BTYD

Introduction

The BTYD package contains models to capture non-contractual purchasing behavior of customers—or, more simply, models that tell the story of people buying until they die (become inactive as customers).

The pareto/NBD model is used for non-contractual settings. Using four parameters, it describes the rate at which customers make purchases and the rate at which they drop out— allowing for heterogeneity in both regards.

Process

1. Reading the data

Data can be extracted using the simple query as provided in the repo ("BTYD.sql")

This query uses the table 'Delivered_GMV' from the data mart 'odsanalytics' db on 192.168.1.13.

This query can be directly accessed by the R code by using an RMySQL connection.

The SMEIDs need to be encoded as integers as BTYD code does not accept strings as the customer id.

This is left as an exercise for the next developer.

2. DATA PREPROCESSING

The data required to estimate pareto/NBD model parameters is surprisingly little. The customer-by-customer approach of the model is retained, but we need only three pieces of information for every person: how many transactions they made in the calibration period (frequency), the time of their last transaction (recency), and the total time for which they were observed. A customer-by sufficient-statistic matrix, as used by the BTYD package, is simply a matrix with a row for every customer and a column for each of the above-mentioned statistics.

The data will be available as an event log. This is a data structure which contains a row for every transaction, with a customer identifier, date of purchase, and (optionally) the amount of the transaction. dc.ReadLines is a function to convert an event log in a comma-delimited file to an data frame in R—you could use another function such as read.csv or read.table if desired, but dc.ReadLines simplifies things by only reading the data you require and giving the output appropriate column names. In the example below, we create an event log from the file "cdnowElog.csv", which has customer IDs in the first column, dates in the second column and sales numbers in the third column.

```
cdnowElog <- system.file("data/cdnowElog.csv", package = "BTYD")
elog <- dc.ReadLines(cdnowElog, cust.idx = 2,
                     date.idx = 3, sales.idx = 5)
elog[1:3,]

#    cust     date sales
# 1     1 19970101 29.33
# 2     1 19970118 29.73
# 3     1 19970802 14.96
```

Note the formatting of the dates in the output above. dc.Readlines saves dates as characters, exactly as they appeared in the original comma-delimited file. For many of the data-conversion functions in BTYD, however, dates need to be compared to each other—and unless your years/months/days happen to be in the right order, you probably want them to be sorted chronologically and not alphabetically. Therefore, we convert the dates in the event log to R Date objects:

```
elog$date <- as.Date(elog$date, "%Y%m%d");
elog[1:3,]

#    cust       date sales
# 1     1 1997-01-01 29.33
# 2     1 1997-01-18 29.73
# 3     1 1997-08-02 14.96
```

Our event log now has dates in the right format, but a bit more cleaning needs to be done. Transaction-flow models, such as the pareto/NBD, is concerned with interpurchase time. Since our timing information is only accurate to the day, we should merge all transactions that occurred on the same day. For this, we use dc.MergeTransactionsOnSameDate. This function returns an event log with only one transaction per customer per day, with the total sum of their spending for that day as the sales number.

```
elog <- dc.MergeTransactionsOnSameDate(elog);
```

To validate that the model works, we need to divide the data up into a calibration period and a holdout period. This is relatively simple with either an event log or a customer-by-time matrix, which we are going to create soon. We are going to use 30 September 1997 as the cutoff date, as this point (39 weeks) divides the dataset in half. The reason for doing this split now will become evident when we are building a customer-by-sufficient-statistic matrix from the customer-by-time matrix—it requires a last transaction date, and we want to make sure that last transaction date is the last date in the calibration period and not in the total period.

```
end.of.cal.period <- as.Date("1997-09-30")
elog.cal <- elog[which(elog$date <= end.of.cal.period), ]
```

The final cleanup step is a very important one. In the calibration period, the pareto/NBD model is generally concerned with repeat transactions—that is, the first transaction is ignored. This is convenient

for firms using the model in practice, since it is easier to keep track of all customers who have made at least one transaction (as opposed to trying to account for those who have not made any transactions at all). The one problem with simply getting rid of customers' first transactions is the following: We have to keep track of a "time zero" as a point of reference for recency and total time observed. For this reason, we use dc.SplitUpElogForRepeatTrans, which returns a filtered event log ($repeat.trans.elog) as well as saving important information about each customer ($cust.data)

```
split.data <- dc.SplitUpElogForRepeatTrans(elog.cal);
clean.elog <- split.data$repeat.trans.elog;
```

3. CREATING A CUSTOMER BY TIME MATRIX

The next step is to create a customer-by-time matrix. This is simply a matrix with a row for each customer and a column for each date. There are several different options for creating these matrices: - Frequency—each matrix entry will contain the number of transactions made by that customer on that day. Use dc.CreateFreqCBT. If you have already used dc.MergeTransactionsOnSameDate, this will simply be a reach customer-by-time matrix. - Reach—each matrix entry will contain a 1 if the customer made any transactions on that day, and 0 otherwise. Use dc.CreateReachCBT. - Spend—each matrix entry will contain the amount spent by that customer on that day. Use dc.CreateSpendCBT. You can set whether to use total spend for each day or average spend for each day by changing the is.avg.spend parameter. In most cases, leaving is.avg.spend as FALSE is appropriate.

```
freq.cbt <- dc.CreateFreqCBT(clean.elog);
freq.cbt[1:3,1:5]

#     date
# cust 1997-01-08 1997-01-09 1997-01-10 1997-01-11 1997-01-12
#    1          0          0          0          0          0
#    2          0          0          0          0          0
#    6          0          0          0          1          0
```

There are two things to note from the output above: 1. Customers 3, 4, and 5 appear to be missing, and in fact they are. They did not make any repeat transactions and were removed when we used dc.SplitUpElogForRepeatTrans.

We will get them back into the data when we use dc.MergeCustomers (soon). 2. The columns start on January 8—this is because we removed all the first transactions (and nobody made 2 transactions withing the first week).

Since we have deleted all the first transactions, the frequency customer-by-time matrix does not have any of the customers who made zero repeat transactions. These customers are still important; in

fact, in most datasets, more customers make zero repeat transactions than any other number. Solving the problem is reasonably simple: we create a customer-by-time matrix using all transactions, and then merge the filtered CBT with this total CBT (using data from the filtered CBT and customer IDs from the total CBT)

```
tot.cbt <- dc.CreateFreqCBT(elog)
cal.cbt <- dc.MergeCustomers(tot.cbt, freq.cbt)
```

From the calibration period customer-by-time matrix (and a bit of additional information we saved earlier), we can finally create the customer-by sufficient-statistic matrix described earlier. The function we are going to use is dc.BuildCBSFromCBTAndDates, which requires a customer-by-time matrix, starting and ending dates for each customer, and the time of the end of the calibration period.

It also requires a time period to use—in this case, we are choosing to use weeks. If we chose days instead, for example, the recency and total time observed columns in the customer-by-sufficient-statistic matrix would have gone up to 273 instead of 39, but it would be the same data in the end. This function could also be used for the holdout period by setting cbt.is.during.cal.period to FALSE. There are slight differences when this function is used for the holdout period—it requires different input dates (simply the start and end of the holdout period) and does not return a recency (which has little value in the holdout period).

```
birth.periods <- split.data$cust.data$birth.per
last.dates <- split.data$cust.data$last.date
cal.cbs.dates <- data.frame(birth.periods, last.dates,
                            end.of.cal.period)
cal.cbs <- dc.BuildCBSFromCBTAndDates(cal.cbt, cal.cbs.dates,
                                      per="week")
```

4. PARAMETER ESTIMATION

Now that we have the data in the correct format, we can estimate model parameters. To estimate parameters, we use pNBD.EstimateParameters, which requires a calibration period customer-by-sufficient-statistic matrix and (optionally) starting parameters. (1,1,1,1) is used as default starting parameters if none are provided. The function which is maximized is pNBD.cbs.LL, which returns the log-likelihood of a given set of parameters for a customer-by-sufficient-statistic matrix.

```
params <- pnbd.EstimateParameters(cal.cbs);
params

# [1]  0.5534 10.5802  0.6061 11.6562

LL <- pnbd.cbs.LL(params, cal.cbs);
LL

# [1] -9595
```

Let's run it a couple more times, with its own output as a starting point, to see if it converges:

```
p.matrix <- c(params, LL);
for (i in 1:2){
params <- pnbd.EstimateParameters(cal.cbs, params);
LL <- pnbd.cbs.LL(params, cal.cbs);
p.matrix.row <- c(params, LL);
p.matrix <- rbind(p.matrix, p.matrix.row);
}
colnames(p.matrix) <- c("r", "alpha", "s", "beta", "LL");
rownames(p.matrix) <- 1:3;
p.matrix;

#        r alpha      s  beta    LL
# 1 0.5534 10.58 0.6061 11.66 -9595
# 2 0.5534 10.58 0.6060 11.66 -9595
# 3 0.5534 10.58 0.6059 11.66 -9595
```

Now that we have the parameters, the BTYD package provides functions to interpret them. As we know, r and alpha describe the gamma mixing distribution of the NBD transaction process. We can see this gamma distribution in the figure, plotted using pNBD.PlotTransactionRateHeterogeneity(params). We also know that s and beta describe the gamma mixing distribution of the pareto (or gamma exponential) dropout process.

5. INDIVIDUAL LEVEL ESTIMATIONS

Now that we have parameters for the population, we can make estimations for customers on the individual level.

First, we can estimate the number of transactions we expect a newly acquired customer to make in a given time period. Let's say, for example, that we are interested in the number of repeat transactions a newly acquired customer will make in a time period of one year. Note that we use 52 weeks to represent one year, not 12 months, 365 days, or 1 year. This is because our parameters were estimated using weekly data.

```
pnbd.Expectation(params, t=52);

# [1] 1.473
```

We can also obtain expected characteristics for a specific customer, conditional on their purchasing behavior during the calibration period. The first of these is pnbd.ConditionalExpectedTransactions, which gives the number of transactions we expect a customer to make in the holdout period. The second is pnbd.PAlive, which gives the probability that a customer is still alive at the end of the calibration period. As above, the time periods used depend on which time period was used to estimate the parameters.

```
cal.cbs["1516",]

#     x   t.x T.cal
# 26.00 30.86 31.00

x <- cal.cbs["1516", "x"]
t.x <- cal.cbs["1516", "t.x"]
T.cal <- cal.cbs["1516", "T.cal"]
pnbd.ConditionalExpectedTransactions(params, T.star = 52,
                                     x, t.x, T.cal)

# [1] 25.46

pnbd.PAlive(params, x, t.x, T.cal)

# [1] 0.9979
```

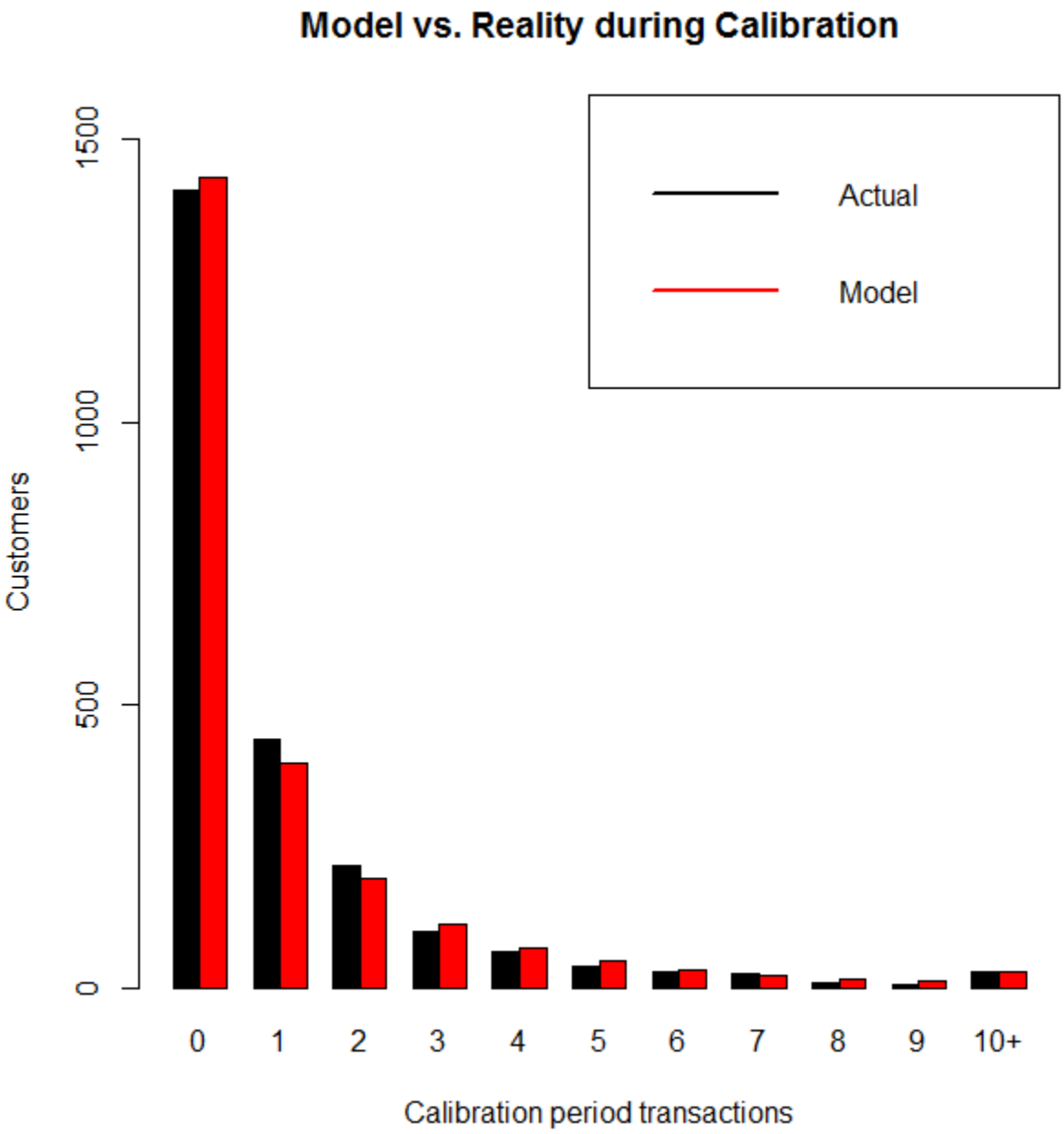
There is one more point to note here—using the conditional expectation function, we can see the "increasing frequency paradox" in action:

```
for (i in seq(10, 25, 5)){
  cond.expectation <- pnbd.ConditionalExpectedTransactions(
                        params, T.star = 52, x = i,
                        t.x = 20, T.cal = 39)

  cat ("x:",i,"\t Expectation:",cond.expectation, fill = TRUE)
}

# x: 10   Expectation: 0.7062
# x: 15   Expectation: 0.1442
# x: 20   Expectation: 0.02251
# x: 25   Expectation: 0.003093
```

6. PLOTTING THE GOODNESS OF FIT ON THE CALIBRATION PERIOD

We would like to be able to do more than make inferences about individual customers. The BTYD package provides functions to plot expected customer behavior against actual customer behaviour in the both the calibration and holdout periods. The first such function is the obvious starting point: a comparison of actual and expected frequencies within the calibration period.

pnbd.PlotFrequencyInCalibration(params, cal.cbs, 7)



This function obviously needs to be able to generate expected data (from estimated parameters) and requires the actual data (the calibration period customer-by-sufficient-statistic). It also requires another number, called the censor number. The histogram that is plotted is right-censored; after a certain number, all frequencies are binned together.

The number provided as a censor number determines where the data is cut off. Unfortunately, the only thing we can tell from comparing calibration period frequencies is that the fit between our model and the data isn't awful. We need to verify that the fit of the model holds into the holdout period.

Firstly, however, we are are going to need to get information for holdout period. dc.ElogToCbsCbt produces both a calibration period customer-by-sufficient-statistic matrix and a holdout period customer-by-sufficient-statistic matrix, which could be combined in order to find the number of transactions each customer made in the holdout period. However, since we did not use dc.ElogToCbsCbt, we are going to get the information directly from the event log.

Note that we subtract the number of repeat transactions in the calibration period from the total number of transactions. We remove the initial transactions first as we are not concerned with them.

```
elog <- dc.SplitUpElogForRepeatTrans(elog)$repeat.trans.elog;
x.star <- rep(0, nrow(cal.cbs));
cal.cbs <- cbind(cal.cbs, x.star);
elog.custs <- elog$cust;
for (i in 1:nrow(cal.cbs)){
  current.cust <- rownames(cal.cbs)[i]

  tot.cust.trans <- length(which(elog.custs == current.cust))
  cal.trans <- cal.cbs[i, "x"]
  cal.cbs[i, "x.star"] <-  tot.cust.trans - cal.trans
}
cal.cbs[1:3,]

#   x    t.x T.cal x.star
# 1 2 30.429 38.86      1
# 2 1  1.714 38.86      0
# 3 0  0.000 38.86      0
```

7. PREDICTING ON THE HOLDOUT PERIOD

Now we can see how well our model does in the holdout period. Figure 4 shows the output produced by the code below. It divides the customers up into bins according to calibration period frequencies and plots actual and conditional expected holdout period frequencies for these bins.

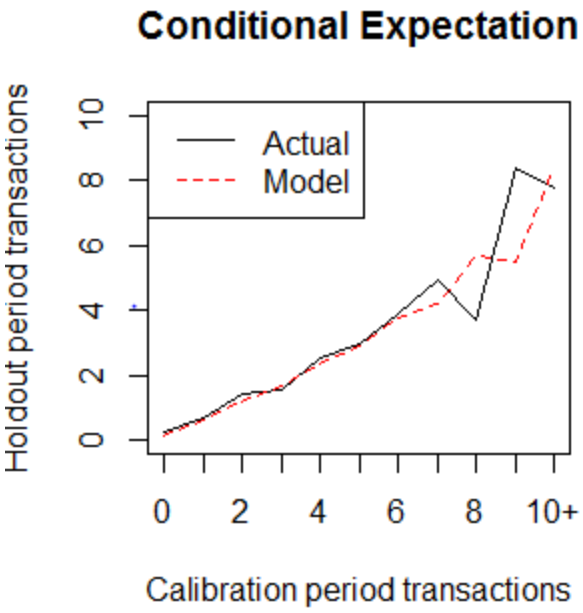
```
T.star <- 39 # length of the holdout period
censor <- 7  # This censor serves the same purpose described above
x.star <- cal.cbs[,"x.star"]
comp <- pnbd.PlotFreqVsConditionalExpectedFrequency(params, T.star,
                                                    cal.cbs, x.star, censor)
```

```
rownames(comp) <- c("act", "exp", "bin")
comp

#        freq.0   freq.1  freq.2  freq.3 freq.4 freq.5 freq.6 freq.7+
# act    0.2367   0.6970   1.393   1.560  2.532  2.947  3.862   6.359
# exp    0.1385   0.5996   1.196   1.714  2.399  2.907  3.819   6.403
# bin 1411.0000 439.0000 214.000 100.000 62.000 38.000 29.000  64.000
```

As you can see above, the graph also produces a matrix output. Most plotting functions in the BTYD package produce output like this. They are often worth looking at because they contain additional information not presented in the graph—the size of each bin in the graph. In this graph, for example, this information is important because the bin sizes show that the gap at zero means a lot more than the precision at 6 or 7 transactions.

Despite this, this graph shows that the model fits the data very well in the holdout period. Aggregation by calibration period frequency is just one way to do it. BTYD also provides plotting functions which aggregate by several other measures.

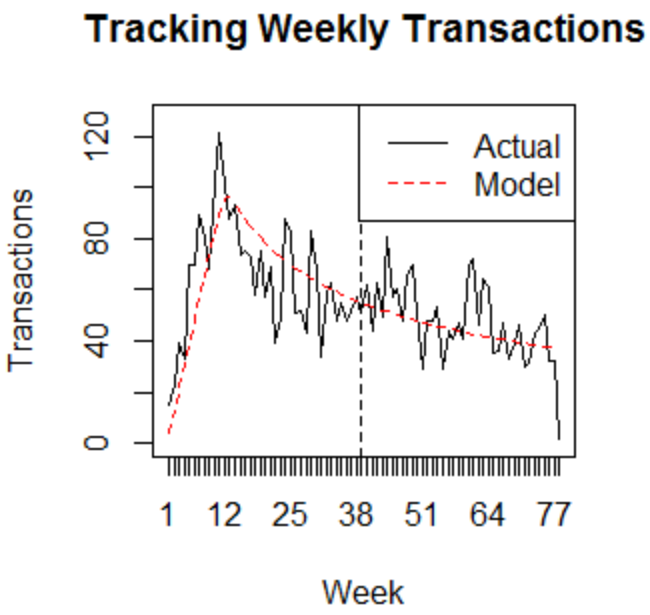
The other one that will be demonstrated here is aggregation by time—how well does our model predict how many transactions will occur in each week? The first step, once again, is going to be to collect the data we need to compare the model to. The customer-by-time matrix has already collected the data for us by time period; so we'll use that to gather the total transactions per day. Then we convert the daily tracking data to weekly data.



Actual vs. expected transactions

```
tot.cbt <- dc.CreateFreqCBT(elog)

# ...Completed Freq CBT

d.track.data <- rep(0, 7 * 78)
origin <- as.Date("1997-01-01")
for (i in colnames(tot.cbt)){
  date.index <- difftime(as.Date(i), origin) + 1;
  d.track.data[date.index] <- sum(tot.cbt[,i]);
}
w.track.data <-  rep(0, 78)
for (j in 1:78){
  w.track.data[j] <- sum(d.track.data[(j*7-6):(j*7)])
}
```

Now, we can make a plot comparing the actual number of transactions to the expected number of transactions on a weekly basis, as shown in figure 5. Note that we set n.periods.final to 78. This is to show that we are workly with weekly data. If our tracking data was daily, we would use 546 here—the function would plot our daily tracking data against expected daily transactions, instead of plotting our weekly tracking data against expected weekly transactions. This concept may be a bit tricky, but is explained in the documentation for pnbd.PlotTrackingInc. The reason there are two numbers for the total period (T.tot and n.periods.final) is that your customer-by sufficient-statistic matrix and your tracking data may be in different time periods.



Actual vs expected behavior by week

```
T.cal <- cal.cbs[,"T.cal"]
T.tot <- 78
n.periods.final <- 78
inc.tracking <- pnbd.PlotTrackingInc(params, T.cal,
                                     T.tot, w.track.data,
                                     n.periods.final)
```

```
inc.tracking[,20:25]

#           [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
# actual   73.00 55.00 70.00 33.00 56.00 99.00
# expected 78.31 76.42 74.65 72.98 71.41 69.93
```

Although the figure shows that the model is definitely capturing the trend of customer purchases over time, it is very messy and may not convince skeptics. Furthermore, the matrix, of which a sample is shown, does not really convey much information since purchases can vary so much from one week to the next. For these reasons, we may need to smooth the data out by cumulating it over time.