MS&E 211
David Newcomb
Edward Dai
Travis Noll
Albert Chen

Question 6

For the given utility functions U(1) and U(2), run the market with model (3), separately with the parameter 0 < param *< 1. Compare the results generated from the two on-line models and their off-line call auction models using simulation data. Which one accepts more bids and which one has better prediction power?*

For the **on-line auction models**, I wrote MATLAB code to solve the optimization problem each time a bid is submitted, and specifically did so for both the exponential utility function and the logarithmic utility function. I derived my code directly from Lecture Note #15, Slide 11. I imported the data from our class Excel extra credit game bids of the following games: Arizona, California, Colorado, Notre Dame, Oregon, Oregon State, UCLA, and Washington State. The code keeps track of how many bids have been accepted and rejected, in addition to per bid updates on the new price (p) and bid (b) vectors. The code for this is in **Appendix A**. In this specific experiment, $m = 8$, $param = 0.5$. The results of the program are as follows:

LOGARITHMIC UTILITY FUNCTION: $U(s) = param/m(\sum log(s\_i))$

	Bids Accepted	Bids Rejected
Arizona	127	0
California	15	60
Colorado	7	74
Notre Dame	15	112
Oregon	5	50
Oregon State	88	0
UCLA	24	54
Washington State	101	0
TOTAL	**382**	**350**

EXPONENTIAL UTILITY FUNCTION: $U(s) = b/m(\sum 1 - exp(-s\_i))$

	Bids Accepted	Bids Rejected
Arizona	9	118
California	4	71
Colorado	5	76
Notre Dame	8	119
Oregon	5	50
Oregon State	3	85
UCLA	3	75
Washington State	10	91
TOTAL	**47**	**685**

Additionally, here are the per game simulation results in terms of predictive power. The states noted in the 1st column are the states which garnered the highest prices at the end of the simulation – the shared price for all these states is in parenthesis. The state in the 2nd column is the winning state.. The 3rd column indicates whether the winning state was among the highest valued states.

LOGARITHMIC UTILITY FUNCTION: $U(s) = param/m(\sum log(s\_i))$

	Highest Price States	Winning State	Y:N
Arizona	1, 2, 3, 4, 8 – (0.2)	5	N
California	2, 3, 4, 5, 6, 7 – (.1667)	7	Y
Colorado	1, 2, 8 – (.3167)	8	Y
Notre Dame	1, 2, 3 – (.3167)	4	N

Oregon	2, 3 – (.4750)	5	N
Oregon State	1, 2, 3, 5, 6, 7, 8 – (.1429)	5	Y
UCLA	2, 3 – (.36)	7	N
Washington State	1, 2, 3, 4 – (.25)	5	N
TOTAL			**3:5**

EXPONENTIAL UTILITY FUNCTION: $U(s) = b/m(\sum 1 - exp(-s\_i))$

	Highest Price States	Winning State	Y:N
Arizona	5, 6, 7 – (.3333)	5	Y
California	1, 8 – (.5)	7	N
Colorado	1, 2, 8 – (.3333)	8	Y
Notre Dame	1, 2, 3 – (.3333)	4	N
Oregon	2, 3 – (.4750)	5	N
Oregon State	3 – (1.00)	5	N
UCLA	2, 3 – (.4992)	7	N
Washington State	5, 6, 7, 8 – (.25)	5	Y
TOTAL			**3:5**

For the **off-line models,** whose objective function is non-linear given that our utility functions are non-linear, I used the CVX solver (*http://www.stanford.edu/~boyd/cvx*). With the CVX solver, I found optimal objective function values for both the exponential and logarithmic models. The code for this is in **Appendix B**.

CVX OPTIMAL VALUES

	Log Utility Function	Exp Utility Function
Arizona	+1081.38	+1081.28

California	+421.822	+422.994
Colorado	+264.365	+264.837
Notre Dame	+405.033	+405.184
Oregon	+69.3689	+70.259
Oregon State	+203.835	+204.922
UCLA	+216.745	+217.874
Washington State	+200.309	+201.093
TOTAL	**+2862.8579**	**+2868.443**

Conclusions

Comparing the two on-line models, it is clear that *U(1)* accepts far more bids (near the order of 10x more bids), likely because it is prone to taking on more risk than *U(2)*. Given that the logarithmic utility function is monotonic and increasing, the worst-case loss represented by *U(1)* is in fact unbounded (approaches ∞). On the other hand, our exponential utility function only approaches a worst-case loss of $(b/m)\log(N)$. For more information on worst-case losses of such utility functions, refer to source [3].

Regarding predictive power, there is no clear winner between the two on-line models. Both predicted the same number of outcomes correctly in our admittedly small sample size. Nevertheless, *U(2)* was able to do so in a more clear, efficient manner. Rather than having the most probable outcomes dispersed among 6 or 7 of the 8 states like *U(1)*, we achieved the same result with narrowing that field consistently down to 3 or 4 probable states. Therefore, in some sense *U(2)* acted with a better, or more efficient, predictive power.

Now let's turn our attention to the off-line models. The objective values obtained from the CVX optimization of the two off-line models were strikingly similar. There are only minor discrepancies, and on the whole, the market-maker turns quite a nice profit, only coming somewhat close to dipping under in the Oregon game (NOTE: the fortuitous result of this game was highly unexpected which may have contributed to the lower earnings).

Given the similarity between optimal values for the offline exponential and logarithmic utility functions, and the large difference between bids accepted from the on-line models, I have to conclude that the on-line model is much more susceptible to risk. Otherwise, we would see a similar number of bids accepted in the on-line models as well as similar optimal values in the off-line models. A few early bids in the on-line logarithmic model specifically can significantly change the price vector in such a manner that the market-maker will accept later bids that will ultimately be bad for business. This simply highlights the grand importance of choosing a utility function with a minimal and strictly bounded worst-case loss. High risk, high reward does not always pay off.

Appendix A

What follows is simply the MATLAB code for solving the on-line formulations of both the logarithmic (Appendix A) and exponential (Appendix B) utility function auction models. The initialization needed for both solvers ironically comes last (Appendix C).

LOGARITHMIC UTILITY FUNCTION

- **logAuction.m**

```
% This function will solve one iteration of the optimization problem, given a bid (a), a
price limit (pi) and a quantity limit (qt) among other given parameters.  AcceptCounter
counts the number of accepted bids and rejectCounter does the same for rejected bids.
Param must be between 0 and 1.  M is the number of states, p is the price vector, and b
is the bid vector.
function [b, p, acceptCounter, rejectCounter] =
logAuction(p,b,param,m,a,pi,qt,acceptCounter,rejectCounter)

% Step 1
if(a*p > pi)
    rejectCounter(:) = rejectCounter(:) + 1;
else
    % Step 2
    % Setting up the equation.
    syms y ybar;
    sumOneInStepTwo = 0;

    for (n=1:length(a))
        ait = a(n);
        bitminus = b(n);
        eqn = ((param/m)*(1/(y - ait*qt - bitminus)));
        sumOneInStepTwo = sumOneInStepTwo + eqn;
    end
    Y = solve(sumOneInStepTwo == 1);

    for(n=1:length(Y))
        if Y(n) > (a(n)*qt - b(n));
            ybar = Y(n);
        end
    end
    sumTwoInStepTwo = 0;

    for (n=1:length(a))
        ait = a(n);
        bitminus = b(n);
        eqn = (ait*(param/m)*(1/(ybar - ait*qt - bitminus)));
        sumTwoInStepTwo = sumTwoInStepTwo + eqn;
    end
    % Check to see if our optimal ybar is acceptable or not
    if (pi > sumTwoInStepTwo)
        x = qt;
        bnew = b + a.'*qt;
        for (n=1:length(p));
            bit = bnew(n);
            p(n) = (param/m)*(1/(ybar - bnew(n)));
        end
        acceptCounter = acceptCounter + 1;
        b = bnew;
        % If ybar is not acceptable move on to Step 3  + pi*x - ybar +
    else
        % Step 3
        syms x y;
```

```
            sumOneInStepThree = 0;

            % Solve for x in terms of y
            for (n=1:length(a))
                ait = a(n);
                bitminus = b(n);
                eqn = ((param/m)*(1/(y - ait*x - bitminus)));
                sumOneInStepThree = sumOneInStepThree + eqn;
            end

            Y = solve(sumOneInStepThree == 1, x);
            x = Y;
            % Solve for y outright
            sumTwoInStepThree = 0;

            for (n=1:length(a))
                ait = a(n);
                bitminus = b(n);
                eqn = (ait*(param/m)*(1/(y - ait*x - bitminus)));
                sumTwoInStepThree = sumTwoInStepThree + eqn;
            end

            Y = solve(sumTwoInStepThree == pi, y);
            y = Y;
            % Solve for x outright
            syms x;
            sumThreeInStepThree = 0;

            for (n=1:length(a))
                ait = a(n);
                bitminus = b(n);
                eqn = ((param/m)*(1/(y - (ait)*x - bitminus)));
                sumThreeInStepThree = sumThreeInStepThree + eqn;
            end

            Y = solve(sumThreeInStepThree == 1, x);
            x = Y;
            % Update variables accordingly
            bnew = b + a.'*x;
            for (n=1:length(p))
                bit = bnew(n);
                p(n) = (param/m)*(1/(y - bnew(n)));
            end
            acceptCounter = acceptCounter + 1;
            b = bnew;
        end
    end
    end
```

- **logAuctionSolver.m**

```
% This function will take our input data from the class game, and run through each bid,
resolving the optimization problem each time, resulting in new price and bid vectors all the
while.
function [b p acceptCounter rejectCounter] =
logAuctionSolver(p,b,param,m,a,pi,qt,acceptCounter,rejectCounter,priceVector,quantityVector,
A)

for (n=1:length(priceVector));
    pi = priceVector(n);
    qt = quantityVector(n);
    a = A(1,:);
    [b p acceptCounter rejectCounter objectiveValue] =
logAuction(p,b,param,m,a,pi,qt,acceptCounter,rejectCounter);
end
p
acceptCounter
rejectCounter
```

Appendix B

EXPONENTIAL UTILITY FUNCTION

- **expAuction.m**

```
% This function will solve one iteration of the optimization problem, given a bid (a), a
price limit (pi) and a quantity limit (qt) among other given parameters.  AcceptCounter
counts the number of accepted bids and rejectCounter does the same for rejected bids.
Param must be between 0 and 1.  M is the number of states, p is the price vector, and b
is the bid vector.

function [b, p, acceptCounter, rejectCounter] =
expAuction(p,b,param,m,a,pi,qt,acceptCounter,rejectCounter)
% Step 1
if(a*p > pi)
    rejectCounter(:) = rejectCounter(:) + 1;
else
    % Step 2
    % Setting up the equation.
    syms y;
    sumOneInStepTwo = 0;

    for (n=1:length(a))
        ait = a(n);
        bitminus = b(n);
        eqn = ((param/m)*exp(-1*(y - ait*qt - bitminus)));
        sumOneInStepTwo = sumOneInStepTwo + eqn;
    end

    Y = solve(sumOneInStepTwo == 1);

    for(n=1:length(Y))
        if Y(n) > (a(n)*qt - b(n));
            ybar = Y(n);
        end
    end

    sumTwoInStepTwo = 0;

    for (n=1:length(a))
        ait = a(n);
        bitminus = b(n);
        eqn = (ait*(param/m)*exp(-1*(ybar - ait*qt - bitminus)));
        sumTwoInStepTwo = sumTwoInStepTwo + eqn;
    end

    % Check to see if our optimal ybar is acceptable or not
    if (pi > sumTwoInStepTwo)
        x = qt;
        b = b + a.'*qt;
        for (n=1:length(p));
            bit = b(n);
            p(n) = (param/m)*exp(-1*(ybar - b(n)));
        end
        acceptCounter = acceptCounter + 1;

        % If ybar is not acceptable move on to Step 3
    else
        % Step 3
        syms x y;
        sumOneInStepThree = 0;

        % Solve for x in terms of y
        for (n=1:length(a))
            ait = a(n);
```

```
            bitminus = b(n);
            eqn = ((param/m)*exp(-1*(y - ait*x - bitminus)));
            sumOneInStepThree = sumOneInStepThree + eqn;
        end

        Y = solve(sumOneInStepThree == 1, x);
        x = Y;

        % Solve for y outright
        sumTwoInStepThree = 0;

        for (n=1:length(a))
            ait = a(n);
            bitminus = b(n);
            eqn = (ait*(param/m)*exp(-1*(y - ait*x - bitminus)));
            sumTwoInStepThree = sumTwoInStepThree + eqn;
        end

        Y = solve(sumTwoInStepThree == pi, y);
        y = Y;

        % Solve for x outright
        syms x;
        sumThreeInStepThree = 0;

        for (n=1:length(a))
            ait = a(n);
            bitminus = b(n);
            eqn = ((param/m)*exp(-1*(y - ait*x - bitminus)));
            sumThreeInStepThree = sumThreeInStepThree + eqn;
        end

        Y = solve(sumThreeInStepThree == 1, x);
        x = Y;

        % Update variables accordingly
        b = b + a.'*x;
        for (n=1:length(p))
            bit = b(n);
            p(n) = (param/m)*exp(-1*(y - b(n)));
        end
        acceptCounter = acceptCounter + 1;
    end
end
end
```

- **expAuctionSolver.m**

```
% This function will take our input data from the class game, and run through each bid,
resolving the optimization problem each time, resulting in new price and bid vectors all the
while.

function [b p acceptCounter rejectCounter] =
expAuctionSolver(p,b,param,m,a,pi,qt,acceptCounter,rejectCounter,priceVector,quantityVector,
A)

for (n=1:length(priceVector));
    pi = priceVector(n);
    qt = quantityVector(n);
    a = A(1,:);
    [b p acceptCounter rejectCounter] = expAuction(p, b, param, m, a, pi, qt, acceptCounter,
rejectCounter);
end
p
acceptCounter
rejectCounter
```

Appendix C

```
% Initializing from class data
% file = 'dataARIZONA.xlsx' -- This could be any of the relevant Excel data
% sets, and needs to be called in order to make the variable below, data,
% meaningful.

% could be the data from any game
file = 'dataND.xlsx'
data = xlsread(file);
priceVector = data(:,2)
quantityVector = data(:,1)
A = data(:, 3:10)
m = 8
param = .5
p = ones(m,1)/m
b = ones(m,1)
rejectCounter = 0
acceptCounter = 0

% Given the setup, here are the lines that would need to be called to solve
% our call auction online model for the ARIZONA data, returning how many
% bids were accepted, how many were rejected, and what the final price
% vector was:

      %InitializeAuction
      %expAuctionSolver(p,b,param,m,a,pi,qt,acceptCounter,rejectCounter,priceVector,q
      uantityVector,A)
      %logAuctionSolver(p,b,param,m,a,pi,qt,acceptCounter,rejectCounter,priceVector,q
      uantityVector,A)

% This code below would create one's own simulation data according to three
% different belief groups that would assign a certian belief value to each
% state.  This example is built for 5 states only and could easily be extended.

%priceVector = zeros(100,1)
%quantityVector = randi(100,[100,1])
%A = zeros(100,5);
%for n=1:100
%a = [randi([0 1], 1, 8) ones(1,8)];
%a = a(randperm(5));
%A(n,:) = a;
%end
%A

%for(n=1:33)
%   priceVector(n) = A(n,:)*[.2+(.4-.2)*rand(1,1); .1*rand(1,1); .1*rand(1,1); .2+(.
3-.2)*rand(1,1); .1*rand(1,1)]
%end

%for(n=33:66)
%   priceVector(n) = A(n,:)*[.1*rand(1,1); .1+(.4-.1)*rand(1,1); .1+(.2-.
1)*rand(1,1); .2*rand(1,1); .1*rand(1,1)]
%end

%for(n=67:length(priceVector))
%   priceVector(n) = A(n,:)*[.2*rand(1,1); .2*rand(1,1); .1*rand(1,1); .
1*rand(1,1); .3+(.4-.3)*rand(1,1)]
%end
```