(SMM641) - Revenue Management & Pricing. Quantity Based Revenue Management Part 2 - R Supplement

Oben Ceryan

Contents

1 Dynamic Programming - An Introductory Example

For description, please see lecture slides. Consider the following game:

- Setup: A pile of 20 toothpicks
- Playing against a computer
- Game consists of rounds. The sequence of events is as follows:
  - You start first. You can pick either one or two toothpicks from the pile.
  - Computer moves next. Picks one with probability 0.5 and picks two with prob 0.5.
  - Game proceeds until all toothpicks are removed from the pile.
- If you hold the last toothpick, you win and receive £20. Otherwise the computer wins and you get nothing.

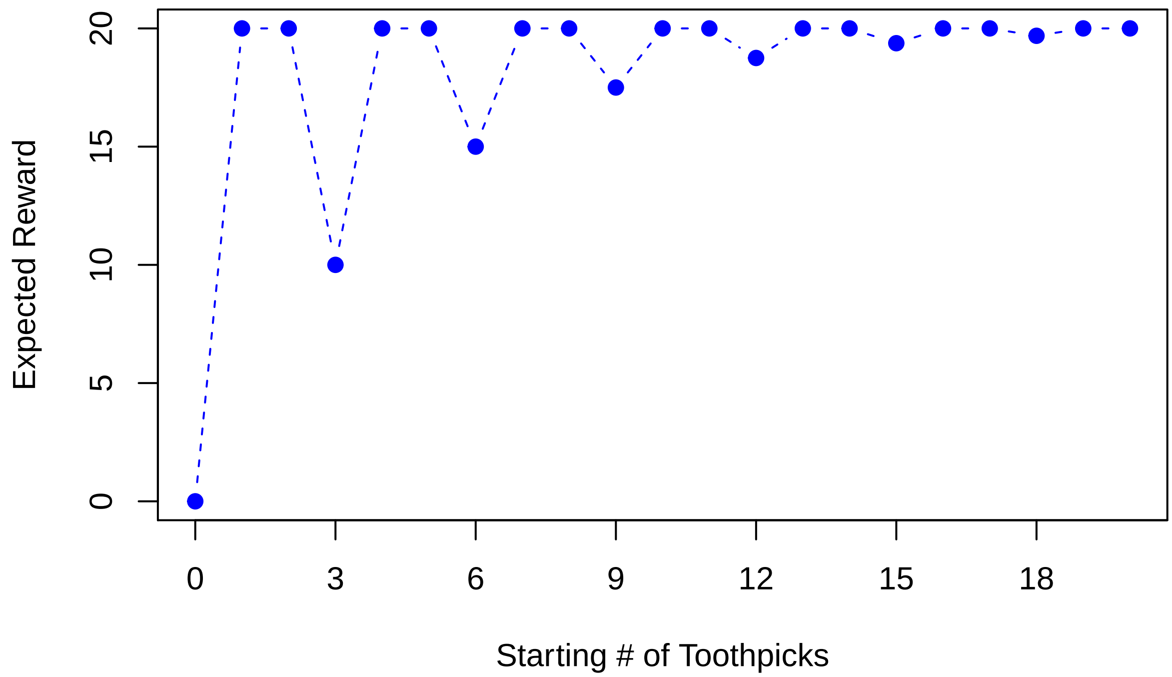
1.1 Implementing the Dynamic Programming Algorithm

```
# Notes:
# Introduce empty v[] and pick[]:
# Since R index starts from 1, we will let v[1] correspond to v(-1) on the notes.
# Similarly, v[2] corresponds to v(0), v[3] to v(1),..., v[n+2] to v(n).
# The same also applies for pick[n], i.e., pick[5] is the decision taken
# when there are 3 toothpicks remaining.

N=20;                    # Number of toothpicks
v=rep(0,N+2);            # Generate empty v[], dimension N+2
pick=rep(0,N+2);         # Generate empty pick[], dimension N+2
v[1]=0;                  # Initialize boundary values
v[2]=0;
v[3]=20;                 # Indicating first few values that are easily found to
v[4]=20;                 # simplify the form of the DP recursions below
pick[3]=1;               # Indicate corresponding actions (we write a few of the
pick[4]=2;               # initial decisions here so that DP recursions below
                         # all have the same form.

# DP recursions (from 3:20 toothpicks)
for(i in 5:(N+2)){
    v[i]=max(0.5*v[i-2]+0.5*v[i-3],0.5*v[i-3]+0.5*v[i-4])
    if(0.5*v[i-2]+0.5*v[i-3]>0.5*v[i-3]+0.5*v[i-4]){
        pick[i]=1;
    }
```

```
    if(0.5*v[i-2]+0.5*v[i-3]<=0.5*v[i-3]+0.5*v[i-4]){
        pick[i]=2;
    }
}
```

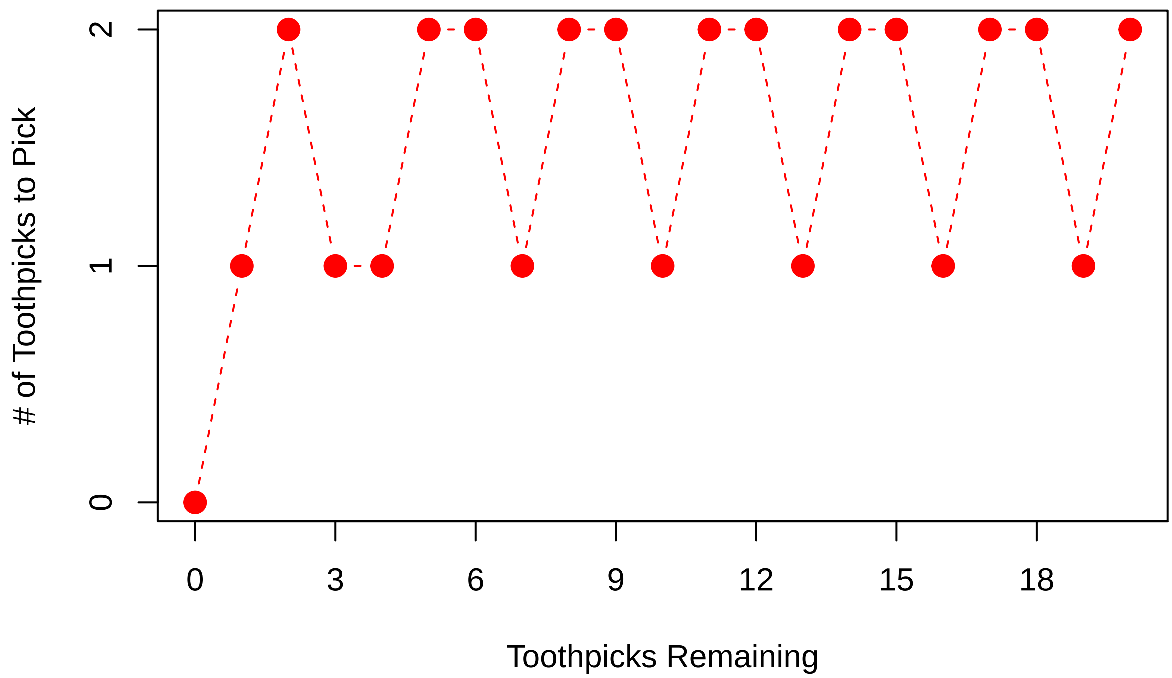
1.2 Visualizing Optimal Expected Reward

```
# Plotting the Value Function
xtick<-0:N
plot(xtick,v[2:(N+2)],type="b",pch = 19,
     col = "blue",lty = 2,xaxt="n",
     xlab="Starting # of Toothpicks",ylab="Expected Reward")
axis(side = 1, at = 3*xtick)
```



1.3 Visualizing the Optimal Policy

```
# Visualizing the Optimal Policy
xtick<-0:N
plot(xtick,pick[2:(N+2)], type="b",pch = 19,cex=1.5,
     col = "red",lty = 2,xaxt="n",yaxt="n",
     xlab="Toothpicks Remaining", ylab="# of Toothpicks to Pick")
axis(side = 1, at = 3*xtick)
axis(side = 2, at = 0:2)
```



2 Optimal Admission Decisions for Multiple Fare Classes with Sequential Arrivals

2.1 Setting up the Dynamic Programming Algorithm in R

```r
J<-5;                          # number of fare classes
price<-c(100,60,40,35,15);  # prices for each fare class, p1 highest as in notes
expd<-c(15,40,50,55,120);   # expected demand for each fare class (Poisson)
N<-200;                        # capacity
# v[j,x] is the optimal total expected revenue from
# fare classes j, j - 1 . . . , 1 given x units of
# remaining capacity just before facing the demand for fare class j.
# j=1 is the end of horizon, j=2 is the last stage (p1 arrivals), etc..
# n=1 means zero seats, n=2 means 1 seat, etc, i.e., x=n-1
v<-matrix(0, nrow = (J+1), ncol = (N+1));    # i.e., j=0:5 and x=0:200
ybest<-matrix(0, nrow = J+1, ncol = (N+1));

# # If need to set Terminal Values other than zero
# for(n in 1:(N+1)){
#     v[1,n]<-10*(n-1); # if we can salvage excess capacity say at 10 per unit.
# }

# Dynamic Programming Recursion
for(i in 2:(J+1)){ # i=2 is stage 1 (i.e., p1 arrivals), i=3 is stage 2, etc.
    for(n in 1:(N+1)){
        x=n-1; # inventory level
        valuebest=-999;
        for(y in 0:x){ # protect for future stages
            avail=x-y; # available for this stage
            value=0; # to start computing the expected revenue
            for(d in 0:175){ # can also set range for each class
                sold=min(avail,d);
                value=value+
                    dpois(d, expd[i-1])*(price[i-1]*sold+v[i-1,n-sold]);
            }
            if(value>valuebest){
                ybest[i,n]=y;
                valuebest=value;
            }
```

```
        }
        v[i,n]=valuebest;
    }
}
```

```
# Optimal Protection Limits
OptimalProtectionLimits<-c(ybest[3,100],ybest[4,100],ybest[5,200],ybest[6,200])
print(OptimalProtectionLimits)
```

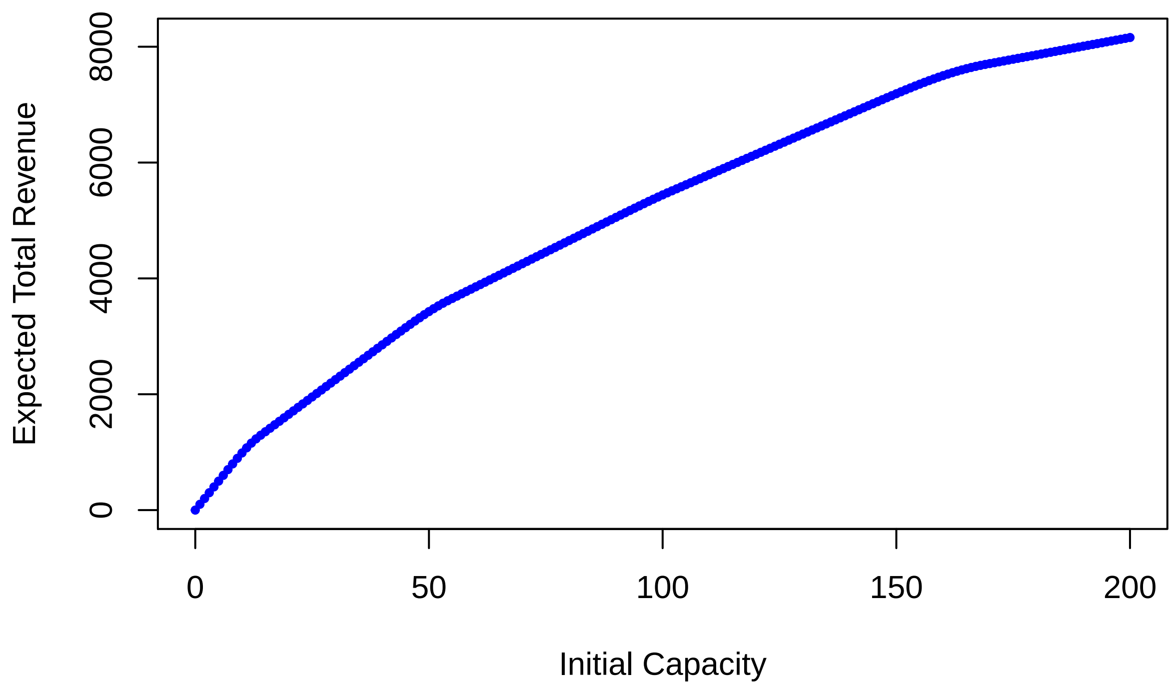
```
## [1]  14  54 101 169
```

```
# Optimal Total Expected Revenue
OptimalTotalExpectedRevenue=v[J+1,N+1]
print(OptimalTotalExpectedRevenue)
```

```
## [1] 8159.128
```

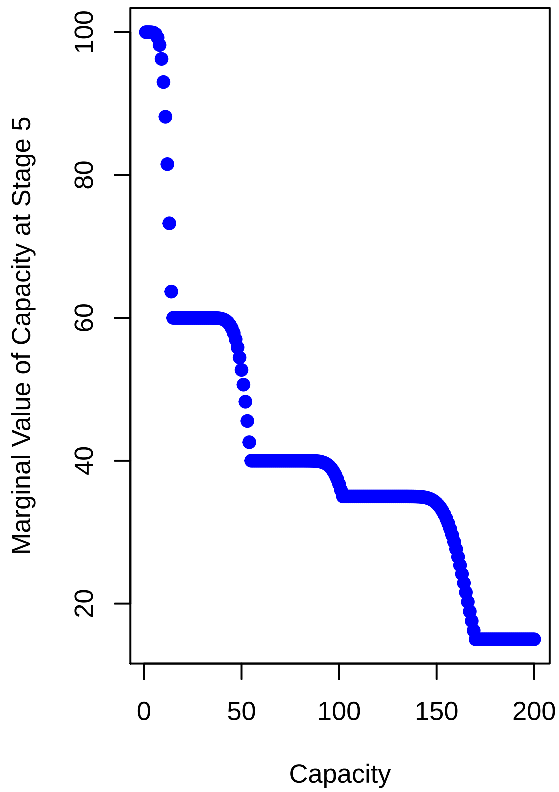
2.2 Visualizing Total Expected Revenue vs Initial Capacity

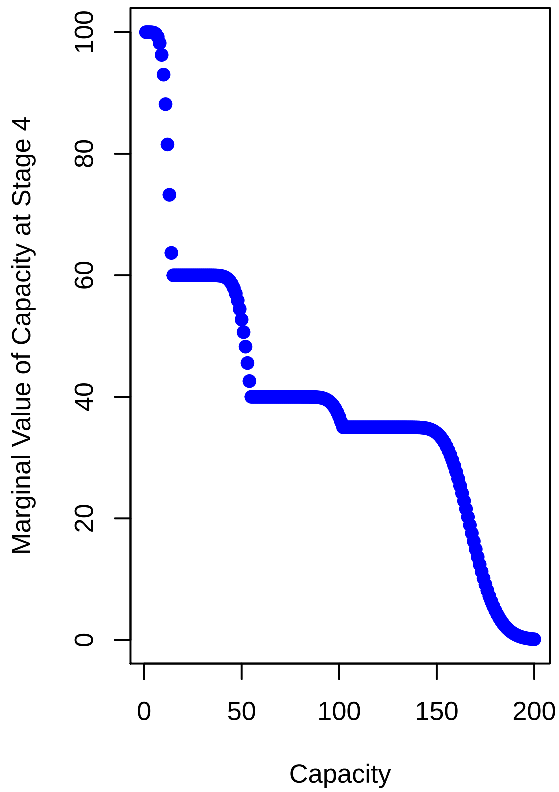
```
# Visualizing Total Expected Revenue vs Initial Capacity
xtick<-0:N
plot(xtick,v[(J+1),1:(N+1)],pch = 19, cex=0.5, col = "blue", type = "p",
     xlab="Initial Capacity", ylab="Expected Total Revenue")
```

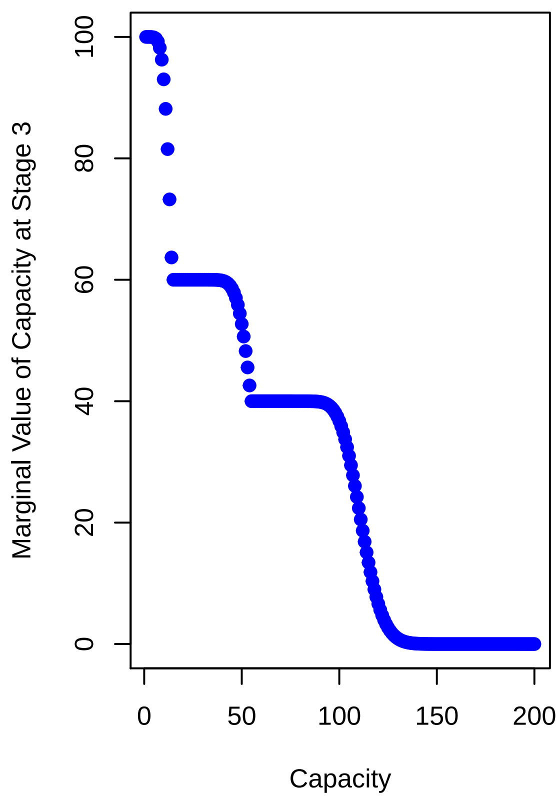


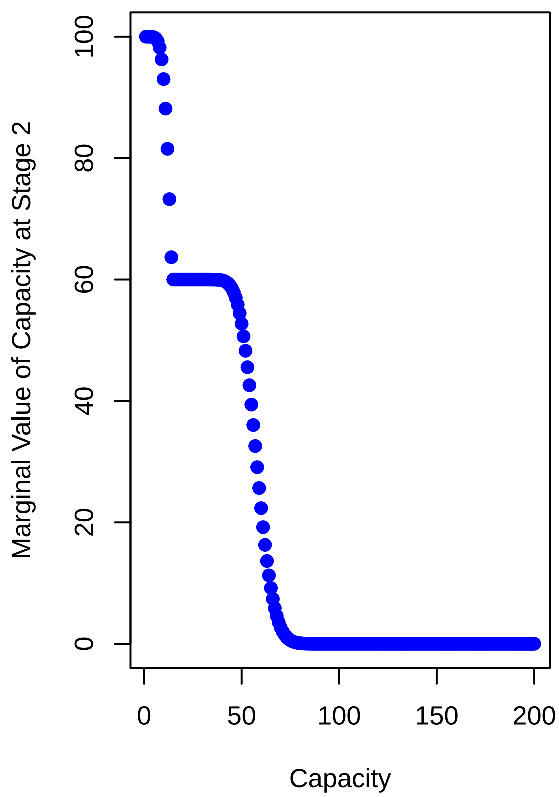
2.3 Visualizing Marginal Value of Capacity

```
# Value of an additional ubnit of capacity
# e.g., value of 100th unit of capacity is v[6,101]-v[6,100]
par(mfrow=c(2,2))
plot(xtick[2:(N+1)],v[6,2:(N+1)]-v[6,1:(N)],
     pch = 19, col = "blue", type = "p",
     xlab="Capacity", ylab="Marginal Value of Capacity at Stage 5")
plot(xtick[2:(N+1)],v[5,2:(N+1)]-v[5,1:(N)],
     pch = 19, col = "blue", type = "p",
     xlab="Capacity", ylab="Marginal Value of Capacity at Stage 4")
plot(xtick[2:(N+1)],v[4,2:(N+1)]-v[4,1:(N)],
     pch = 19, col = "blue", type = "p",
     xlab="Capacity", ylab="Marginal Value of Capacity at Stage 3")
plot(xtick[2:(N+1)],v[3,2:(N+1)]-v[3,1:(N)],
     pch = 19, col = "blue", type = "p",
     xlab="Capacity", ylab="Marginal Value of Capacity at Stage 2")
```


Marginal Value of Capacity at Stage 5
Capacity
20
40
60
80
100
0
50
100
150
200


Marginal Value of Capacity at Stage 4
Capacity
0
20
40
60
80
100
0
50
100
150
200


Marginal Value of Capacity at Stage 3
Capacity
0
20
40
60
80
100
0
50
100
150
200


Marginal Value of Capacity at Stage 2
Capacity
0
20
40
60
80
100
0
50
100
150
200

3 Heuristics for Multi Fare Classes

3.1 Obtaining Heuristic Protection Levels

```
# Please see notes for detailed description
remDemand<-rep(0, J);
remEffPrice<-rep(0, J);
CritFrac<-rep(0, J);
yheur<-rep(0, J);
for(i in 2:(J)){
 remDemand[i]=sum(expd[1:(i-1)]);
 remEffPrice[i]=sum(expd[1:(i-1)]*price[1:(i-1)])/remDemand[i];
 CritFrac[i]=(remEffPrice[i]-price[i])/remEffPrice[i];
 yheur[i]=qpois(CritFrac[i], remDemand[i]);
}
```

```
# Heuristic Protection Limits
HeuristicProtectionLimits<-yheur[2:5]
print(HeuristicProtectionLimits)
```

```
## [1]  14  54 102 166
```

3.2 Testing the Performance of the Heuristic Policy

```
# Testing the Performance of the Heuristic Policy
vheur<-matrix(0, nrow = (J+1), ncol = (N+1));   # i.e., j=0:5 and x=0:200
# Dynamic Programming Recursion
for(i in 2:(J+1)){ # i=2 is stage 1 (i.e., p1 arrivals), i=3 is stage 2, etc.
    for(n in 1:(N+1)){
        x=n-1; # inventory level
        y=yheur[i-1]; # protect for future
        avail=max(0,x-y); # available for this stage
        value=0; # to start computing the expected revenue
        for(d in 0:175){ # can also set range for each class
            sold=min(avail,d);
            value=value+
                dpois(d, expd[i-1])*(price[i-1]*sold+vheur[i-1,n-sold]);
            }
        vheur[i,n]=value;
```

```
    }
}
```

```
# Heuristic Total Expected Revenue
HeuristicTotalExpectedRevenue=vheur[J+1,N+1]
print(HeuristicTotalExpectedRevenue)
```

```
## [1] 8151.426
```

```
# Percent Difference between Optimal and Heuristic Profit
PercentDiffHeurOpt=(v[6,201]-vheur[6,201])/v[6,201]*100;
print(PercentDiffHeurOpt)
```

```
## [1] 0.09438732
```

4 Optimal Admission Decision for Two Fare Classes with Mixed Arrivals

4.1 Setting up the Dynamic Programming Algorithm

```
N=100; # seat availability
TT=200; # Length of time horizon
prob0=0.1;
prob1=0.3;
prob2=0.6;
price1=200;
price2=100;

v=matrix(rep( 0, len=(N+1)*(TT+1)), nrow=N+1);
accept2=matrix(rep( 0, len=(N+1)*(TT+1)), nrow=N+1); # decision for low fare

# Terminal Values
for(i in 1:(N+1)){
    v[i,1]=0;
}

# Dynamic Programming Recursion
for(t in 2:201){ #2:TT+1
    for(i in 1:(N+1)){ #1:N1+1
```

```r
        # For no arrivals:
        vtogo0=v[i,t-1];

        # For Product 1 arrival:
        vtogo1=v[i,t-1]; # default
        # If resource available:
        if(i>1){
            vtogo1=price1+v[i-1,t-1];
        }

        # For Product 2 arrival:
        vtogo2=v[i,t-1];
        accept2[i,t]=0;
        # If resource available:
        if(i>1){
            vtogo2=max(price2+v[i-1,t-1],v[i,t-1]);
            # Recording the decision in the accept2 variable:
            if(price2+v[i-1,t-1]>v[i,t-1]){
                accept2[i,t]=1;
            }
        }

        # Obtaining the overall value function from its parts:
        v[i,t]=prob0*vtogo0+prob1*vtogo1+prob2*vtogo2;
    }
}
```

```r
# Optimal Total Expected Revenue
OptimalTotalExpectedRevenue=v[101,201]
print(OptimalTotalExpectedRevenue)
```

```
## [1] 15980.4
```

4.2 Visualizing the Optimal Policy Sytructure

```r
# Visualizing the Optimal Policy Sytructure
acceptance<-t(accept2[2:101,2:201]); # transpose of accept2 (horizontal:time)
xaxis<-1:TT
yaxis<-1:N
```

```r
filled.contour(xaxis,yaxis,acceptance,xaxt="n",yaxt="n",
               key.axes = axis(4, seq(0, 1, by = 1)), nlevels = 2,
               color.palette = rainbow,
               xlab="Remaining Time", ylab="Remaining Number of Seats")
```