpha={alpha}")
            plt.xlabel(str(num_round) +" Rounds")
            plt.ylabel("Expected Revenue")
            plt.legend()
            revenues[(tau, alpha)] = expected_revenue_list
        plt.plot( [optimal_expected_revenue]*num_round, 'k--', label='optimal expected re
        plt.show()

# implementation of gradient descent algorithm
def gradient_descent(reserve_price, value_distribution, learning_rate):
    n = len(value_distribution)
    derivative = (-2 / n) * sum([value - reserve_price for value in value_distribution i
    return reserve_price - learning_rate * derivative

# calculate optimal reserve price for the distribution
def optimal_reserve_price(value_distribution):
    return sum(value_distribution) / len(value_distribution)

# compare performance of learning algorithm to optimal reserve price
def compare_performance(vd, learning_rate, num_round):
```

```
    # initialize reserve price
    reserve_price = 0

    # calculate optimal reserve price
    optimal_reserve = optimal_reserve_price(vd)

    # calculate optimal expected revenue
    optimal_expected_revenue = expected_revenue(optimal_reserve, vd)

    # calculate expected revenue over time
    for i in range(num_round):
        # update reserve price using gradient descent algorithm
        reserve_price = gradient_descent(reserve_price, vd, learning_rate)

        # calculate expected revenue
        expected_revenue_list.append(expected_revenue(reserve_price, vd))
        vd = value_distribution(num_bidders)


    return expected_revenue_list, optimal_expected_revenue

# true value distribution of bidders
vd = value_distribution(num_bidders)

# set the range of values for learning rate
learning_rates = [0.001,0.01,0.03,0.05, 0.1,1]
# initialize dictionary to store expected revenue for each learning rate
revenues = {}


# compare performance of learning algorithm to optimal reserve price for each learning r
for num_round in num_rounds:
    for learning_rate in learning_rates:
        # initialize list to store expected revenue over time
        expected_revenue_list = []
        expected_revenue_list, optimal_expected_revenue = compare_performance(vd, learni
        revenues[learning_rate] = expected_revenue_list
        plt.plot(expected_revenue_list, label=str(learning_rate))
        plt.xlabel(str(num_round) +" Rounds")
        plt.ylabel("Expected Revenue")
        plt.legend()
    plt.plot( [optimal_expected_revenue]*num_round, 'k--', label='optimal expected revenu
    plt.show()

import numpy as np
import matplotlib.pyplot as plt

def gsp_auction(bids, click_probs):
    """
    A function to implement the Generalized Second Price (GSP) auction.
    The inputs are the bids submitted by the bidders and the click probabilities for eac
    The function returns the revenue generated by the auction.
    """
    positions = np.argsort(-np.array(bids))
    revenue = 0
    for j in range(len(bids)):
```

```
        w = click_probs[j]
        if len(bids) > 1:
            second_highest_bid = bids[-2]
        else:
            second_highest_bid = 0
        revenue += w * second_highest_bid
    return revenue

def online_gsp_auction(m, value_dists, num_rounds):
    """
    A function to implement the online learning algorithm for the GSP auction.
    The inputs are:
        m: the number of positions
        value_dists: a list of value distributions for each position
        num_rounds: the number of rounds to run the algorithm for
    The function returns the revenue generated in each round and the theoretical optimal
    """
    revenue_per_round = np.zeros(num_rounds)
    theoretical_optimal_revenue = np.zeros(num_rounds)
    bids_per_round = []
    for i in range(num_rounds):
        # Draw bids from the value distributions
        bids = [np.random.choice(value_dists[j], size=1)[0] for j in range(m)]
        bids_per_round.append(bids)
        click_probs = np.arange(m, 0, -1) / m
        revenue_per_round[i] = gsp_auction(bids, click_probs)
        # Calculate the empirical virtual values
        virtual_values = np.zeros(m)
        for j in range(m):
            # Determine the set of bidders that could win position j
            eligible_bidders = [bids[k] for k in range(j, m)]
            virtual_values[j] = np.mean(eligible_bidders)

        # Sort the virtual values
        virtual_values = np.sort(virtual_values)[::-1]

        # Calculate the theoretical optimal revenue for this round
        theoretical_optimal_revenue[i] = sum(virtual_values[j] * (m - j) / m for j in ra

    return revenue_per_round, theoretical_optimal_revenue

# Run the simulation
m = 5
value_dists = [np.random.randint(0, 100, size=100) for i in range(m)]
num_rounds = 500
revenue_per_round, theoretical_optimal_revenue = online_gsp_auction(m, value_dists, num_

# Plot the results
plt.plot(revenue_per_round, label='Revenue per Round')
plt.plot(theoretical_optimal_revenue, label='Theoretical Optimal Revenue')
plt.xlabel('Round')
plt.ylabel('Revenue')
plt.legend()
plt.show()

# Measure the extent to which the online performance matches the theoretical guarantee
```

```
difference = np.abs(revenue_per_round - theoretical_optimal_revenue)
print("Average difference between online revenue and theoretical optimal revenue:", np.m
```