1 Cookbook for Monitoring Models

In R, the functions in this package are made available with

```
> library("monitor")
```

This section gives a brief recipe for building short term forecasting models. It is intended to be self-contained although there are references to other sections for additional information.

The term "monitoring" comes from the fact that one is often trying to monitor the current state of the economy based on data from prior periods, since there is typically some lag before statistical agencies release data for the current period. The steps, explained in more detail below, are:

1/ specify the data series to use in the model

2/ estimate a model and confirm that it is reasonable

3/ repeat 1 and 2 if other series are to be considered for competing models (beware that fishing can be dangerous)

4/ run the monitoring program to produce forecasts

and optionally

5/ set up an automatic program to run the monitoring program and distribute results

This library use the TS PADI interface explained in more detail in an appendix. For example purposes it is assumed that the data can be retrieved from an "economic time series" (ets) server. The examples use names of series which are used internally at the Bank of Canada and are probably not available elsewhere. Start S/R and open a graphics window with

```
> x11()
```

If running remotely it may be necessary to use an argument like "-display YourWorkstation:0.0" to display on your workstation. A few more details on running S/R are given in Section 2 of this guide.

1.1 Step 1- specify the data

The data is specified in an variable which indicates the name of the series, the source, any transformations which should be applied, and possibly some other options. For more details see the section on *TSdata* in the guide for dse1. An example of a model which contains two outputs and no inputs is

```
> if (require("dsepadi")) cbps.manuf.data2.ids <- TSPADIdata2(output = list(c("ets",
      "", "i37013", "percentChange", "cbps.prod."), c("ets", "",
      "i37005", "percentChange", "manuf.prod.")), pad.start = FALSE,
      pad.end = TRUE)
```

With the above, the data will be converted to percent change when it is read from the database. The default behaviour for data retrieval is to trim all series to the same length. The length is such that there are no missing values on the ends. pad.start and pad.end can be used to modify this behaviour. With pad.end=TRUE all series are padded on the end with NAs to give a length which will include the most recent data value from any series. This is preferred for forecasting but the NAs have to be trimmed with trimNA for estimation procedures. The data is actually retrieved from the database with

```
> if (require("padi") && checkPADIserver("ets")) cbps.manuf.data2 <- freeze(cbps.manuf.data2
```

This example and others below will not work without a database server that provides the indicated data. The *if* in the above allows automatic example checking to work (at the Bank of Canada).

The following example specifies one input series and one output series. It uses an alternate constructor (TSPADIdata vs. TSPADIdata2) which takes arguments in a different format. (The result is the same but different styles sometimes seem more convenient.)

```
> manuf.data.ids <- TSPADIdata(input = "lfsa455", input.transforms = "percentChange",
      input.names = "manuf.emp.", output = "I37005", output.transforms = "percentChange",
      output.names = "manuf.prod.", server = "ets", pad.start = FALSE,
      pad.end = TRUE)
> if (require("padi") && checkPADIserver("ets")) manuf.data <- freeze(manuf.data.ids)
```

The data can be plotted with

```
> if (require("padi") && checkPADIserver("ets")) tfplot(manuf.data)
```

In this example the plot shows missing data in the middle. In this somewhat unusual case it is necessary to trim the beginning of the data set to remove the portion up to the end of the missing data. This could be done with

```
> if (require("padi") && checkPADIserver("ets")) manuf.data <- tfwindow(manuf.data,
      start = c(1976, 2))
```

However, the trimming would have to be repeated each time the data is updated from the database, which is especially inconvenient for automatic procedures described further below. A better way is to set the starting period for retrieved data with

```
> manuf.data.ids <- modify.TSPADIdata(manuf.data.ids, start = c(1976,
      2))
```

then when data is retrieved with

```
> if (require("padi") && checkPADIserver("ets")) manuf.data <- freeze(manuf.data.ids)
```

it will start after the missing data. The start can also be specified with the argument start for the function TSPADIdata.

A more detailed plot of the last portion of the data can be produced with

```
> if (require("padi") && checkPADIserver("ets")) tfplot(manuf.data,
      start. = c(1995, 11))
```

Note the "." after start is part of the name of the argument. It is often not necessary since truncated arguments usually match without problem, but is required in the case of tfplot so that the argument is not confused with the function start. To specify and retrieve data with two input series and one output series

```
> cbps.manuf.data.ids <- TSPADIdata(input = c("lfsa462", "lfsa455"),
      input.transforms = "percentChange", input.names = c("cbps.emp.",
          "manuf.emp"), output = "i37013", output.transforms = "percentChange",
      output.names = "cbps.prod.", start = c(1976, 2), server = "ets",
      db = "", pad.start = FALSE, pad.end = TRUE)
> if (require("padi") && checkPADIserver("ets")) cbps.manuf.data <- freeze(cbps.manuf.data.i
```

To specify and retrieve data with one input variable and two output variable

```
> cbps.manuf.data3.ids <- TSPADIdata(input = "lfsa462", input.transforms = "percentChange",
      input.names = "cbps.emp.", output = c("i37013", "i37005"),
      output.transforms = c("percentChange", "percentChange"),
      output.names = c("cbps.prod.", "manuf.prod."), start = c(1976,
          2), server = "ets", db = "", pad.start = FALSE, pad.end = TRUE)
> if (require("padi") && checkPADIserver("ets")) cbps.manuf.data3 <- freeze(cbps.manuf.data3
```

Setting start is only necessary because of this rather unusual case were there are missing values in the middle of one series

1.2 Step 2 - estimate model

At this point it may be useful to make S/R prompt for a return before each new graph is produced. This is done with

```
> par(ask = TRUE)
```

A model can be estimated with various estimation techniques, some of which are described in Section 6. For example:

```
> if (require("padi") && checkPADIserver("ets")) manuf.model <- bft(trimNA(manuf.data))
```

This uses a "brute force technique" described in Gilbert (1995). It might take some time to run. It uses a default maximum number of lags of 12. The estimation is faster if a smaller number of lags is specified using

```
> if (require("padi") && checkPADIserver("ets")) manuf.model <- bft(trimNA(manuf.data),
      max.lag = 5)
```

By default the bft procedure prints information as it proceeds. This can be stopped using

```
> if (require("padi") && checkPADIserver("ets")) manuf.model <- bft(trimNA(manuf.data),
      verbose = FALSE, max.lag = 5)
```

To display the parameters of the estimated model just type the name of the variable in which it was stored:

```
> if (require("padi") && checkPADIserver("ets")) manuf.model
```

and to plot it:

```
> if (require("padi") && checkPADIserver("ets")) {
      tfplot(manuf.model)
      tfplot(manuf.model, start. = c(1990, 1))
      tfplot(manuf.model, start. = c(1995, 1))
  }
```

Models for the other specified data sets can be estimated in the same way:

```
> if (require("padi") && checkPADIserver("ets")) {
      cbps.manuf.model <- bft(trimNA(cbps.manuf.data), verbose = FALSE)
      tfplot(cbps.manuf.model)
      tfplot(cbps.manuf.model, start. = c(1995, 1))
  }
```

To forecast with the model using all available data (This example is artificially truncated with tfwindow because some of the data has been discontinued.)

```
> if (require("padi") && checkPADIserver("ets")) {
      z <- forecast(TSmodel(manuf.model), tfwindow(manuf.data,
          end = c(1995, 1)), conditioning.inputs = tfwindow(inputData(manuf.data),
          end = c(1996, 12)))
      tfplot(z, start. = c(1995, 1))
  }
```

To see the forecast use

```
> if (require("padi") && checkPADIserver("ets")) {
      forecasts(z)[[1]]
      tfwindow(forecasts(z)[[1]], start = c(1996, 3))
  }
```

Forecasting is discussed in the dse2 Guide.

To evaluate how well the model does at forecasting, look at the covariance of the forecast error at different horizons with

```
> if (require("padi") && checkPADIserver("ets")) {
      fc <- forecastCov(manuf.model)
      tfplot(fc)
  }
```

It is also good to consider how well the forecast does relative to a zero and a trend forecast:

```
> if (require("padi") && checkPADIserver("ets")) {
      fc <- forecastCov(manuf.model, zero = TRUE, trend = TRUE)
      tfplot(fc)
  }
```

The above forecast error analysis is done within the sample which was used for estimating the model. An out-of-sample forecast error analysis is typically a better indication of how well the model will really do. This can be done by using tfwindow to truncate the data to a subset for estimation and then evaluate the forecast error on the remainder. Another compromise, which is attractive when short data sets are involved, is to do an out-of-sample evaluation of the performance of an estimation procedure, and then hope that the procedure will continue to estimate good models when the whole data set is used.

```
> if (require("padi") && checkPADIserver("ets")) {
      outfc <- outOfSample.forecastCovEstimatorsWRTdata(trimNA(manuf.data),
          estimation.sample = 0.5, estimation.methods = list(bft = list(verbose = FALSE),
              estVARXls = NULL), trend = TRUE, zero = TRUE)
      tfplot(outfc)
  }
```

The bft procedure is generally fairly good but it can sometimes be out performed by a simple least squares estimation, especially for univariate models. Its real strength is for multivariate models:

```
> if (require("padi") && checkPADIserver("ets")) {
      outfc <- outOfSample.forecastCovEstimatorsWRTdata(trimNA(cbps.manuf.data3),
          estimation.sample = 0.5, estimation.methods = list(bft = list(verbose = FALSE),
              estVARXls = NULL), trend = TRUE, zero = TRUE)
      tfplot(outfc)
  }
```

More details are given in Section 8.

Once a model has been chosen it can be re-used, rather than re-estimating each time there is a new data point. This is done by extracting the model from the object returned by the estimation procedure. This object is a model with data and some estimation information. If you want to use different data then the data needs to be retrieved again using the variable which indicates the source. For example

```
> if (require("padi") && checkPADIserver("ets")) new.data <- freeze(manuf.data.ids)
```

To run the model and get one-step-ahead predictions with the new data use

```
> if (require("padi") && checkPADIserver("ets")) z <- l(TSmodel(manuf.model),
      trimNA(new.data))
```

Or the data retrieval can be done in the same step with

```
> if (require("padi") && checkPADIserver("ets")) {
      z <- l(TSmodel(manuf.model), trimNA(tfwindow(freeze(manuf.data.ids),
          start = c(1976, 2))))
      tfplot(z)
      tfplot(z, start. = c(1995, 8))
  }
```

Forecasts more than one-step-ahead require input series up to the horizon for which the forecast is to be produced. To run the model and get forecasts when more input than output data is available [tfwindow(..., end=c(1996,1)) is used in this example to simulate the situation. The data series have been terminated, so this example needs to be redone.]:

```
> if (require("padi") && checkPADIserver("ets")) {
      z <- forecast(TSmodel(manuf.model), tfwindow(trimNA(new.data),
          end = c(1996, 1)), conditioning.inputs = trimNA(inputData(new.data)))
      tfplot(z, start. = c(1995, 6))
  }
```

The effect of this is to trim NAs from input separately from output so that input will not be truncated to the same ending period as output. If you actually want the numbers rather than plots of the data use

```
> if (require("padi") && checkPADIserver("ets")) forecasts(z)[[1]]
```

or

```
> if (require("padi") && checkPADIserver("ets")) tfwindow(forecasts(z)[[1]],
      start = c(1996, 2))
```

will print values starting in the second period of 1996.

The horizon for a model with no inputs is determined by the argument horizon, which has a default value of 36. For a model which requires input (conditioning) data, the horizon for the forecast is determined by the input data, conditioning.inputs or conditioning.inputs.forecasts. If none of these are supplied then the argument horizon is used to replicate the last period of input data to the indicated horizon.

At the Bank of Canada PADI is an interface to a Fame server. The forecast data can be put into a Fame database with

```
> if (require("padi") && checkPADIserver("ets")) putpadi(forecasts(z)[[1]],
      dbname = "nameofdatabase.db", series = seriesNamesOutput(z))
```

In the above

```
> if (require("padi") && checkPADIserver("ets")) seriesNamesOutput(z)
```

extracts a character vector of the series names.

1.3 Step 3 - reconsider the data and model

The performance of alternative models on a given data set can be compared by looking at the forecast error covariance from forecastCov. Repeat the required parts of steps one and two and choose the model which does best at the horizons of interest. Sometimes the real purpose of a monitoring model is just to forecast one series (the series of primary interest). Other series are included only because they provide additional information for forecasting the series of primary interest. One disadvantage of including additional series is that it increases the number of parameters which must be estimated, and thus reduces the quality of the estimates. At this step you should reconsider what series are included for the model. Choose the model which does best on the series of primary interest (but see also ”Juice Functions”).

1.4 Step 4 - run the monitoring

During the S session, variables (e.g. models and data) are saved in a subdirectory .Data below the directory where you started S. (In R they are in the file .RData.) The variables will be available the next time S/R is started from the same subdirectory. One danger is that you can overwrite an existing variable just by assigning a new value to the name. Once you have a model to use for forecasting it is a good idea to save it in a separate file so it will not be lost by accident. The model manuf.model and the corresponding data identifiers can be saved in the file ”manuf.model.definition” with

```
> if (require("padi") && checkPADIserver("ets")) dump(c("manuf.model",
      "manuf.data.ids"), file = "manuf.model.definition")
```

If necessary they can then be retrieved with

```
> if (require("padi") && checkPADIserver("ets")) source("manuf.model.definition")
```

The model can be run to produce a forecast and mail the results to a list of recipients. The function to do this compares the current data to a previous copy of the data in order to determine if an updated forecast should be run. The comparison data is first initialized with

```
> if (require("padi") && checkPADIserver("ets")) manuf.previous.data <- freeze(manuf.data.ic
```

then in order to make the data look like it has changed

```
> if (require("padi") && checkPADIserver("ets")) outputData(manuf.previous.data)[1,
      1] <- NA
```

and to run the forecast and E-mail the results

```
> if (FALSE) {
      r <- simpleMonitoring(manuf.model, manuf.data.ids, manuf.previous.data,
```

```
        mail.list = "pgilbert@bank-banque-canada.ca", message.title = " Manufacturing Moni
        message.subject = "Manufacturing Monitoring", show.start = c(0,
            -3), report.variables = seriesNames(manuf.data.ids),
        data.sub.heading = " %chg %chg", message.footnote = " f - forecast value",
        data.tag = " ", forecast.tag = "f")
}
```

The status of the result can be checked with

```
> if (FALSE) r$status
```

and the comparison data should also be updated with

```
> if (FALSE) manuf.previous.data <- r$data
```

Especially for debugging purposes it is often useful to keep a more complete record of the data and model used to produce the forecast This can be done with the simpleMonitoring argument save.as which can be set to specify a file name. Setting save.as=paste("Manufacturing.monitoring.", make.names(date()), sep="") in the above would make a file name which includes a time stamp. Also, setting the argument run.again=TRUE will run the forecast without checking to see if the data has been updated.

The argument mail.list allows the output to be mailed to a list of recipients, but it may be more convenient to mail the result to a list server which can be used for distribution purposes. This may be easier to maintain, as the list server list of recipients can be changed at any time (and in automatic mode described next the program does not have to be restarted.)

1.5 Step 5 - automatic program to run the monitoring

To run the above and e-mail forecast directly from the Unix command prompt a shell script can be set up as follows:

Below it is assumed this is in a file called manufacturing. To run this automatically every 20 minutes from 7am to 10am the script

could be put in a file monitoring.daemon and then this can be started at the Unix prompt with the command

unix prompt: monitoring.daemon manufacturing

The disadvantage of this approach is that the overhead for starting Splus is fairly heavy and it may be difficult to use your computer for much else from 7am to 10am. (R may be better in this respect.) If you have direct access to the files used for the database then the script could be modified to check time stamps on the files and only run if the file date has changed. If database files are used to store many series, and not all are updated at the same time, then the savings will not be much. At the Bank of Canada another script called Data.trigger.daemon can be used to run a Fame procedure to check if the particular series have been updated, and then run manufacturing only in that case.