Buy 'Til You Die - A Walkthrough

Daniel McCarthy, Edward Wadsworth

November, 2014

1 Version 2.4.3 Overview

This version patches the Pareto/NBD component of BTYD 2.4 using the fix that Theo Strinopoulos proposed here.

Everything below this section reads identically to the vignette of BTYD 2.4. The main difference is that all evaluated references to BTYD file paths, libraries, etc. now point to their patched counterparts.

As of version 2.4.1 the patched BTYD functions have been modified as follows:

- Some functions defined in `R/pnbd.R` and in `R/bgnbd.R` (you can tell them by their names, which start with `pnbd` and `bgnbd` respectively) have been changed.
- Some of these functions now take an extra logical argument, `hardie`; if TRUE, this function (or a function it calls) uses Bruce Hardie's algorithm for estimating the real part of the Gaussian hypergeometric function (see MATLAB code on page 4 of Fader et al. (2005)); if FALSE, it makes use of the hypergeo R package from CRAN. For the purposes of this vignette, this parameter is set globally as `allHardie = TRUE`. This mirrors the choice of the BTYD package authors, who use Bruce Hardie's algorithm everywhere.
- Instead of `base::optim`, the `pnbd` functions use `optimx::optimx` and they now allow you to pick your optimization method, using the `method` argument, which defaults to `L-BFGS-B` without any constraints, as in the original package.
- Some of the functions defined in `R/bgnbd.R` have been changed in order to implement the fix for the `NUM!` error problem proposed in this note.

As of version 2.4.2 the `hardie` argument defaults to TRUE with the exception of `bgnbd.generalParams` where it defaults to NULL.

Version 2.4.3 adds a fix to the `B1B2` component of the function `pnbd.pmf.General` defined in `R/pnbd.R` for a bug discovered by Patrik Schilter. The fix now specifies correctly the third parameter of the B2 function described by equation (24) here as $r + s + x + 1$. The previous, incorrect version added the i component from the first term.

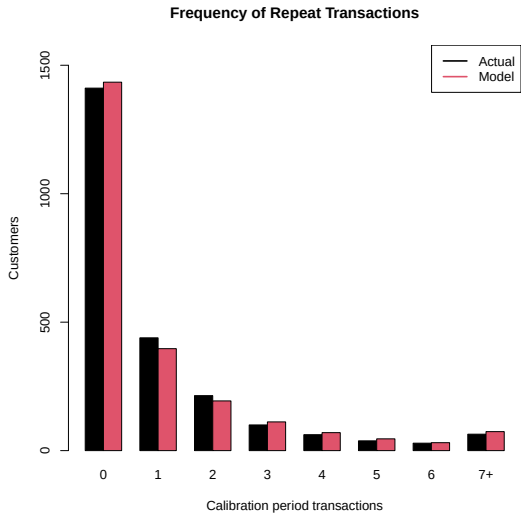

Figure 1: Calibration period fit of Pareto/NBD model to CDNOW dataset.

2 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 main models presented in the package are the Pareto/NBD, BG/NBD and BG/BB models, which describe scenario of the firm not being able to observe the exact time at which a customer drops out. We will cover each in turn. If you are unfamiliar with these models, Fader et al. (2004) provides a description of the BG/NBD model, Fader et al. (2005) provides a description of the Pareto/NBD model and Fader et al. (2010) provides a description of the BG/BB model.

3 Pareto/NBD

The Pareto/NBD model is used for non-contractual situations in which customers can make purchases at any time. 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, of course. We will walk through the Pareto/NBD functionality provided by the BTYD package using the CDNOW[1] dataset. As shown by figure 1, the Pareto/NBD model describes this dataset quite well.

[1]Provided with the BTYD package and available at brucehardie.com. For more details, see the documentation of the cdnowSummary data included in the package.

3.1 Data Preparation

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.

You may find yourself with the data 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 second column, dates in the third column and sales numbers in the fifth 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. I am 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
```

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`. Do not worry though—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).

We have a small problem now—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")
```

You'll be glad to hear that, for the process described above, the package contains a single function to do everything for you: `dc.ElogToCbsCbt`. However, it is good to be aware of the process above, as you might want to make small changes for different situations—for example, you may not want to remove customers' initial transactions, or your data may be available as a customer-by-time matrix and not as an event log. For most standard situations, however, `dc.ElogToCbsCbt` will do. Reading about its parameters and output in the package documentation will help you to understand the function well enough to use it for most purposes.

3.2 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 = cal.cbs,
                                  hardie = allHardie)
round(params, digits = 3)

# [1]  0.553 10.580  0.606 11.656

LL <- pnbd.cbs.LL(params = params,
                  cal.cbs = cal.cbs,
                  hardie = allHardie)
LL

# [1] -9594.976
```

As with any optimization, we should not be satisfied with the first output we get. 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 = cal.cbs,
                                    par.start = params,
                                    hardie = allHardie)
  LL <- pnbd.cbs.LL(params = params,
                    cal.cbs = cal.cbs,
                    hardie = allHardie)
  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
round(p.matrix, digits = 3)

#       r alpha     s   beta        LL
# 1 0.553 10.58 0.606 11.656 -9594.976
# 2 0.553 10.58 0.606 11.657 -9594.976
# 3 0.553 10.58 0.606 11.658 -9594.976
```

This estimation does converge. I am not going to demonstrate it here, but it is always a good idea to test the estimation function from several starting points.

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 figure 2, 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. We can see this gamma distribution in figure 3, plotted using `pnbd.PlotDropoutRateHeterogeneity(params)`.

From these, we can already tell something about our customers: they are more likely to have low values for their individual poisson transaction process parameters, and more likely to have low values for their individual exponential dropout process parameters.

3.3 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.

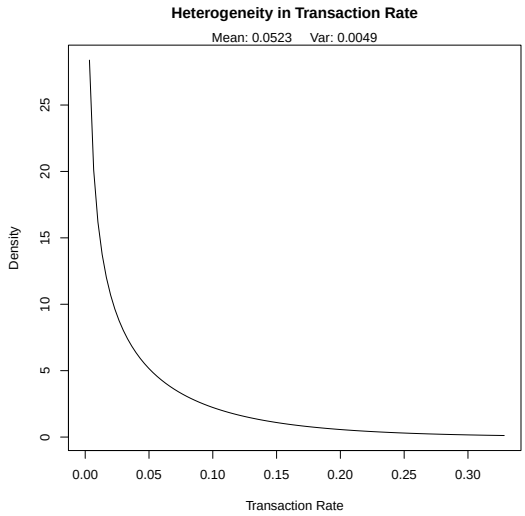

Figure 2: Transaction rate heterogeneity of estimated parameters.

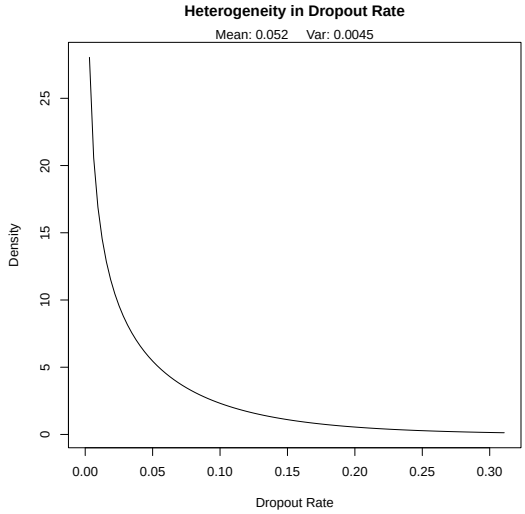

Figure 3: Dropout rate heterogeneity of estimated parameters.

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

# [1] 1.473434
```

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
# 26.00000 30.85714 31.00000

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,
                                     hardie = allHardie)

# [1] 25.45647

pnbd.PAlive(params,
            x,
            t.x,
            T.cal,
            hardie = allHardie)

# [1] 0.997874
```

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

```
# avoid overflow in LaTeX code block here:
cet <- "pnbd.ConditionalExpectedTransactions"
for (i in seq(10, 25, 5)){
  cond.expectation <- match.fun(cet)(params,
                                     T.star = 52,
                                     x = i,
                                     t.x = 20,
```

```
                                        T.cal = 39,
                                        hardie = allHardie)
  cat ("x:",i,"\t Expectation:",cond.expectation, fill = TRUE)
}

# x: 10   Expectation: 0.7062289
# x: 15   Expectation: 0.1442396
# x: 20   Expectation: 0.02250658
# x: 25   Expectation: 0.00309267
```

3.4 Plotting/ Goodness-of-fit

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 behavior 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. This is figure 1, which was generated using the following code:

```
pnbd.PlotFrequencyInCalibration(params = params,
                                cal.cbs = cal.cbs,
                                censor = 7,
                                hardie = allHardie)
```

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`, I am going to get the information directly from the event log. Note that I 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
}
round(cal.cbs[1:3,], digits = 3)

#   x    t.x  T.cal x.star
# 1 2 30.429 38.857      1
# 2 1  1.714 38.857      0
# 3 0  0.000 38.857      0
```

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"]
pdf(file = 'pnbdCondExpComp.pdf')
comp <- pnbd.PlotFreqVsConditionalExpectedFrequency(params,
                                                    T.star,
                                                    cal.cbs,
                                                    x.star,
                                                    censor,
                                                    hardie = allHardie)
dev.off()

# pdf
#   2

rownames(comp) <- c("act", "exp", "bin")
round(comp, digits = 3)

#       freq.0  freq.1  freq.2  freq.3 freq.4 freq.5 freq.6 freq.7+
# act    0.237   0.697   1.393   1.560  2.532  2.947  3.862   6.359
# exp    0.138   0.600   1.196   1.714  2.399  2.907  3.819   6.403
# bin 1411.000 439.000 214.000 100.000 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

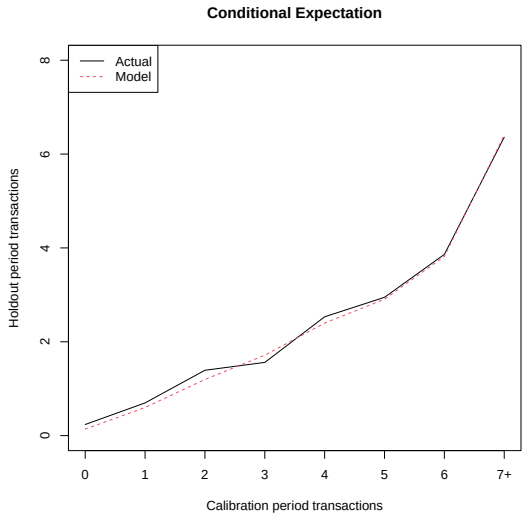

Figure 4: Actual vs. conditional expected transactions in the holdout period.

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 I will demonstrate 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.

```
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 working 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.

```
T.cal <- cal.cbs[,"T.cal"]
T.tot <- 78
n.periods.final <- 78

pdf(file = 'pnbdTrackingInc.pdf')
inc.tracking <- pnbd.PlotTrackingInc(params = params,
                                     T.cal = T.cal,
                                     T.tot = T.tot,
                                     actual.inc.tracking.data = w.track.data,
                                     n.periods.final = n.periods.final)
dev.off()

# pdf
#   2

round(inc.tracking[,20:25], digits = 3)

#             [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
# actual   73.000 55.000 70.000 33.000 56.000 99.000
# expected 78.308 76.419 74.648 72.983 71.414 69.933
```

Although figure 5 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 cummulating it over time, as shown in figure 6.

```
cum.tracking.data <- cumsum(w.track.data)
pdf(file = 'pnbdTrackingCum.pdf')
cum.tracking <- pnbd.PlotTrackingCum(params = params,
                                     T.cal = T.cal,
```

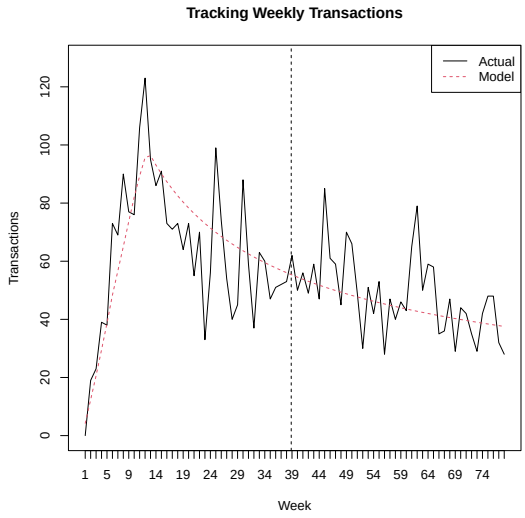

Figure 5: Actual vs. expected incremental purchasing behaviour.

```
                                    T.tot = T.tot,
                                    actual.cu.tracking.data = cum.tracking.data,
                                    n.periods.final = n.periods.final)
dev.off()

# pdf
#   2

round(cum.tracking[,20:25], digits = 3)

#              [,1]     [,2]     [,3]     [,4]    [,5]     [,6]
# actual   1359.000 1414.000 1484.000 1517.000 1573.00 1672.000
# expected 1308.856 1385.275 1459.923 1532.906 1604.32 1674.253
```

4 BG/NBD

The BG/NBD model, like the Pareto/NBD model, is used for non-contractual situations in which customers can make purchases at any time. It describes the rate at which customers make purchases and the rate at which they drop out with four parameters—allowing for heterogeneity in both. We will walk through the BG/NBD functions provided by the BTYD package using the CDNOW[2]

[2]Provided with the BTYD package and available at brucehardie.com. For more details, see the documentation of the cdnowSummary data included in the package.

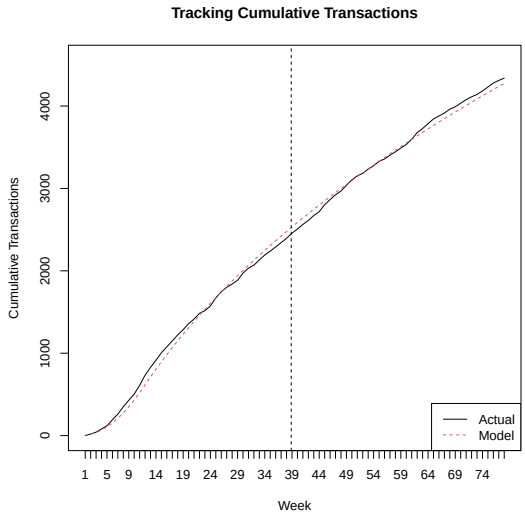

Figure 6: Actual vs. expected cumulative purchasing behaviour.

dataset. As shown by figure 7, the BG/NBD model describes this dataset quite well.

4.1 Data Preparation

The data required to estimate BG/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. This is the same as what is needed for the Pareto/NBD model. Indeed, if you have read the data preparation section for the Pareto/NBD model, you can safely skip over this section and move to the section on Parameter Estimation.

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.

You may find yourself with the data 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 second

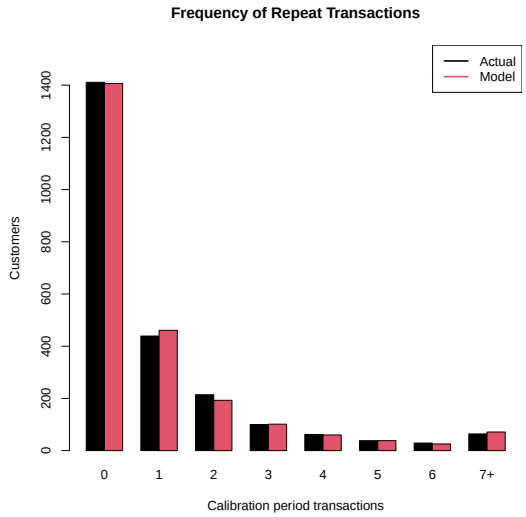

Figure 7: Calibration period fit of BG/NBD model to CDNOW dataset.

column, dates in the third column and sales numbers in the fifth 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 BG/NBD, is concerned with inter-purchase 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. I am 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 BG/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;
```

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`. Do not worry though—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).

We have a small problem now—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")
```

You'll be glad to hear that, for the process described above, the package contains a single function to do everything for you: `dc.ElogToCbsCbt`. However, it is good to be aware of the process above, as you might want to make small changes for different situations—for example, you may not want to remove customers' initial transactions, or your data may be available as a customer-by-time matrix and not as an event log. For most standard situations, however, `dc.ElogToCbsCbt` will do. Reading about its parameters and output in the package documentation will help you to understand the function well enough to use it for most purposes.

4.2 Parameter Estimation

Now that we have the data in the correct format, we can estimate model parameters. To estimate parameters, we use `bgnbd.EstimateParameters`, which requires a calibration period customer-by-sufficient-statistic matrix and (optionally) starting parameters. (1,3,1,3) is used as default starting parameters if none are provided as these values tend to provide good convergence across data sets. The function which is maximized is `bgnbd.cbs.LL`, which returns the log-likelihood of a given set of parameters for a customer-by-sufficient-statistic matrix. Below, we start with relatively uninformative starting values:

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

#        p1        p2        p3        p4
# 0.2425982 4.4136842 0.7929899 2.4261667
```

```
LL <- bgnbd.cbs.LL(params, cal.cbs);
LL

# [1] -9582.429
```

As with any optimization, we should not be satisfied with the first output we get. 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 <- bgnbd.EstimateParameters(cal.cbs, params);
LL <- bgnbd.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", "a", "b", "LL");
rownames(p.matrix) <- 1:3;
p.matrix;

#           r    alpha         a        b        LL
# 1 0.2425982 4.413684 0.7929899 2.426167 -9582.429
# 2 0.2425965 4.413685 0.7929888 2.426166 -9582.429
# 3 0.2425967 4.413659 0.7929869 2.426164 -9582.429
```

This estimation does converge. I am not going to demonstrate it here, but it is always a good idea to test the estimation function from several starting points.

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 figure 8, plotted using `bgnbd.PlotTransactionRateHeterogeneity(params)`. We also know that a and b describe the beta mixing distribution of the beta geometric dropout probabilities. We can see this beta distribution in figure 9, plotted using `bgnbd.PlotDropoutHeterogeneity(params)`.

From these, we can already tell something about our customers: they are more likely to have low values for their individual poisson transaction process parameters, and more likely to have low values for their individual geometric dropout probability parameters.

4.3 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

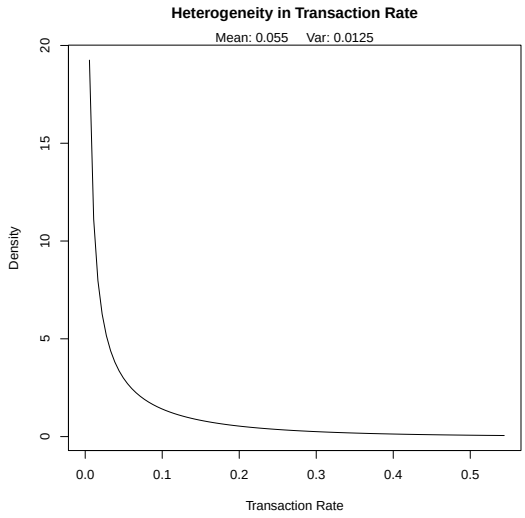

Figure 8: Transaction rate heterogeneity of estimated parameters.

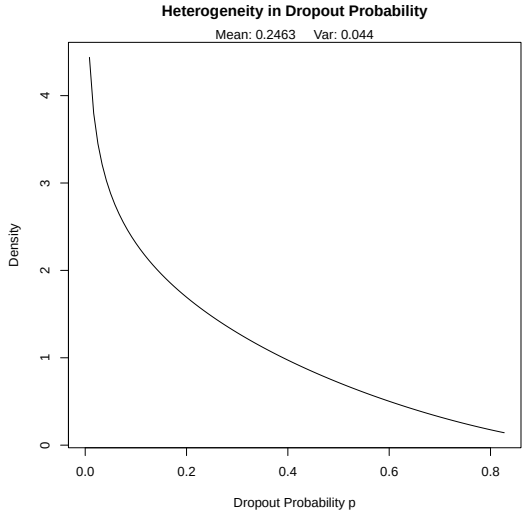

Figure 9: Dropout probability heterogeneity of estimated parameters.

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.

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

#       p3
# 1.444004
```

We can also obtain expected characteristics for a specific customer, conditional on their purchasing behavior during the calibration period. The first of these is `bgnbd.ConditionalExpectedTransactions`, which gives the number of transactions we expect a customer to make in the holdout period. The second is `bgnbd.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
# 26.00000 30.85714 31.00000

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

#       p3
# 25.75659

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

#        p3
# 0.9688523
```

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 <- bgnbd.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: 0.3474606
```

```
# x: 15   Expectation: 0.04283391
# x: 20   Expectation: 0.004158973
# x: 25   Expectation: 0.0003583685
```

4.4 Plotting/ Goodness-of-fit

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. This is figure 7, which was generated using the following code:

```
bgnbd.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`, I am going to get the information directly from the event log. Note that I 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 30.428571 38.85714      1
# 2 1  1.714286 38.85714      0
# 3 0  0.000000 38.85714      0
```

Now we can see how well our model does in the holdout period. Figure 10 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"]

pdf(file = 'bgnbdCondExpComp.pdf')
comp <- bgnbd.PlotFreqVsConditionalExpectedFrequency(params, T.star,
                                                     cal.cbs, x.star, censor)
dev.off()

# pdf
#   2

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

#           freq.0      freq.1     freq.2     freq.3    freq.4    freq.5
# act    0.2367116   0.6970387   1.392523   1.560000  2.532258  2.947368
# exp    0.2250893   0.5231364   1.044126   1.520256  2.163824  2.653789
# bin 1411.0000000 439.0000000 214.000000 100.000000 62.000000 38.000000
#        freq.6   freq.7+
# act  3.862069  6.359375
# exp  3.503957  6.157036
# bin 29.000000 64.000000
```

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 I will demonstrate 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.

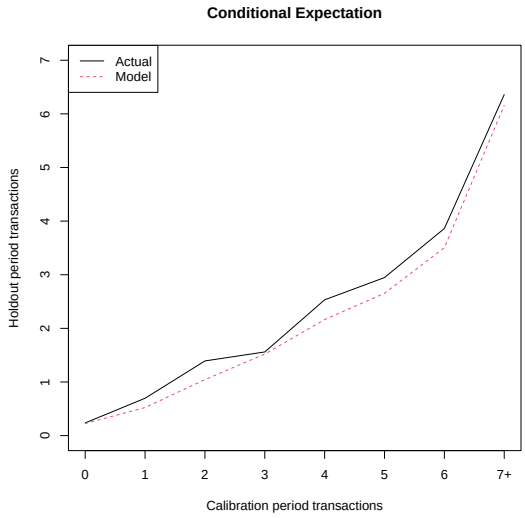

Figure 10: Actual vs. conditional expected transactions in the holdout period.

```
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 11. Note that we set `n.periods.final` to 78. This is to show that we are working 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 `bgnbd.PlotTrackingInc`. The reason there are two numbers for the total period (`T.tot` and `n.periods.final`) is that your customer-by-

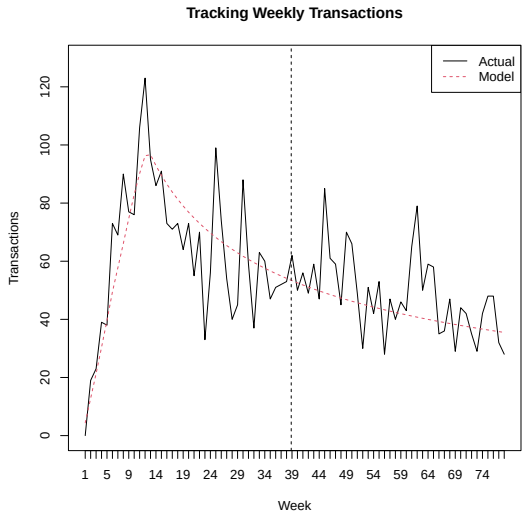

Figure 11: Actual vs. expected incremental purchasing behaviour.

sufficient-statistic matrix and your tracking data may be in different time periods.

```
T.cal <- cal.cbs[,"T.cal"]
T.tot <- 78
n.periods.final <- 78
pdf(file = 'bgnbdTrackingInc.pdf')
inc.tracking <- bgnbd.PlotTrackingInc(params,
                                      T.cal,
                                      T.tot,
                                      w.track.data,
                                      n.periods.final,
                                      allHardie)
dev.off()

# pdf
#   2

inc.tracking[,20:25]

#              [,1]     [,2]     [,3]     [,4]     [,5]     [,6]
# actual   73.00000 55.00000 70.00000 33.00000 56.00000 99.00000
# expected 76.86531 74.88843 73.04554 71.32166 69.70412 68.18213
```

Although figure 11 shows that the model is definitely capturing the trend of customer purchases over time, it is very messy and may not convince skeptics.

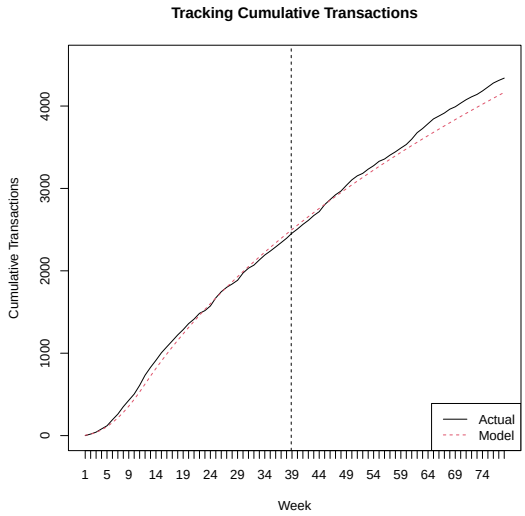

Figure 12: Actual vs. expected cumulative purchasing behaviour.

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, as shown in Figure 12.

```
cum.tracking.data <- cumsum(w.track.data)
pdf(file = 'bgnbdTrackingCum.pdf')
cum.tracking <- bgnbd.PlotTrackingCum(params,
                                      T.cal,
                                      T.tot,
                                      cum.tracking.data,
                                      n.periods.final,
                                      allHardie)
dev.off()

# pdf
#   2

cum.tracking[,20:25]

#              [,1]     [,2]     [,3]     [,4]     [,5]   [,6]
# actual   1359.000 1414.000 1484.000 1517.000 1573.000 1672.0
# expected 1312.458 1387.346 1460.392 1531.713 1601.418 1669.6
```

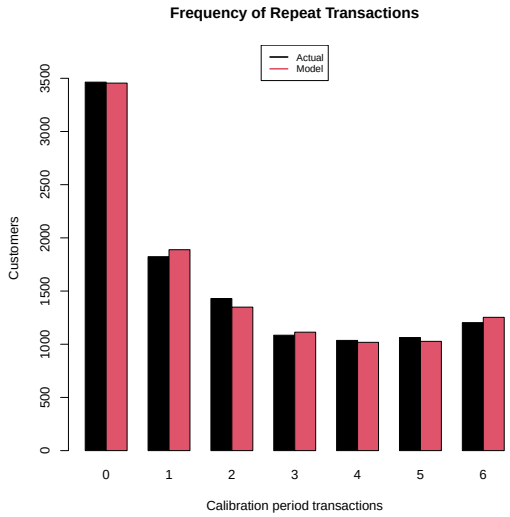

Figure 13: Calibration period fit of BG/BB model to the donations dataset.

5 BG/BB

The BG/BB model is also used for non-contractual settings. In many regards, it is very similar to the Pareto/NBD model—it also uses four parameters to describe a purchasing process and a dropout process. The difference between the models is that the BG/BB is used to describe situations in which customers have discrete transaction opportunities, rather than being able to make transactions at any time. For this section, we will be using donation data presented in Fader et. al. (2010). Figure 13 shows that this model also fits the right type of data well.

5.1 Data Preparation

Luckily, preparing data to be used by the BTYD package BG/BB functions is going to be very easy if you understood how to set up the data for the Pareto/NBD package. The BG/BB model uses exactly the same data as the Pareto/NBD model, but since it is discrete we can go one step further and create a recency-frequency matrix from our customer-by-sufficient-statistic matrix. We are able to do this because the data is discrete—a recency-frequency matrix consists of a row for every possible calibration period recency/frequency combination, and contains the total number of customers which had that particular combination of recency and frequency. While this is not strictly necessary, it greatly reduces the amount of space required to store the data and makes parameter estimation much faster—for the donation data, for example, there is a reduction from 11,104

rows (number of customers) down to 22.

Since I don't have access to the original event log for the donation data, I am going to demonstrate the data preparation process with simulated data (included in the package):

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

#   cust       date
# 1    1 1970-01-01
# 2    1 1975-01-01
# 3    1 1977-01-01

elog$date <- as.Date(elog$date, "%Y-%m-%d")

max(elog$date);

# [1] "1983-01-01"

min(elog$date);

# [1] "1970-01-01"

# let's make the calibration period end somewhere in-between
T.cal <- as.Date("1977-01-01")

simData <- dc.ElogToCbsCbt(elog, per="year", T.cal)
cal.cbs <- simData$cal$cbs

freq<- cal.cbs[,"x"]
rec <- cal.cbs[,"t.x"]
trans.opp <- 7 # transaction opportunities
cal.rf.matrix <- dc.MakeRFmatrixCal(freq, rec, trans.opp)
cal.rf.matrix[1:5,]

#      x t.x n.cal custs
# [1,] 0   0     7  2900
# [2,] 1   1     7   933
# [3,] 1   2     7   218
# [4,] 2   2     7   489
# [5,] 1   3     7    95
```

5.2 Parameter Estimation

Estimating BG/BB parameters is very similar to estimating Pareto/NBD parameters. The same rules apply—we use `bgbb.EstimateParameters`, which also uses (1,1,1,1) as default starting parameters. Once again, we should run the estimation several times with its own output as starting parameters, to make sure that our estimation converges. One should also test the estimation from different starting points, but I am not going to do that here.

```
data(donationsSummary);
rf.matrix <- donationsSummary$rf.matrix
params <- bgbb.EstimateParameters(rf.matrix);
LL <- bgbb.rf.matrix.LL(params, rf.matrix);
p.matrix <- c(params, LL);
for (i in 1:2){
params <- bgbb.EstimateParameters(rf.matrix, params);
LL <- bgbb.rf.matrix.LL(params, rf.matrix);
p.matrix.row <- c(params, LL);
p.matrix <- rbind(p.matrix, p.matrix.row);
}
colnames(p.matrix) <- c("alpha", "beta", "gamma", "delta", "LL");
rownames(p.matrix) <- 1:3;
p.matrix;

#      alpha      beta     gamma    delta        LL
# 1 1.203507 0.7497667 0.6567568 2.783887 -33225.58
# 2 1.203533 0.7497272 0.6567788 2.783812 -33225.58
# 3 1.203528 0.7497260 0.6567808 2.783796 -33225.58
```

The parameter estimation converges very quickly. It is much easier, and faster, to estimate BG/BB parameters than it is to estimate Pareto/NBD parameters, because there are fewer calculations involved.

We can interpret these parameters by plotting the mixing distributions. Alpha and beta describe the beta mixing distribution of the beta-Bernoulli transaction process. We can see the beta distribution with parameters alpha and beta in figure 14, plotted using `bgbb.PlotTransactionRateHeterogeneity(params)`. Gamma and Delta describe the beta mixing distribution of the beta-geometric dropout process. We can see the beta distribution with parameters gamma and delta in figure 15, plotted using `bgbb.PlotDropoutHeterogeneity(params)`. The story told by these plots describes the type of customers most firms would want—their transaction parameters are more likely to be high, and their dropout parameters are more likely to be low.

5.3 Individual Level Estimations

We can estimate the number of transactions we expect a newly acquired customer to make in a given time period, just as with the Pareto/NBD model. For

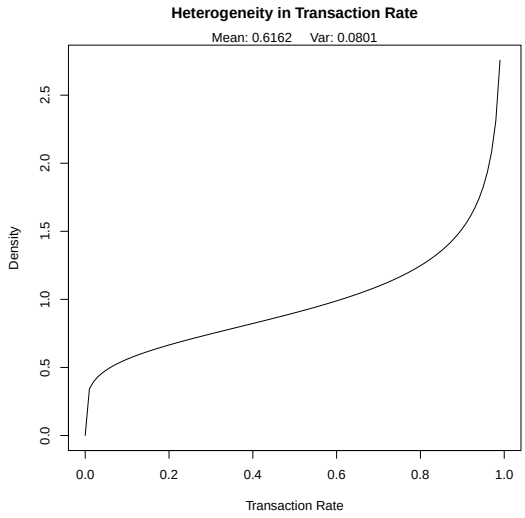

Figure 14: Transaction rate heterogeneity of estimated parameters.

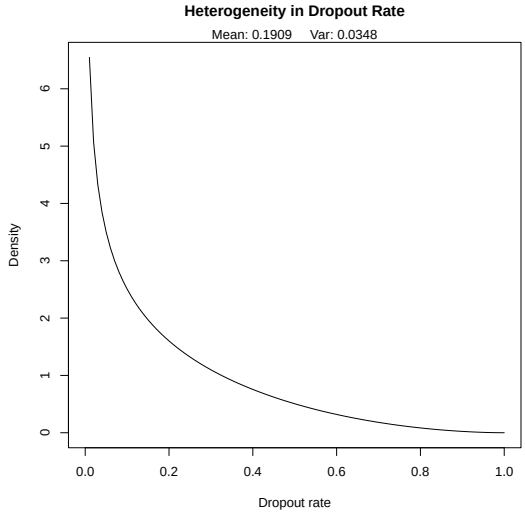

Figure 15: Dropout rate heterogeneity of estimated parameters.

this example, let's say we are interested in estimating the number of repeat transactions we expect a newly-acquired customer to make in a period of 10 years. The same rules that we used for Pareto/NBD functions also apply to BG/BB functions: since we used years to estimate the parameters, we stick to years to represent time periods.

```
bgbb.Expectation(params, n=10);

# [1] 3.179805
```

But we want to be able to say something about our existing customers, not just about a hypothetical customer to be acquired in the future. Once again, we use conditional expectations for a holdout period of 10 years. I am going to do this for 2 customers: A, who made 0 transactions in the calibration period; and B, who made 4 transactions in the calibration period, with the last transaction occuring in the 5th year.

```
# customer A
n.cal = 6
n.star = 10
x = 0
t.x = 0
bgbb.ConditionalExpectedTransactions(params, n.cal,
                                     n.star, x, t.x)

# [1] 0.1302169

# customer B
x = 4
t.x = 5
bgbb.ConditionalExpectedTransactions(params, n.cal,
                                     n.star, x, t.x)

# [1] 3.627858
```

As expected, B's conditional expectation is much higher than A's. The point I am trying to make, however, is that there are 3464 A's in this dataset and only 284 B's—you should never ignore the zeroes in these models.

5.4 Plotting/ Goodness-of-fit

Figure 1, is the first plot to test the goodness-of-fit: a simple calibration period histogram.

```
bgbb.PlotFrequencyInCalibration(params, rf.matrix)
```

As with the equivalent Pareto/NBD plot, keep in mind that this plot is only useful for an initial verification that the fit of the BG/BB model is not terrible.

The next step is to see how well the model performs in the holdout period. When we used `dc.ElogToCbsCbt` earlier, we ignored a lot of the data it generated. It is easy to get the holdout period frequencies from that data:

```
holdout.cbs <- simData$holdout$cbs
x.star <- holdout.cbs[,"x.star"]
```

At this point, we can switch from simulated data to the real discrete data (donation data) provided with the package. The holdout period frequencies for the donations data is included in the package. Using this information, we can generate figure 16, which compares actual and conditional expected transactions in the holdout period. It bins customers according to calibration period frequencies.

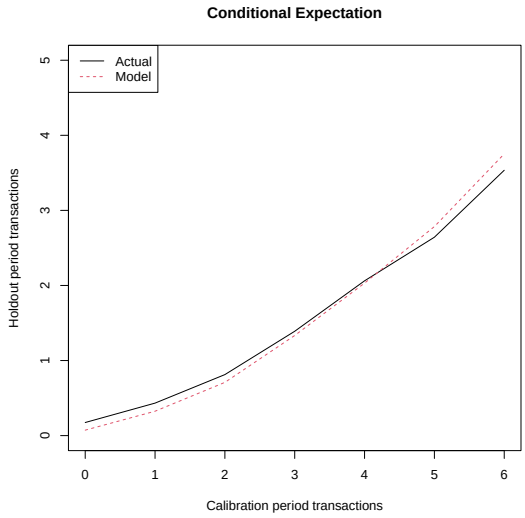

Figure 16: Actual vs. conditional expected transactions in the holdout period, binned by calibration period frequency.

```
n.star <- 5 # length of the holdout period
x.star <- donationsSummary$x.star
pdf(file = 'bgbbCondExpComp.pdf')
comp <- bgbb.PlotFreqVsConditionalExpectedFrequency(params, n.star,
                                                    rf.matrix, x.star)
dev.off()

# pdf
#   2

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

#           freq.0       freq.1       freq.2      freq.3      freq.4      freq.5
# act 1.743649e-01    0.4328031    0.8125874    1.389862    2.061776    2.642521
# exp 7.286318e-02    0.3248841    0.7089417    1.333666    2.031203    2.784482
# bin 3.464000e+03 1823.0000000 1430.0000000 1085.000000 1036.000000 1063.000000
#          freq.6
# act    3.534497
# exp    3.752500
# bin 1203.000000
```

Since the BG/BB model uses discrete data, we can also bin customers by recency. Figure 17 shows the fit of holdout period frequencies, with customers binned in this manner.

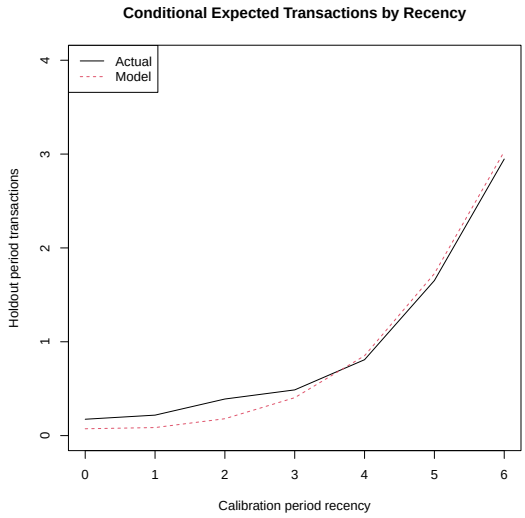

Figure 17: Actual vs. conditional expected transactions in the holdout period, binned by calibration period recency.

```
pdf(file = 'bgbbCondExpCompRec.pdf')
comp <- bgbb.PlotRecVsConditionalExpectedFrequency(params, n.star,
                                                   rf.matrix, x.star)
dev.off()

# pdf
#   2

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

#            rec.0        rec.1       rec.2       rec.3       rec.4       rec.5
# act 1.743649e-01    0.2181485   0.3898876   0.4872521   0.8088685    1.652289
# exp 7.286318e-02    0.0856939   0.1798423   0.4041111   0.8510811    1.726323
# bin 3.464000e+03 1091.0000000 890.0000000 706.0000000 654.0000000 1136.000000
#           rec.6
# act    2.946886
# exp    3.027190
# bin 3163.000000
```

Another plotting function provided by the BTYD package is the tracking function—binning the data according to which time period transactions occurred in. Since we do not have the granular data available, we once again cannot get the required data from the event log; however, the process will be exactly the same as described in section 3.4. The annual transaction data is provided by

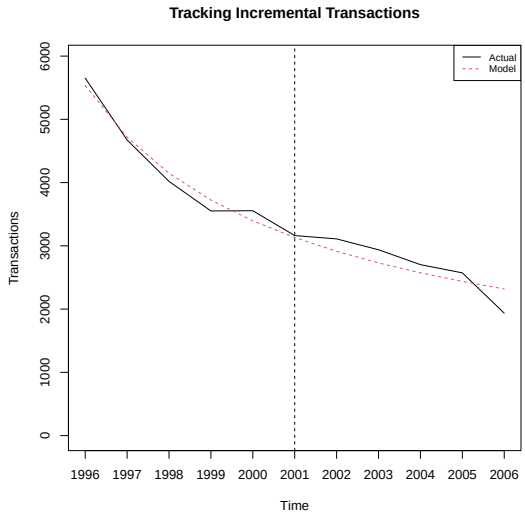

Figure 18: Actual vs. expected incremental purchasing behaviour.

the package.

```
inc.track.data <- donationsSummary$annual.trans
n.cal <- 6
xtickmarks <- 1996:2006
pdf(file = 'bgbbTrackingInc.pdf')
inc.tracking <- bgbb.PlotTrackingInc(params, rf.matrix,
                                     inc.track.data,
                                     xticklab = xtickmarks)
dev.off()

# pdf
#   2

rownames(inc.tracking) <- c("act", "exp")
inc.tracking

#         [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
# act 5652.000 4674.000 4019.000 3552.000 3555.000 3163.000 3110.000 2938.000
# exp 5535.833 4717.059 4147.621 3724.665 3395.888 3131.647 2913.778 2730.482
#         [,9]    [,10]    [,11]
# act 2703.000 2573.000 1936.000
# exp 2573.731 2437.854 2318.727
```

Figure 18 shows remarkable fit, but we can smooth it out for a cleaner graph

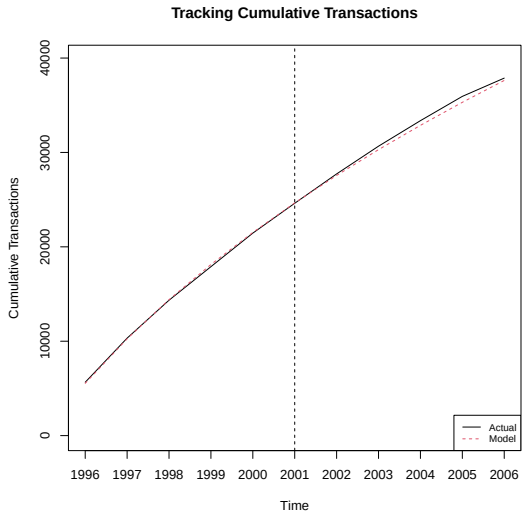

Figure 19: Actual vs. expected cumulative purchasing behaviour.

by making it cumulative (as shown in figure 19).

```
cum.track.data <- cumsum(inc.track.data)
pdf(file = 'bgbbTrackingCum.pdf')
cum.tracking <- bgbb.PlotTrackingCum(params, rf.matrix, cum.track.data,
                                     xticklab = xtickmarks)
dev.off()

# pdf
#   2

rownames(cum.tracking) <- c("act", "exp")
cum.tracking

#         [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
# act 5652.000 10326.00 14345.00 17897.00 21452.00 24615.00 27725.00 30663.00
# exp 5535.833 10252.89 14400.51 18125.18 21521.07 24652.71 27566.49 30296.97
#        [,9]    [,10]    [,11]
# act 33366.0 35939.00 37875.00
# exp 32870.7 35308.56 37627.29
```

6 Further analysis

The package functionality I highlighted above is just a starting point for working with these models. There are additional tools, such as functions for discounted

expected residual transactions (the present value of the remaining transactions we expect a customer to make) and an implementation of the gamma-gamma spend model, which may come in useful for customer analysis. Hopefully you now have an idea of how to start working with the BTYD package - from here, you should be able to use the package's additional functions, and may even want to implement some of your own. Enjoy!

References

Fader, Peter S., and Bruce G.S. Hardie. "A Note on Deriving the Pareto/NBD Model and Related Expressions." November. 2005. Web.
<http://www.brucehardie.com/notes/008/>

Fader, Peter S., Bruce G.S. Hardie, and Jen Shang. "Customer-Base Analysis in a Discrete-Time Noncontractual Setting." *Marketing Science*, **29(6)**, pp. 1086-1108. 2010. INFORMS.
<http://www.brucehardie.com/papers/020/>

Fader, Peter S., Hardie, Bruce G.S., and Lee, Ka Lok. ""Counting Your Customers" the Easy Way: An Alternative to the Pareto/NBD Model." *Marketing Science*, **24(2)**, pp. 275-284. 2005. INFORMS.
<http://brucehardie.com/papers/018/>