H2020-SFS-2018-2020

DECIDE

Data-driven control and prioritisation of non-EU-regulated contagious animal diseases

Deliverable 2.5

Warning systems

WP2 – Methods for data analysis and modelling

Authors Dan Børge Jensen (UCPH)
Leonardo Victor de Knegt (UCPH)
Lead participant UCPH
Delivery date 08 July 2025
Dissemination level Public
Type Other

Revision History

Author Name (Partner short name)	Description	Date
Leonardo Víctor de Knegt (UCPH) Dan Børge Jensen (UCPH)	Draft deliverable report	09.06.2025
Inge Santman-Berends (GD)	Revision 1	18.06.2025
Leonardo Víctor de Knegt (UCPH)	Upload of code	26.06.2025
Leonardo Víctor de Knegt (UCPH) Dan Børge Jensen (UCPH)	Final version	26.06.2025
Gerdien van Schaik (UU) Elisa Vrieze (LELY) Johannes Ripperger (accelCH)	Final checks and formatting	08.07.2025

Content

EXECUTIVE SUMMARY ... 6
1 INTRODUCTION ... 7
1.1 Background ... 7
1.2 Previous work leading to this deliverable ... 7
1.3 Objectives and content of the deliverable ... 8
2 DATA DESCRIPTION ... 9
2.1 Data collection and background ... 9
2.2 Variable dictionary ... 9
3 SHEWHART CONTROL CHARTS ... 11
3.1 Overview of the method ... 11
3.2 Assumptions of the Classical Shewhart Chart ... 11
3.3 Shewhart Three-Sigma Limits ... 11
3.4 The Four *Montgomery* (Western Electric) Rules ... 11
3.5 Illustrative Example ... 12
3.6 Practical Considerations ... 19
3.7 References ... 19
4 V-MASKS ... 20
4.1 Introduction to the method ... 20
4.2 The V-Mask Geometry ... 20
4.3 Assumptions ... 20
4.4 Relevant functions ... 21
4.5 Choosing realistic values for `dist` and `angle` ... 22
4.6 Plotting V-masks and alarms ... 24
4.7 Conclusion about the method ... 28
4.8 Further reading ... 29
5 TABULAR CUSUMS ... 30
5.1 Introduction to the method ... 30
5.2 Theory ... 30
6 R IMPLEMENTATION ... 31
6.1 Choosing k and h ... 32
6.2 Practical Tips ... 33

6.3 Conclusions about the method 36
6.4 References 36
7 CHOLESKY DECOMPOSITION AND THE MALAHANOBIS DISTANCE 37
7.1 Introduction to the method 37
7.2 Method details 37
7.3 R implementation 38
7.4 Illustrative Example 43
7.5 Discussion of the method 46
7.6 List of references 46
8 USING DLM OUTPUTS AS INPUTS FOR A RANDOM FOREST 47
8.1 Introduction to the example 47
8.2 Setting up your working environment 47
8.3 Training and validating the random forest 49
9 OVERALL CONCLUSION...52

Abbreviations

Abbreviation	Description
ACF	Autocorrelation factor
CUSUM	Cumulative Sum
DLM	Dynamic Linear Model
DGLM	Dynamic Generalized Linear Model
EU	European Union
H2020	Horizon 2020
MMA	Mean major accuracy
SCC	Shewhart Control Chart
WP	Work Package

Partner short names

Short name	Organisation
UU	Universiteit Utrecht
UCPH	Københavns Universitet
GD	Gezondheidsdienst voor Dieren BV
LELY	Lely Industries NV
accelCH	accelopment Schweiz AG

Executive Summary

This deliverable presents a walk-through example of implementation of early-warning tools for animal health monitoring, based on the output of state-space models developed with the help of Deliverables 2.1.("Open-source software tools to perform multivariate monitoring of time-series data") and 2.2.("Open-source software tools to perform multi-level monitoring of time-series data")-. Health monitoring plays a crucial role in modern animal production, increasingly relying on automated systems. In process control theory, unexpected changes signal that the system is out of control; in livestock production, behavioural shifts such as reduced activity, altered feeding (when not following contextual changes like movement from pasture to indoors or insemination), or increased mortality, may indicate the presence of disease, or changes in the environment which carry negative effects for herd health. As long as a model can simulate the observed normal state of a system, deviations from that model's expected predictions suggest that the system has moved away from normal functioning. Tools like milking robot sensors, weight scales, accelerometers, and behaviour-detecting cameras support monitoring, but are most effective when paired with early-warning systems. Statistical methods such as Shewhart Control Charts, V-masks, or cusums, as well as machine-learning techniques like Random Forests, can be applied to the forecast errors of state-space models to detect meaningful changes in animal health. These methods are valuable tools for monitoring animal diseases at farm, regional, or national level."

Objectives of the Deliverable

With the help of this deliverable, the reader will walk through example approaches of four methods (three statistical and one machine-learning-based) to develop warning systems based on the outputs of Dynamix Linear Models in R, with the final objective of detecting respiratory disease in calves. Pre-programmed functions and R scripts needed to implement those methods are also provided and can be found in the same repositories as this report.

Notice that the performances of the warning tools are not great when applied to the data used in these examples. We remind the reader that the objective of this deliverable is to serve as a handbook for how to apply the methods, and that the specific results when applied to these data are incidental to this purpose.

Activities

The deliverable presents the background theory for Shewhart Control Charts, v-masks, cusums and random forests, and illustrates how they can be used to automatically detect respiratory disease in calves, with software scripts that walk the user through examples to guide implementation.

The present deliverable was based on R codes written by Dan Børge Jensen.

Outcome

With the R scripts containing the walkthrough and the functions used to create early warning systems based on the output of DLMs, the reader should be able to develop systems that can be applied to their own datasets.

1 Introduction

1.1 Background

Health monitoring constitutes a fundamental aspect of contemporary animal production and increasingly depends on automated systems, to the same extent as industrial production. When comparing with industrial production, a situation is considered in control (or «normal») when the processes happen as expected. Out-of-control situations can be generated by failure in machinery, lack of components/ingredients or human error. In animal production, the normal state is defined by the measurements and behaviours observed when production and health are as expected. Changes in animal behaviour, like activity levels and patterns, feeding behaviour or changes in demeanour, are frequently indicative of changes in health status. For large-scale production types, like broilers or fish, sudden increases in mortality offer us a glimpse into the effects of disease outbreaks, or failure in environmental control systems (temperature, humidity, water pH, ventilation). In short, a model optimized for a dynamic system's normal state will predict accurately, as long as the system remains stable. Inaccurate forecasts indicate that the system has changed or is transitioning to an abnormal state.

Somatic cell count or electric conductivity measurements in automated milking robots, automatic scales to monitor weight, accelerometers measuring changes in activity, or cameras which detect clustering or other abnormal behaviours, are examples of systems used to monitor different animal species. The usefulness of those monitoring systems, however, is limited unless they are coupled with alarm systems that inform the responsible persons about unusual changes from the expected state, preferably at an early stage. Therefore, animal health monitoring at farm, region or country-level demands dynamic monitoring methods, where key variables are registered and analysed statistically on a continuous basis, so an alarm can happen quickly when any deviations from the norms occur.

The Dynamic Linear Models explored in the previous two DECIDE Deliverables offer a good framework for the establishment of such early warning systems, as they are designed to filter observations, and detect changes from normal situations. These changes can be further analysed using classical process monitoring methods like Shewhart Control Charts, which constitute simple but effective early warning systems, when coupled with pattern deviation detection techniques like V-masks or sums of cumulative forecast errors, (cusums). They are, however, not the only tools available for this finality, with more recent methods like Random Forests being more and more used to classify data points as normal or altered during monitoring.

It was, therefore, considered relevant for DECIDE to present and describe the methods cited above, and provide examples of early-warning systems in which they can be applied to monitor production diseases. In this deliverable the added value of the different methods is explored using an example of data collected as part of the DECIDE project.

1.2 Previous work leading to this deliverable

In 2022, the University of Copenhagen held a one-week workshop on Dynamic Linear Models (DLMs) for partners of the DECIDE Project, as part of WP2. Dynamic Linear Models predict the expected outcome of a certain parameter e.g. milk production at t=1 based on the observed milk production in that same unit in all preceding weeks in the data. The workshop focused on the development of univariate DLMs in R, for monitoring and detection of state changes in data series. Some of the course work continued in partnership after the workshop was finished, with the Cattle Case Study in collaboration with LELY as an example.

In September 2023, a second one-week workshop was held in Copenhagen as a precursor for the second deliverable (D2.2 – Open-source software tools to perform multi-level monitoring of time-series data) with focus on multi-level monitoring of time series data using DLMs.

In September 2024, a third one-week workshop was organised in Copenhagen, with a focus on monitoring of non-Gaussian time series data using Dynamic Generalized Linear Models (DGLMs), and a brief introduction to classic univariate early warning systems.

A fourth and last workshop was held during the General Assembly in Uppsala in June 2025, with a focus on early warning systems.

1.3 Objectives and content of the deliverable

The main objective of the present deliverable is to build on the knowledge developed through Deliverables 2.1. and 2.2., as well as during the workshops organised by WP2 as part of the DECIDE project. It includes a report written by Dan Børge Jensen and Leonardo Víctor de Knegt, an anonymised dataset provided by LELY, and two R scripts produced by Dan Børge Jensen. The R scripts are applied to the example dataset to monitor respiratory disease in dairy calves, making it possible for uni- or multivariate DLMs and DGLMs built previously to be added with early-warning systems based on both statistical and machine learning techniques. The report contains the theoretical framework for the methods used, as well as a step-by-step walkthrough for the scripts.

The scripts to walk the user through the use of Shewhart control Charts, V-Masks, Cusums and Random Forests, as well as the datasets they are applied to, were collected in a zip file (Deliverable_2.5_Rscripts.zip). Scripts containing exercises and examples are also available in the same compressed file. Functions and code lines are written in a generalized format to facilitate their application in other situations. It is needed that the data is in the same format as the example datasets "Data_restandardized.RDS " and "DLM_preprocessed_data__NEW_WithoutActivity.RDS ", and that the user has a working knowledge of the methods used.

The scripts and datasets (all with names starting with "D25_") can be found on the DECIDE GitHub repository, at:

https://github.com/decide-project-eu/WP2-Deliverables

The zip file Deliverable_2.5_Rscripts.zip containing all scripts and datasets is available on the project internal document storage system along with this deliverable (accelCLOUD) at:

https://cloud.accelopment.com/index.php/apps/files/?dir=/DECIDE/Deliverables/Submitted/D2.5_EarlyWarningSystems

Users can get support when using the scripts by contacting the UCPH group at daj@sund.ku.dk, and if any relevant developments and updates to the codes occur, the most recent versions will be uploaded to both repositories.

2 Data description

2.1 Data collection and background

The study took place on 4 dairy farms in which LELY is testing sensoring techniques since early 2019. Individual feeding (drinking) behaviour is being collected continuously and has been made available for the project for the period Sept-Dec 2023. All calves were fed with milk replacer using automatic feeders. Three of the farms used individual feeding plans in a restrictive diet system, while one let the calves feed *ad libitum*. The calves were introduced to this system at around 14 days old and were weaned at around 65 days old. The dataset contains data from 100 calves, with 25, 26, 40 and 9 calves from Herd 1, Herd 2, Herd 3 and Herd 4, respectively. Farm names and calf numbers were anonymised to unique herd and calf identifiers and contextual variables include the date of each observation, the age of the calf in days, and the number of days since the calf was housed in that section.

The automatic feeders collected data on the amount of milk replacer consumed, the speed at which it was consumed and the number of visits to the feeder during and after the daily allowance was finished.

In the study period, the group-housed preweaned calves from these farms were monitored for signs of sickness twice a week during regular farm visits conducted by two students, using the University of Wisconsin-Madison School of Veterinary Medicine calf health scoring guide (https://www.vetmed.wisc.edu/fapm/svm-dairy-apps/calf-health-scorer-chs/). According to this system, coat appearance, coughing, ocular discharge, nasal discharge, respiration and manure structure were scored 0-3, where zero means the absence of alterations, and three means severe alterations. Body temperature of the calves was also measured. Based on this information, variables implying sickness were created, with a focus on respiratory disease.

A DLM was trained on healthy calves and used to filter independent chosen consumption measurements for all calves. The modelling is performed in a per-herd cross-validation scheme:

1. **Hold out** one entire herd as the *test* herd.
2. **Train** the DLM on the remaining herds (healthy calves only).
3. Apply the trained DLM to every calf in the held-out herd.
4. Repeat for each herd so that every herd serves exactly once as the test set.

The model outputs were added as columns to the dataset. This mirrors a real-world scenario where a newly enrolled farm is evaluated with knowledge learned elsewhere.

2.2 Variable dictionary

Variable name	Description	Type
Herd	Herd number (1,2,3 or 4)	Character
calf_herd	Individual calf number. Composed by a four-digit number, followed by a dot and the herd number	Character
Date	Date of collection of a data point	Date(yyy-mm-dd)
DayOfYear	Day of the year for the corresponding date	Numeric
age_days	Age of the calf in days	Integer
DaySinceHousingDate	Days since the calf was moved to this section	Integer
Bodytemperature	Calf body temperature	Numeric

Anysickness	Any symptoms (1/0)	Binary
Sick	Sick based on case definition (1/0)	Binary
SickOrHealthy	Is TRUE if either Sick, Coughmax, Manuremax, Coat, Coughing, Ocular discharge, Nasal discharge, Respiration, or Manure structure has a health score greater than 0.	Binary
Coughmax	Maximum cough score observed for a calf	Numeric
Manuremax	Maximum manure structure score observed for a calf	Numeric
Coat	Wisconsin Calf Health Score (0-3)	Numeric
Coughing	Wisconsin Calf Health Score (0-3)	Numeric
Ocular discharge	Wisconsin Calf Health Score (0-3)	Numeric
Nasal discharge	Wisconsin Calf Health Score (0-3)	Numeric
Respiration	Wisconsin Calf Health Score (0-3)	Numeric
Fecal score	Wisconsin Calf Health Score (0-3)	Numeric
consumption_liters	Volume of milk replacer (powder+water) consumed, standardized per herd)	Numeric
visits	Number of times the calf visited the feeder, standardized per herd	Numeric
visitswoent	Number of times the calf visited the feeder after their entitlement was fully consumed	Numeric
visitswent	Number of times the calf visited the feeder to consume their entitlement, standardized per herd	Numeric
DrinkingSpeed	Drinking speed, standardized per herd	Numeric
mt_*	Filtered means of corresponding consumption variables replacing the * on the column name	Numeric
mt_d.*	Filtered trend components of corresponding consumption variables replacing the * on the column name	Numeric
ft_*	One-step forecasts of corresponding consumption variables replacing the * on the column name	Numeric
et_*	Forecast errors of corresponding consumption variables replacing the * on the column name	Numeric
ut_*	Standardized forecast errors of corresponding consumption variables replacing the * on the column name	Numeric
Qt_*	One-step variance of corresponding consumption variables replacing the * on the column name	Numeric

3 Shewhart control charts

3.1 Overview of the method

Shewhart (often pronounced **"SHOO-art"**) control charts are graphical tools used in statistical process control (SPC) to decide whether a sequence of measurements arises from a stable system or whether an **alarm** (signal of special-cause variation) should be triggered. When these charts are applied to DLMs—which supply real-time predictions and residuals for streaming data—they serve as a low-latency layer that converts probabilistic forecasts into crisp "in-control/out-of-control" decisions.

3.2 Assumptions of the Classical Shewhart Chart

1. **Independence** – Successive charted statistics (e.g., individual observations, subgroup means, or residuals) are mutually independent.
2. **Identically distributed** – Under the in-control state the study subjects have common mean μ and variance σ^2.
3. **Normality (for $\bar{X}$ charts)** – The charted statistic is approximately Normal; this is guaranteed for subgroup means by the Central Limit Theorem.
4. **Process stability at start-up** – The historical data used to estimate μ and σ is itself in-control.

Violating these assumptions (especially independence) inflates false-alarm rates. In a DLM context, independence can often be restored by charting *one-step-ahead forecast errors* rather than raw observations.

3.3 Shewhart Three-Sigma Limits

For a generic statistic T_t that is in-control with mean μ and standard deviation σ, the classical 3-sigma chart uses

$$\begin{aligned} \text{Center Line (CL)} &= \mu, \\ \text{Upper Control Limit (UCL)} &= \mu + k\,\sigma, \\ \text{Lower Control Limit (LCL)} &= \mu - k\,\sigma, \quad k = 3. \end{aligned}$$

The chart signals **out-of-control** when any plotted point falls outside [LCL, UCL].

3.3.1 Linking to a DLM

Let y_t be the observation at time t and $\hat{y}_{t|t-1}$ its one-step forecast with variance Q_t from the Kalman filter. Define the *standardized forecast error*

$$z_t = \frac{y_t - \hat{y}_{t|t-1}}{\sqrt{Q_t}} \sim \mathcal{N}(0,1) \quad \text{(in-control)}.$$

Use z_t as T_t in the formula above: CL = 0, UCL = 3, LCL = –3. This produces an **Individuals (X) chart** that is perfectly tuned to the predictive distribution generated by the DLM.

3.4 The Four *Montgomery* (Western Electric) Rules

Douglas C. Montgomery's *Introduction to Statistical Quality Control* promotes four supplementary rules that detect subtler patterns than a single 3-sigma point:

Rule	Description	Rationale
1	**One point outside $\pm 3\sigma$.**	Large, sudden shift.
2	**Two of three consecutive points outside $\pm 2\sigma$ (same side).**	Moderate sustained shift.
3	**Four of five consecutive points outside $\pm 1\sigma$ (same side).**	Small sustained shift.
4	**Eight (or nine) consecutive points on the same side of the center line.**	Persistent drift in mean.

A violation of *any* rule constitutes an alarm.

3.5 Illustrative Example

Here we will look at the data set *DLM_preprocessed_data__NEW_WithoutActivity.RDS*, in wich the variables “consumption_liters”, “visitswent”, “visits”, and “DrinkingSpeed” have been filtered using a multivariate DLM.

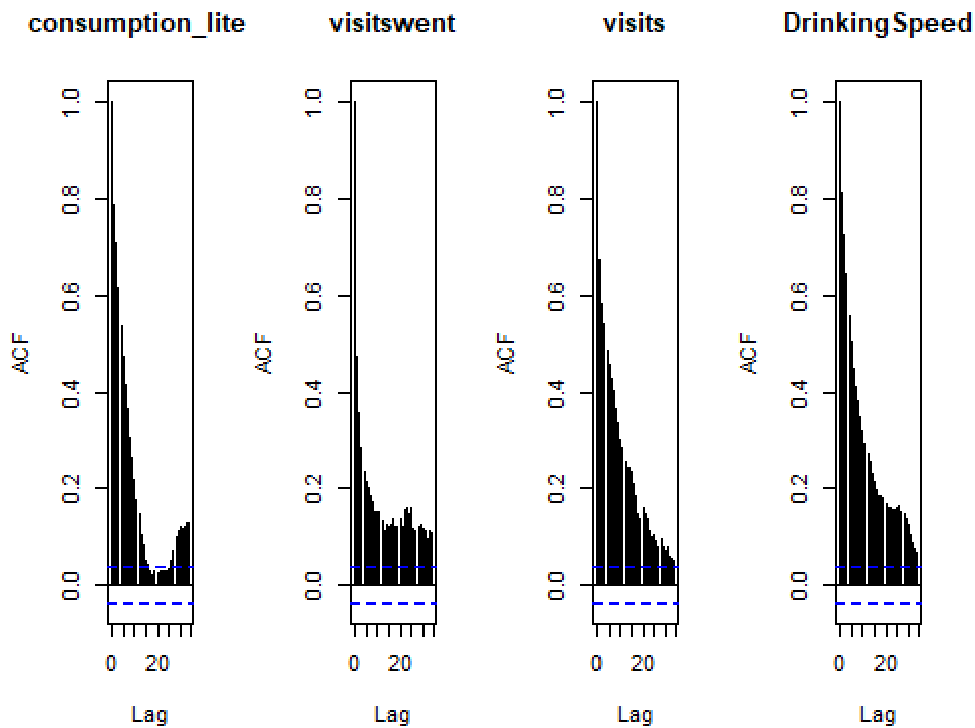
Furthermore, we will source the script *Functions for monitoring and filtering.R*.

```
res <- readRDS("DLM_preprocessed_data__NEW_WithoutActivity.RDS")
source('Functions for monitoring and filtering.R')
```

3.5.1 Checking for autocorrelation

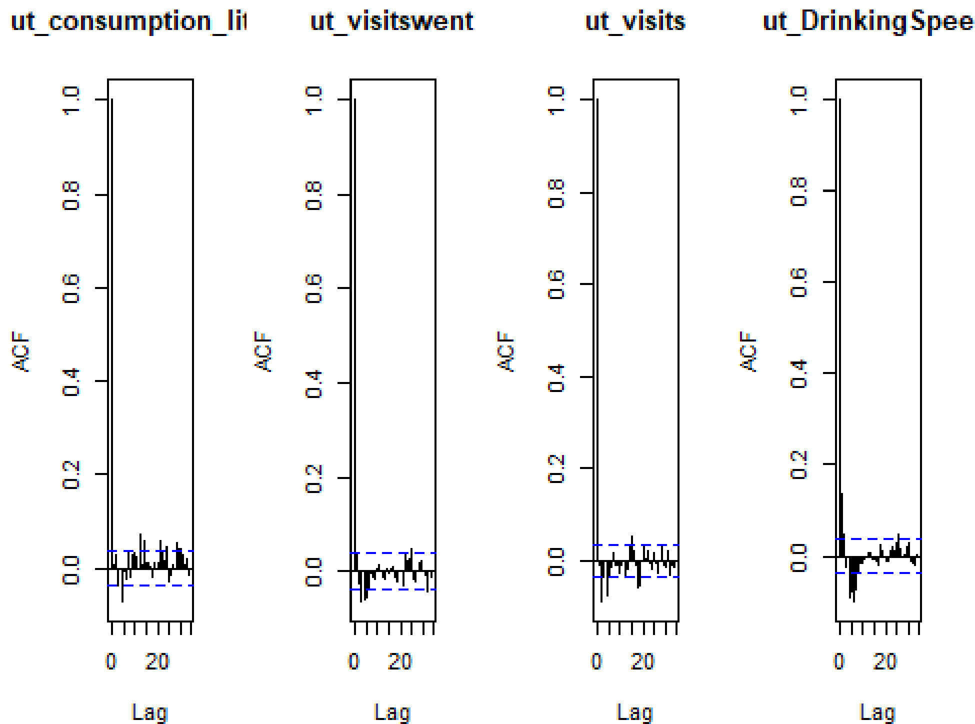
The DLM was applied to the variables “consumption_liters”, “visitswent”, “visits”, “DrinkingSpeed”. We can use the *acf* to confirm that the raw data do not live up to the assumption of each observation in the time series being mutually independent, i.e. there being no autocorrelations. The plot produced shows the autocorrelation factor (ACF) fluctuation over an increasing lag between the first and the current observation. In autocorrelated data, the ACF starts high and slowly decreases over time, as each observation is highly correlated with the previous, in a chain leading back to the first. In non-autocorrelated data, the ACF behaves randomly.

```
relevant.names <- c("consumption_liters", "visitswent", "visits", "DrinkingSpee
d")

par(mfrow=c(1,4))
for(name in relevant.names){
  acf(res[,name], main=name)
}
```



We can use the same function to confirm that the standardized forecast errors (ut) produced by the DLM has no autocorrelations. In the plot produced, the ACF for the standardized forecast errors resulting from modelling the different variables can be observed.

```
ut.names <- c("ut_consumption_liters", "ut_visitswent", "ut_visits", "ut_Drinki
ngSpeed")

par(mfrow=c(1,4))
for(name in ut.names){
  acf(res[,name], main=name)
}
```



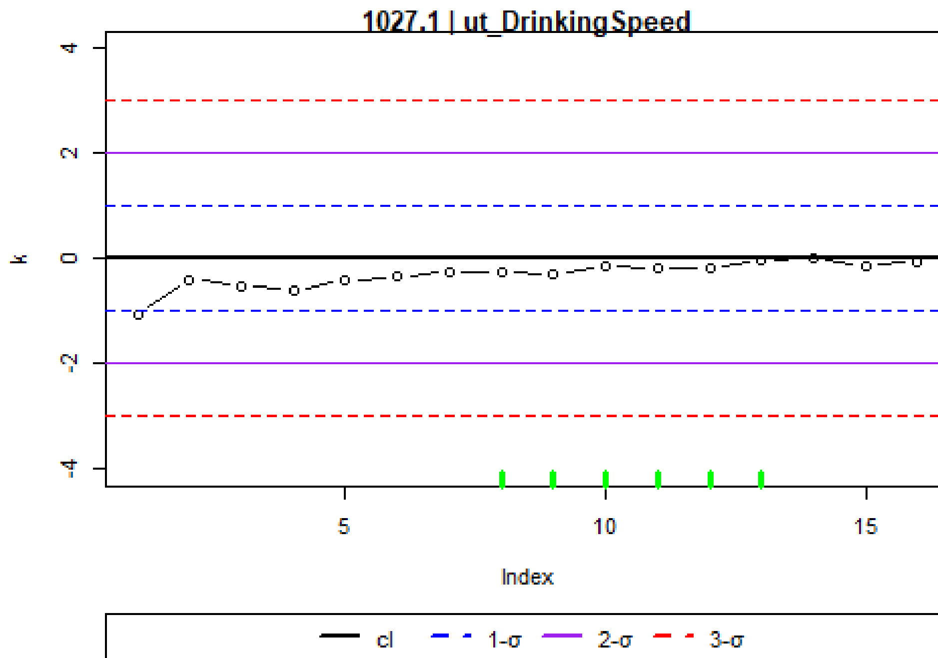
3.5.2 Applying Montgomery rules

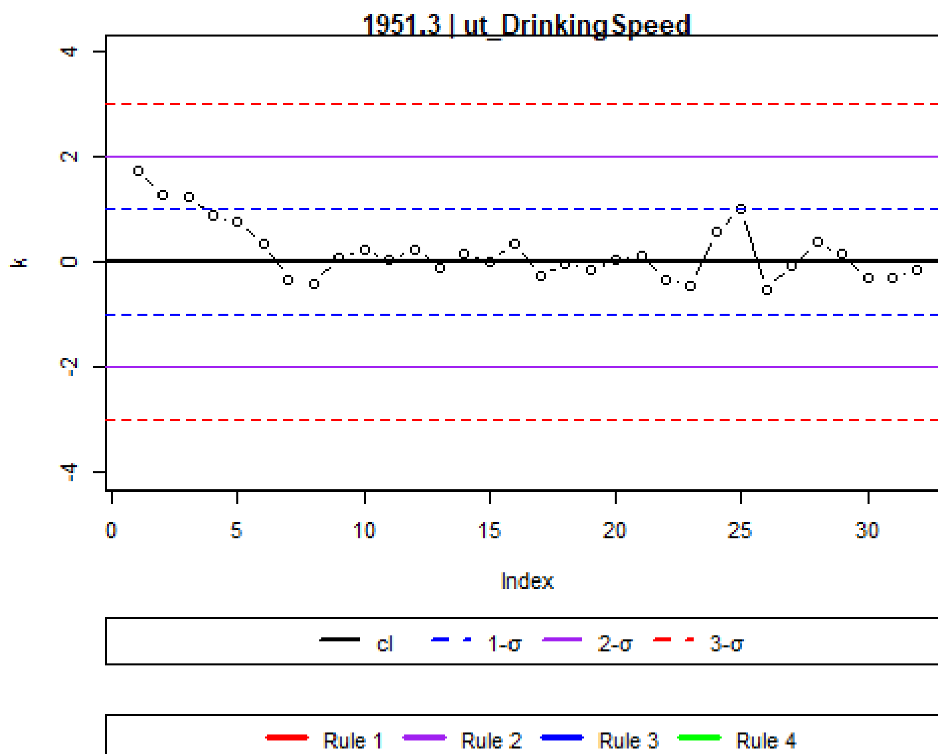
3.5.2.1 *Illustrative examples*

Initially, we can randomly select a healthy and a sick calf to apply the 4 Montgomery rules to for illustrative purposes. For this illustration, we will only apply the function to the *ut_consumption_litres* column.

In theory, the standardized forecast errors should follow a standard normal distribution with a mean of 0 and a standard deviation of 1. Therefore, we will first apply the *alarmsMontgomery* function with these settings. In the produced plots, observations which generate alarms according to each Montgomery rule are marked at the bottom, using colours. Red corresponds to rule 1, purple to 2, blue to 3 and green to 4.

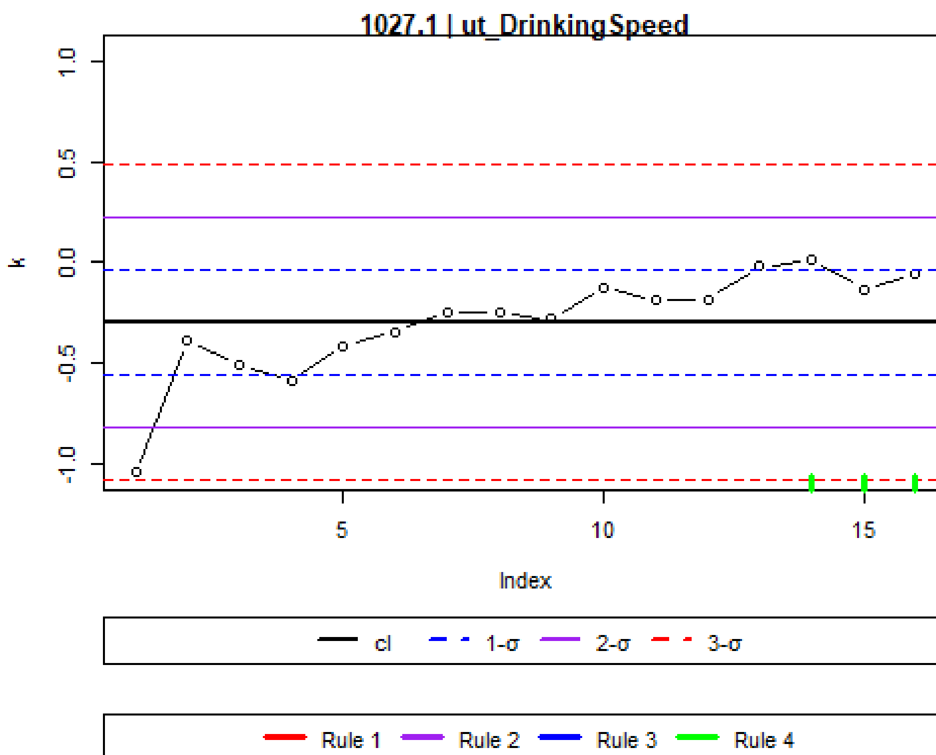
```
# Select a random healthy calf
set.seed(42)
healthy.calves <- unique(res$calf.herd[which(res$AnySickness == 0)])
healthy.calves <- sample(x = healthy.calves, size = 1)

# Select a random sick calf
set.seed(42)
sick.calves <- unique(res$calf.herd[which(res$AnySickness == 1)])
sick.calves <- sample(x = sick.calves, size = 1)

# Plot the selected calves
calves <- c(healthy.calves, sick.calves)

for(calf in calves){
    calf.set <- subset(res, res$calf.herd == calf)
    alarmsMontgomery(k = calf.set[,'ut_consumption_liters'], cl = 0, SD = 1,
Ylim = c(-4,4), Main = paste(calf, '|', name), plot.it = TRUE)
}
```

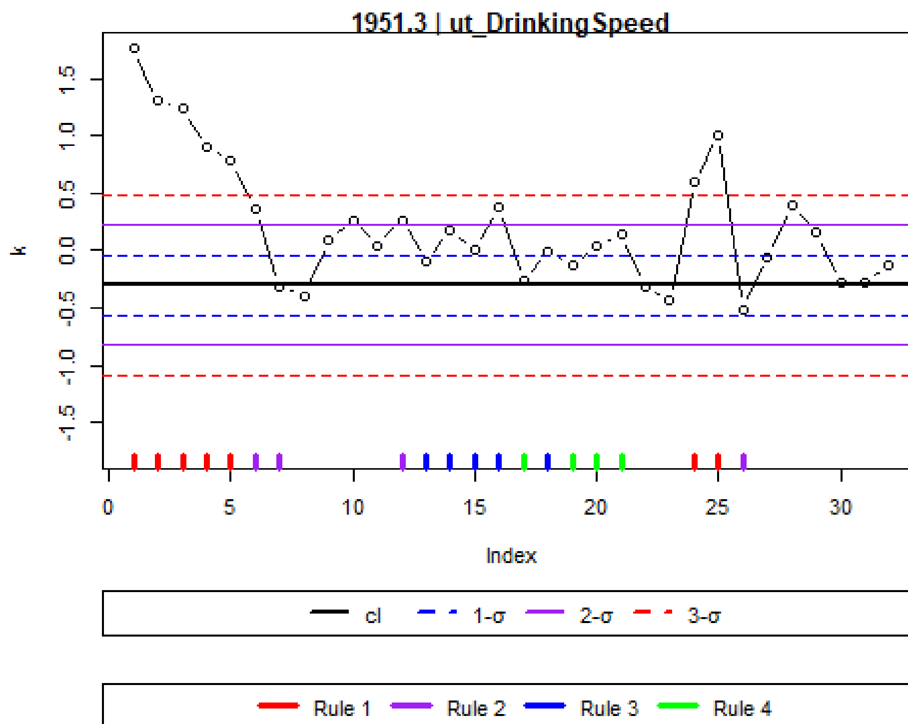

1027.1 | ut_DrinkingSpeed
k
Index
cl
1-σ
2-σ
3-σ
Rule 1
Rule 2
Rule 3
Rule 4


1951.3 | ut_DrinkingSpeed
k
Index
cl
1-σ
2-σ
3-σ
Rule 1
Rule 2
Rule 3
Rule 4

It may be the case that we do not trust that the standardized forecast errors follow a true standard normal distribution. In that case, we can estimate the mean and standard deviation from the healthy calves.

```
# Select only the healthy calves
healthy.set <- subset(res, res$calf.herd %in% healthy.calves)
Mean <- mean(healthy.set[,'ut_consumption_liters'])
SD <- sd(healthy.set[,'ut_consumption_liters'])

# Plot the selected calves
calves <- c(healthy.calves, sick.calves)

for(calf in calves){
    calf.set <- subset(res, res$calf.herd == calf)
    alarmsMontgomery(k = calf.set[,'ut_consumption_liters'], cl = Mean, SD = SD
, Ylim = NA, Main = paste(calf, '|', name), plot.it = TRUE)
}
```





This time we get many alarms for the sick calf, but not for the healthy calf.

3.5.2.2 *Assessing performance*

First, we will see what kind of performance we get if we apply the 4 Montgomery rules to each of the *ut* columns with the assumptions of standard normal distributions. After applying the *alarmsMontgomery* function to the forecast errors, we use the *getPerformance* to assess the performance of the alarms. The performance is assessed by using the Mean Major Accuracy, calculated as the average between the sensitivity and the specificity, thus providing a balanced appraisal of the model regarding those two features and how they can fluctuate in relation to each other. The ideal value for the MMA varies depending on the model objectives, and on the consequences of false positives and false negatives. In theory, an MMA of 0.5 indicates a model that is as bad in predicting as any random guess, and an MMA of 1 is absolutely perfect. When comparing different models or datasets, it is customary to choose the highest MMA as the best performance among your choices. Here the observations are the values in the *SickOrHealthy* column.

```
performance.all <- data.frame()
for(name in ut.names){
    alarms <- alarmsMontgomery(k = res[,name], cl = 0, SD = 1, Ylim = NA, Main
= paste(calf, '|', name), plot.it = FALSE)
    for(i in 1:ncol(alarms)){
      performance <- getPerformance(observations = res$SickOrHealthy, alarms[,i
])
      performance <- cbind('Name'=name, 'Rule'=colnames(alarms)[i], performance
)
      performance.all <- rbind(performance.all, performance)
    }

}
```

```
print(summary(as.numeric(performance.all$MMA)))

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.4700  0.5000  0.5050  0.5055  0.5112  0.5350

print(performance.all[which(performance.all$MMA == max(performance.all$MMA)),])

##                  Name    Rule  TP FP  TN   FN Sensitivity Specificity   MMA
## 20 ut_DrinkingSpeed AnyRule 238 46 804 1684        0.12        0.95 0.535
##    MMA 95 % CI
## 20 0.525-0.545
```

We see that the highest MMA for any names/rules combination is 53.5% (CI 0.525-0.545).

Next, we can now test the performance if we estimate the distribution from healthy calves. To do this, we will use the same per-herd cross-validation scheme as was used when training and applying the DLM.

```
# Iterate over each of the names
performance.all <- data.frame()
for(name in ut.names){

  alarms.all.name <- data.frame()
  for(herd in unique(res$Herd)){

    # Get the data from just this herd
    herd.set <- subset(res, res$Herd == herd)

    # Use only the healthy calves from the other herds as a learning set
    learning.set  <- subset(res, res$Herd != herd)
    learning.set <- subset(learning.set, learning.set$calf.herd %in% healthy.ca
lves )

    #Learn the distribution
    Mean <- mean(learning.set[,name])
    SD <- sd(learning.set[,name])

    # Make the alarms for this herd
    alarms <- alarmsMontgomery(k = herd.set[,name], cl = Mean, SD = SD, Ylim =
NA, Main = paste(calf, '|', name), plot.it = FALSE)
    alarms.all.name <- rbind(alarms.all.name, alarms)

  }

  # Get the performance for each rule on this name
   for(i in 1:ncol(alarms.all.name)){
      performance <- getPerformance(observations = res$SickOrHealthy, alarms.al
l.name[,i])
      performance <- cbind('Name'=name, 'Rule'=colnames(alarms.all.name)[i], pe
rformance)
      performance.all <- rbind(performance.all, performance)
    }
```

```
}

print(summary(as.numeric(performance.all$MMA)))

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
## 0.5000  0.5100  0.5150  0.5298  0.5437  0.5900 

print(performance.all[which(performance.all$MMA == max(performance.all$MMA)),])

##                 Name    Rule  TP  FP  TN   FN Sensitivity Specificity  MMA
## 20 ut_DrinkingSpeed AnyRule 719 158 692 1203        0.37        0.81 0.59
##    MMA 95 % CI
## 20 0.574-0.606
```

We see that the best MMA has increased to 59 (CI 0.57-0.61)%.

3.6 Practical Considerations

- **Autocorrelation** – If the DLM's residuals are not white noise, consider charting batches or using EWMA/CUSUM charts instead.
- **Phase I vs Phase II** – Estimate μ and σ from a stable *Phase I* period; then lock them for monitoring (*Phase II*).
- **Multiple streams** – For multivariate DLM outputs, build separate Shewhart charts.

3.7 References

1. Shewhart, W. A. (1931). *Economic Control of Quality of Manufactured Product.*
2. Montgomery, D. C. (2020). *Introduction to Statistical Quality Control* (9th ed.). Wiley.

West, M. & Harrison

4 V-masks

4.1 Introduction to the method

Cumulative-sum (CUSUM) charts are a staple of statistical process control for detecting small, persistent shifts in the mean of a process. **V-masks** provide an intuitive, *retrospective* decision rule for CUSUM charts: a pre-defined V-shaped overlay is slid along the tail of the chart, and an alarm is raised whenever any part of the historic CUSUM path penetrates the V. Illustrations to this concept, as well as to the ones presented in the following theory, will be provided as outputs of the implementation script example.

When the monitored process is produced by a **Dynamic Linear Model (DLM)**, the standardized one-step-ahead forecast errors form the natural input to a CUSUM/V-mask scheme. This section reviews the underlying assumptions, presents the geometry and equations of the V-mask, and supplies reproducible R code for applying the method to DLM residuals.

4.2 The V-Mask Geometry

The two-sided V-mask is defined by two parameters:

- **Decision interval** $dist$ – controls the *vertical arm length* (distance from the vertex to each arm at the current time).
- **Reference value** $angle$ – controls the *slope* of each arm (in CUSUM units per time step).

Let C_t be the cumulative sum of standardized forecast errors up to time t,

$$C_t = \sum_{i=1}^{t} u_i\,, \qquad u_i = \frac{y_i - \hat{y}_{i|i-1}}{\hat{\sigma}},$$

where $u_i \sim \mathrm{N}(0,1)$ *if* the DLM is correctly specified.

With the **vertex**—the pivot point where the two arms meet—placed at (t_0, C_{t_0}), the upper and lower arm equations are

$$\begin{aligned} UCL_t &= C_{t_0} + dist + angle\,(t - t_0), \\ LCL_t &= C_{t_0} - dist - angle\,(t - t_0), \qquad t \geq t_0. \end{aligned}$$

Alarm rule:

Reject the in-control hypothesis at the first $t > t_0$ such that $C_t > UCL_t$ or $C_t < LCL_t$.

The choice $(dist, angle)$ is often linked to desired average run length (ARL) properties via approximations; a common heuristic sets $angle = 0.5\,\delta$ and $dist = 5\,\sigma$ for detecting a shift of size δ. Alternatively, the values can be optimized using a learning data set.

4.3 Assumptions

3. **Independence** – The standardized forecast errors u_t are assumed serially uncorrelated (white noise) when the system is in control.
4. **Normality** – $u_t \sim \mathrm{N}(0,1)$

5. **Parameter stability** – The DLM parameters used to compute $\hat{y}_{t|t-1}$ and $\hat{\sigma}$ are either known or re-estimated infrequently so as not to distort the control limits.
6. **Fixed sampling interval** – Observations arrive at equally spaced time points.

Violations of these assumptions will increase false-alarm rates. Using the (standardized) forecast errors from a well-made DLM ensures that these assumptions are met.

4.4 Relevant functions

The following is a function for calculating the CUSUM of the standardized forecast errors.

```
################################################################################
# Function to calculate the cumulative sum of forecast errors for univariate mo
dels.
# The forecast errors can, optionally, be standardized.
################################################################################
# x: A vector of values, for which the CUSUM will be calculated
################################################################################
runCusumUnivariate = function(x) {
  cusum = c()
  cusum[1] = 0
  for (i in 2:length(x)) {
    cusum[i] = sum(x[2:i])
  }
  return(cusum)
}
```

The function below is a direct implementation of a V-mask alarm detector frequently used in epidemiological surveillance. Each block is heavily commented to clarify its purpose.

```
###############################################################################
# runV.mask
# ---------
# Inputs:
#   cusums: result object returned by runCusumUnivariate(), containing cumulati
ve sums of forecast errors.
#   dist  : V-mask lead distance (horizontal offset of the vertex in time).
#   angle : slope of each arm of the V-mask (in CUSUM units per time step).
#   reset : logical; if TRUE, restart CUSUM at 0 after an alarm.
###############################################################################

runV.mask <- function(cusums, dist, angle, reset = TRUE) {
  ## Pre-allocate container ---------------------------------------------------
  n    <- length(cusums)
  mask <- matrix(FALSE, nrow = n, ncol = 6)  # will store metadata
  colnames(mask) <- c("t", "CUSUM", "alarmLow", "alarmHigh", "triggerIdx", "CUS
UMreset")

  mask[1, ] <- c(1, cusums[1], FALSE, FALSE, NA, cusums[1])
  lastAlarm <- 0             # index of most recent alarm (for reset logic)
  alarms    <- integer(0)    # vector to hold alarm time-steps
```

```
  for (i in 2:n) {
    ## Populate bookkeeping columns ----------------------------------------
    mask[i, 1] <- i
    mask[i, 2] <- cusums[i]

    alarmPrev <- mask[i - 1, 3] || mask[i - 1, 4]  # alarm at t-1?

    # Update reset CUSUM column (6) ----------------------------------------
    delta <- cusums[i] - cusums[i - 1]
    mask[i, 6] <- if (alarmPrev) delta else delta + mask[i - 1, 6]

    currentLevel <- if (reset) mask[i, 6] else cusums[i]

    ## Skip if we're immediately after a reset -----------------------------
    if (!reset || !alarmPrev) {
      for (j in (i - 1):(lastAlarm + 1)) {
        lower <- currentLevel - (i - j + dist) * angle
        upper <- currentLevel + (i - j + dist) * angle
        prev  <- if (reset) mask[j, 6] else cusums[j]

        if (prev < lower) {
          mask[i, 3] <- TRUE     # lower-tail alarm
          mask[i, 5] <- j
          alarms <- c(alarms, i)
          if (reset) lastAlarm <- i
        }
        if (prev > upper) {
          mask[i, 4] <- TRUE     # upper-tail alarm
          mask[i, 5] <- j
          if (reset) lastAlarm <- i
        }
      }
    }
  }

  Pred <- apply(X = mask[,c(3,4)], MARGIN = 1, FUN = max)

  return(list('mask' = mask, 'Pred' = Pred))
}
```

When `reset = TRUE`, each alarm restarts the cumulative sum so that subsequent shifts can be detected independently. The returned `mask` matrix can be visualised directly (e.g., with `geom_tile`) or post-processed to extract alarm times.

4.5 Choosing realistic values for `dist` and `angle`

In a **standardised CUSUM** (forecast errors scaled to roughly N(0, 1)) the two V-mask parameters (lead distance and angle of arm) translate directly into *how steep* and *how far back* you look:

Parameter	Interpretation	Practical range
`angle`	Slope of each arm	0.20 – 0.60
`dist`	Lead distance – historic points the mask covers	4 – 15

4.5.1 Heuristic for `angle`

`angle` can be chosen from the size of shift (in σ) you want to detect quickly:

Target mean shift δ (σ)	Recommended `angle` (= k)
0.3 – 0.5 (very small)	0.15 – 0.25
0.5 – 1.0 (moderate)	**0.25 – 0.50**
1.0 – 1.5 (large)	0.50 – 0.75

Most routine surveillance use **`angle` ≈ 0.4–0.5**.

4.5.2 Heuristic for dist

A simple rule is `dist ≈ h / angle`, where `h` is the **vertical half-height**—the vertical distance (in CUSUM σ-units) from the V-mask's vertex to either arm at the pivot point; in other words, the full opening of the V at its apex is `2 h`. This keeps the geometry of the mask self-consistent across different slopes. If you adopt **h ≈ 4–5 σ**, typical pairs are:

`angle`	`h = 4` → `dist`	`h = 5` → `dist`
0.25	16	20
0.40	10	12–13
0.50	8	**10**
0.60	7	8–9

Shorter `dist` values (4–7) react faster but double the in-control false-alarm rate.

4.5.3 Ready-to-use parameter sets

Scenario	`angle`	`dist`
Very cautious – catch ≥ 0.5 σ	0.30	14
Balanced default	**0.45**	**10**
Aggressive – large shifts	0.60	8

4.5.4 Quick helper function

The simple rule for choosing the *dist* parameter based on the angle for a given *h* can be implemented as a simple function, as follows:

```
choose_dist <- function(angle, h = 5) round(h / angle)
choose_dist(0.45)  # returns 11

## [1] 11
```

4.6 Plotting V-masks and alarms

The next two helper functions support the visualization of alarms **and to retrieve all alarm time-steps in a single call**:

- **`addMaskToPlot()`** overlays a V-mask on an existing CUSUM plot for a single alarm point.
- **`runAndPlotV.mask()`** is an all-in-one wrapper that runs `runV.mask()`, draws the CUSUM series, and calls `addMaskToPlot()` for *every* alarm. Both functions now take the **CUSUM vector directly** *and* **return the exact time-steps where alarms occur**—so downstream analyses (e.g., computing run-lengths) don't need to parse the mask matrix manually.

```
##################################################################################
# addMaskToPlot
# -------------
# Overlay a V-mask for a single alarm point on an existing CUSUM plot.
#
# Parameters
#   mask  : matrix returned by runV.mask().
#   obs   : index (integer) at which the alarm is raised and the mask pivoted.
#   dist  : V-mask lead distance (horizontal).
#   angle : arm slope (σ-units per time step).
#   reset : was runV.mask() executed with reset = TRUE?
##################################################################################
addMaskToPlot <- function(mask, obs, dist, angle, reset) {
  size <- dist # min(20, obs - 1)          # how many points to draw backwards
  rows <- max(nrow(mask), obs + dist)      # ensure matrix is long enough
  arms <- matrix(NA_real_, nrow = rows, ncol = 4)  # store three arm levels + x
  base <- if (reset) mask[obs, 6] else mask[obs, 2]  # CUSUM level at vertex

  ## Build the three arms point-by-point (working backwards in time) --------
  for (i in (obs + dist):(obs + dist - size)) {
    arms[i, 1] <- base - (obs + dist - i) * angle   # lower arm
    if (i > obs - 1) arms[i, 2] <- base             # central arm
    arms[i, 3] <- base + (obs + dist - i) * angle   # upper arm
    arms[i, 4] <- i                                  # x-coordinate
  }

  ## Draw the arms -----------------------------------------------------------
  lines(arms[, 4], arms[, 1], lty = 2, col = "red", lwd = 1)
  lines(arms[, 4], arms[, 2], lty = 2, col = "red", lwd = 1)
  lines(arms[, 4], arms[, 3], lty = 2, col = "red", lwd = 1)
}

##################################################################################
# runAndPlotV.mask
# ----------------
# Complete wrapper: runV.mask() + plot CUSUM + overlay V-masks for every alarm.
#
# Parameters
#   cusums: numeric vector of CUSUM values (length n >= 2)
#   dist  : V-mask lead distance
```

```
#   angle : arm slope
#   ylims : two-element numeric vector giving y-axis limits for the plot
#   reset : logical; if TRUE the detector restarts after every alarm
#
# Returns
#   The mask matrix produced by runV.mask()
###############################################################################
runAndPlotV.mask <- function(cusums, dist, angle, reset = TRUE) {
  ## Run detector -----------------------------------------------------------
  Vmask.out   <- runV.mask(cusums, dist, angle, reset)  # list(mask, alarms)
  mask  <- Vmask.out$mask
  Pred <- Vmask.out$Pred

  ## Choose which CUSUM series to visualise --------------------------------
  y <- if (reset) mask[, 6] else mask[, 2]

  ## Baseline plot -----------------------------------------------------------
  plot(y, type = "b", col = "blue",
       xlab = "Time", ylab = "Cumulative sum of standardised errors",
       ylim = range(y))
  abline(h = 0)

  ## Overlay V-masks only at alarm obs ---------------------------------
  alarms <- which(Pred > 0)
  for (obs in alarms) addMaskToPlot(mask, obs, dist, angle, reset)

  return(alarms)
}
```

4.6.1 Applying the V-mask

Here we will once again look at the data set *DLM_preprocessed_data__NEW_WithoutActivity.RDS*, in which the variables “consumption_liters”, “visitswent”, “visits”, and “DrinkingSpeed” have been filtered using a multivariate DLM. The meaning and units of those variables have been previously described in the variable dictionary (section 2.2.).

Furthermore, we will source the script *Functions for monitoring and filtering.R*, containing codes for assessing the performances.

Lastly, we will source the script called *V-mask functions.R*.

```
res <- readRDS("DLM_preprocessed_data__NEW_WithoutActivity.RDS")
source('Functions for monitoring and filtering.R')
source('V-mask functions.R')
```

We can check the assumption of standard-normality for the standardized forecast errors of each of the DLM-filtered variables for the healthy calves.

```
# Define the names of the standardized forecast errors
ut.names <- c("ut_consumption_liters", "ut_visitswent", "ut_visits", "ut_Drinki
ngSpeed")
```

```
# Identify the healthy calves
agg <- aggregate(x = res$SickOrHealthy, by=list(res$calf.herd), FUN=max)
colnames(agg) <- c('calf.herd', 'sick')
healthy.calves <- agg$calf.herd[which(agg$sick == 0)]
healthy.set <- subset(res, res$calf.herd %in% healthy.calves)
sick.calves <- agg$calf.herd[which(agg$sick == 1)]
sick.set <- subset(res, res$calf.herd %in% sick.calves)

# Check the overall distribution of the standardized forecast errors for each v
ariable
for(name in ut.names){
  Mean <- round(mean(healthy.set[,name]),2)
  SD <- round(sd(healthy.set[,name]),2)
  print(paste(name, Mean, SD))
}

## [1] "ut_consumption_liters -0.02 1.03"
## [1] "ut_visitswent 0 0.96"
## [1] "ut_visits 0.05 1.01"
## [1] "ut_DrinkingSpeed 0.01 1.04"
```

We see that the standardized errors deviate to varying degrees from a proper standard normal distribution (i.e., a mean of 0 and a standard deviation of 1), but they are all pretty close.

4.6.1.1 *Illustrative example*

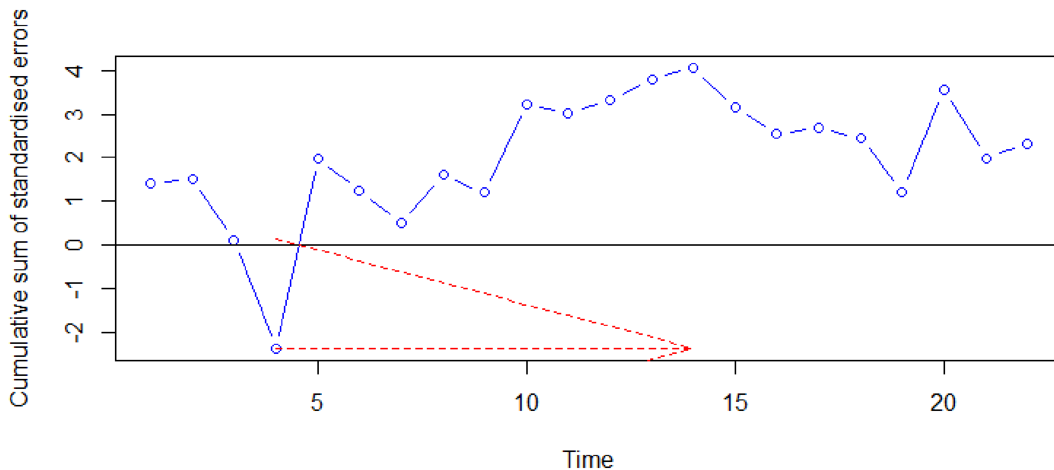
Initially, we can randomly select a sick calf to apply the V-mask for illustrative purposes. For this illustration, we will only apply the function to the *ut_consumption_litres* column.

The *runAndPlotV.mask* function returns a vector of the time steps for which an alarm was raised.

Varying the values for *dist*, *angle*, and *reset* will affect the number and positions of alarms being raised.

Notice that an alarm for a given observation can only be raised *dist* time steps **after** the observation was made, meaning that the leading distance will define the time lag between the abnormal observation and its alarm, as that is the distance that the mask vertix will be placed from the observation.

```
# Select a random sick calf
set.seed(42)
calf <- sample(x = sick.calves, size = 1)
calf.set <- subset(res, res$calf.herd == calf)

# Run the V-mask and plot it
cusums = calf.set$ut_consumption_liters
alarms <- runAndPlotV.mask(cusums =  cusums,
                 dist = 10,
                 angle = 0.25,
                 reset = TRUE)
```




```
# Look at the alarms which were raised
print(alarms)

## [1] 4
```

4.6.1.2 Performance assessment

We can now apply the V-mask to the standardized forecast errors of all DLM-variables for all calves and assess the performance of the method. We will systematically test the performance using different angles and the corresponding distances

For this example, we will assume that the standardized forecast errors produced by the DLM are good to go as they are.

We will iteratively try different values for the *angle* and set the corresponding *dist* using the predefined helper function. We will reset the cusums to 0 when an alarm is raised, so it doesn't keep being repeatedly raised for the following observations.

Since we are applying the V-mask multiple times to each calf, we will do this without plotting. This is done simply by using the *runV.mask* and saving the *Pred* object

```
# Make output data frames
performance.all <- data.frame()

# Define the angles to use
angles <- seq(from=0.10, to=0.90, by=0.05)

# Iterate over the ut names
for(name in ut.names){

  # Iterate over the angles
  for(angle in angles){
    dist <- choose_dist(angle)

    # Iterate over the calves
```

```
    Pred.all <- c()
    for(calf in unique(res$calf.herd)){
      calf.set <- subset(res, res$calf.herd == calf)
      # Apply the V-mask to this column of standardized forecast errors using t
hese settings
      Vmask.out <- runV.mask(cusums = calf.set[,name],
                  dist = choose_dist(angle),
                  angle = angle,
                  reset = TRUE)
      Pred <- Vmask.out$Pred
      Pred.all <- c(Pred.all, Pred)
    }



    )
  }
}
```

We can now look at the performances.

```
# Get the best performance
best.i.naive <- which(performance.all$MMA == max(performance.all$MMA))
print(performance.all[best.i.naive,])

##                  Name Angle TP FP  TN   FN Sensitivity Specificity  MMA
## 35          ut_visits  0.10 56  7 843 1866        0.03        0.99 0.51
## 36          ut_visits  0.15 53  6 844 1869        0.03        0.99 0.51
## 52 ut_DrinkingSpeed  0.10 66  9 841 1856        0.03        0.99 0.51
## 53 ut_DrinkingSpeed  0.15 65  9 841 1857        0.03        0.99 0.51
## 54 ut_DrinkingSpeed  0.20 61  7 843 1861        0.03        0.99 0.51
## 55 ut_DrinkingSpeed  0.25 55  7 843 1867        0.03        0.99 0.51
## 56 ut_DrinkingSpeed  0.30 50  6 844 1872        0.03        0.99 0.51
## 57 ut_DrinkingSpeed  0.35 50  6 844 1872        0.03        0.99 0.51
## 58 ut_DrinkingSpeed  0.40 52  6 844 1870        0.03        0.99 0.51
## 63 ut_DrinkingSpeed  0.65 40  4 846 1882        0.02           1 0.51
## 64 ut_DrinkingSpeed  0.70 42  4 846 1880        0.02           1 0.51
## 65 ut_DrinkingSpeed  0.75 36  3 847 1886        0.02           1 0.51
## 66 ut_DrinkingSpeed  0.80 41  4 846 1881        0.02           1 0.51
## 67 ut_DrinkingSpeed  0.85 37  3 847 1885        0.02           1 0.51
## 68 ut_DrinkingSpeed  0.90 35  2 848 1887        0.02           1 0.51
```

We see that 15 different combinations of settings and name results in the highest performance of MMA = 51 %.

4.7 Conclusion about the method

V-masks offer a transparent, visually appealing decision tool for CUSUM charts derived from DLM innovations. While the classical design assumes independent, Gaussian errors, the method extends naturally to the

state-space context once residual diagnostics validate these assumptions or suitable bootstrap calibration is applied.

Notice that an alarm for a given observation can only be raised *dist* time steps **after** the observation was made. Thus, unlike the other methods, the alarms of the v-mask are raised retrospectively.

4.8 Further reading

Basseville & Nikiforov (1993). *Detection of Abrupt Changes*.

Montgomery (2009). *Introduction to Statistical Quality Control*, 8th ed. West & Harrison (1997). *Bayesian Forecasting and Dynamic Models*, 2nd ed.

5 Tabular cusums

5.1 Introduction to the method

Cumulative sum (CUSUM) control charts are classical tools for rapid detection of small, sustained shifts in process level. When the *in-control* distribution of the monitored statistic is Normal $\mathcal{N}(\mu_0, \sigma^2)$, the **tabular CUSUM** (also called "one-sided CUSUM" or "sequential probability-ratio CUSUM") offers an intuitive, recursive formulation that is straightforward to implement in R.

Dynamic Linear Models (DLMs) provide one-step-ahead forecasts and forecast errors (residuals) that—under correctly specified models—are uncorrelated, mean-zero and normally distributed. Monitoring those forecast errors with a CUSUM chart is therefore an effective way to raise early alarms for structural change in the underlying time series.

In this section we review the tabular CUSUM, list its assumptions, and demonstrate an R implementation.

5.2 Theory

Let u_t be the standardized forecast error at time t. Define the slack parameter k (often set to half the desired detectable mean shift $\delta\sigma/2$). The upper and lower tabular CUSUM statistics are updated as:

$$C_t^+ = \max(0,\, C_{t-1}^+ + u_t - k), \quad t \geq 1.$$

$$C_t^- = \max(0,\, C_{t-1}^- - u_t - k), \quad t \geq 1.$$

An *alarm* is triggered whenever either statistic exceeds its decision limit h:

$$\text{Signal if} \quad C_t^+ \geq h \text{ or } C_t^- \geq h.$$

The decision limit h is chosen to achieve a target in-control Average Run Length (ARL_0), i.e. the average number of observations between two consecutive alarms when the system is in control (false alarms). This is typically done via simulation or standard tables (e.g. Lucas & Crosier 1982).

The following are the assumptions of the method:

- **Independence & Normality.** Standardized forecast errors u_t are assumed i.i.d. $\mathcal{N}(0, \sigma^2)$ *when the process is in control*.
- **Known (or consistently estimated) $\boldsymbol{\sigma}$** The slack parameter k and decision limit h depend on the standard deviation of the errors.
- **Correct model specification.** Violations such as autocorrelated residuals will inflate the false-alarm rate.

6 R Implementation

Below is the function *runTabularCusumUnivariate*. It returns the running upper and lower CUSUM paths.

```
################################################################################
# Function calculating the (upper and lower) tabular cusum for forecast errors
################################################################################
# ut: a vector of standardized forecast errors produced by the DLM
# slack:  The slack value (k)
# decision: the decision limit (h); when exceeded by the tabular cusum, an alar
m is raised
################################################################################
# Returns a list with two vectors of cusums (upper and lower)
################################################################################
runTabularCusumUnivariate = function(ut, slack, decision) {
  # Calculate the tabular cusums
  upperC <- numeric(length(ut))
  lowerC <- numeric(length(ut))
  upperC[1] <- 0
  lowerC[1] <- 0
  for (i in 2:length(ut)) {
    upperC[i] <- max(0,  ut[i] - slack + upperC[i-1])
    lowerC[i] <- max(0, -ut[i] - slack + lowerC[i-1])
  }
  # Apply the decision limit to automatically raise alarms
  upperAlarms <- rep(0, length(ut))
  lowerAlarms <- rep(0, length(ut))
  anyAlarms <- rep(0, length(ut))

  upperAlarms[which(upperC > decision)] <- 1
  lowerAlarms[which(lowerC < -decision)] <- 1
  anyAlarms[which(upperAlarms == 1 | lowerAlarms == 1)] <- 1


  return(list(upper = upperC,
              lower = lowerC,
              upperAlarms = upperAlarms,
              lowerAlarms = lowerAlarms,
              anyAlarms = anyAlarms))
}
```

The following function, *createTabularCusumPlot* wraps the *runTabularCusumUnivariate* function and creates a plot of the tabular cusums.

```
################################################################################
# Function creating a cusum plot with decision lines
################################################################################
# ut: a vector of standardized forecast errors produced by the DLM
# slack: The slack value (k)
# decision: The decision level (h)
################################################################################
```

```
createTabularCusumPlot = function(ut, slack, decision) {
  tabs = runTabularCusumUnivariate(ut, slack, decision)
  amp = max(abs(tabs$upper), abs(tabs$lower))
  plot(tabs$upper, type ='h', ylim = c(min(-amp, -decision), max(amp, decision)
), ylab='Tabular cusum')
  lines(-tabs$lower, type='h')
  abline(h=decision, col = "red")
  abline(h=-decision, col = "red")
  abline(h = 0)
}
```

6.1 Choosing *k* and *h*

For a Normal in-control process with known σ the slack k is often fixed at $k = \delta\sigma/2$, where δ is the shift size (in σ units) that should be detected promptly. The advantage of the tabular cusum over the standard Shewart control chart is in detecting small level shifts quickly, and so the shift size that should be detected with this method is usually relatively small, e.g. 1 σ unit.

The decision limit h can be determined from Montgomery (2009) tables, Lucas & Crosier (1982), or by Monte-Carlo simulation to hit a target ARL_0. According to Montgomery (2009), **“a reasonable value for h is five times the process standard deviation, s”**.

If the process follows a standard normal distribution and we desire to detect a level shift of 1 σ, and we use 5 as our value for h, then the average run length between false alarms will be roughly 370 observations, as shown with the *xcusum.arl* below.

```
## Example Monte-Carlo ARL estimation
# install.packages("spc")   # first-time only
library(spc)

## Warning: pakke 'spc' blev bygget under R version 4.3.3

arl <- xcusum.arl(k = 0.5, h = 5, mu = 0, sided = "two") # may take some time t
o run
arl

## [1] 465.4435
```

In Table 9.4 of Montgomery (2009), it is recommended to use the following combinations of slack k and decision limits h to achieve an average run length between false alarms (ARL_0) of 370 observations:

slack k	decision limit h
0.25	8.01
0.50	4.77
0.75	3.34
1.00	2.52
1.25	1.99
1.50	1.61

Notice that as the slack value increases, the decision limit value decreases.

6.2 Practical Tips

- **Scaling:** If σ is unknown, standardise residuals before computing the CUSUM.
- **Autocorrelation:** Check the ACF of u_t. Significant lags suggest model misspecification—adjust the DLM or widen h to control the false-alarm rate.

6.2.1 Applying the tabular cusum

Here we will once again look at the data set *DLM_preprocessed_data__NEW_WithoutActivity.RDS*, in which the variables “consumption_liters”, “visitswent”, “visits”, and “DrinkingSpeed” have been filtered using a multivariate DLM.

Furthermore, we will source the script *Functions for monitoring and filtering.R*, containing codes for assessing the performances.

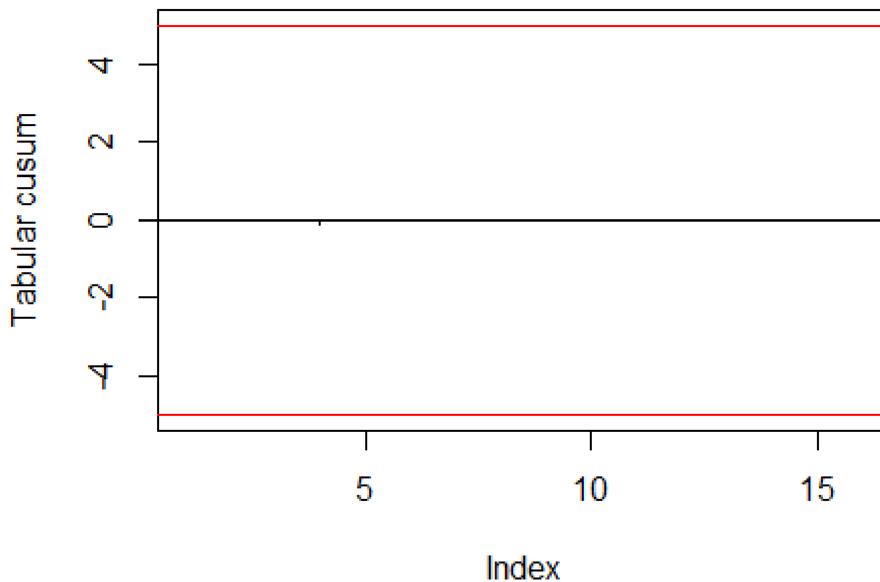
```
res <- readRDS("DLM_preprocessed_data__NEW_WithoutActivity.RDS")
source('Functions for monitoring and filtering.R')
```

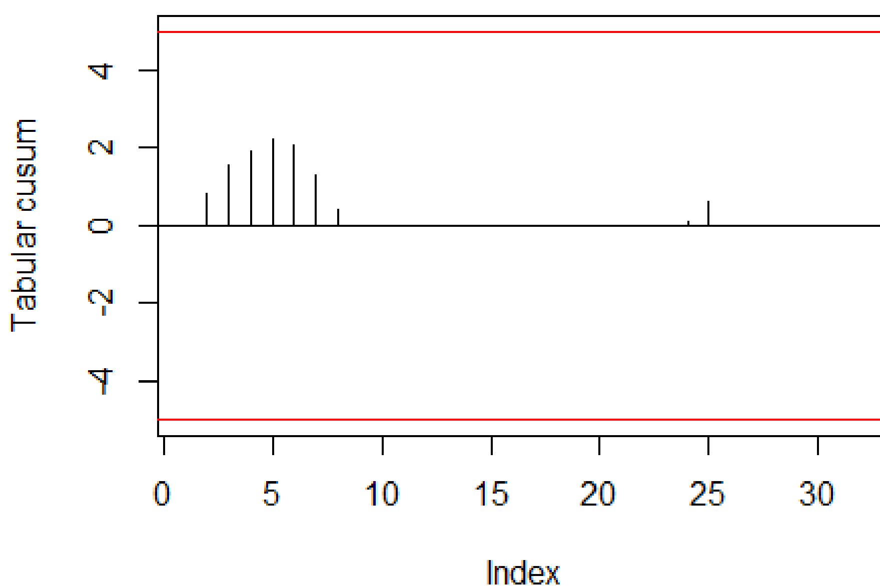
6.2.1.1 Illustrative example

Initially, we can randomly select a healthy and a sick calf to apply the tabular cusum method to for illustrative purposes. For this illustration, we will only apply the function to the *ut_consumption_litres* column.

In theory, the standardized forecast errors should follow a standard normal distribution with a mean of 0 and a standard deviation of 1. Therefore, the slack k is fixed at 0.5 based on $k = \delta\sigma/2$. For these illustrations we will set to 1 sigma unit and use a decision limit h of 5, as recommended by Montgomery (2009).

```
# Select a random healthy calf
set.seed(42)
healthy.calves <- unique(res$calf.herd[which(res$AnySickness == 0)])
healthy.calves <- sample(x = healthy.calves, size = 1)

# Select a random sick calf
set.seed(42)
sick.calves <- unique(res$calf.herd[which(res$AnySickness == 1)])
sick.calves <- sample(x = sick.calves, size = 1)

# Plot the selected calves
calves <- c(healthy.calves, sick.calves)
slack <- 0.5
decision = 5
for(calf in calves){
    calf.set <- subset(res, res$calf.herd == calf)
    createTabularCusumPlot(ut = calf.set[,'ut_consumption_liters'],
                           slack = slack,
                           decision = decision)
}
```





Notice that no alarms were raised for either the healthy or the sick calf using these settings, as the cusums (vertical black lines) never extend beyond the decision limits (horizontal red lined).

6.2.1.2 Performance assessment - using Montgomery recommendations

We can now apply the tabular cusum method to the standardized forecast errors of all DLM-variables for all calves and assess the performance of the method. We will systematically test the performance using values for slack k from 0.25 to 1.5 by steps of 0.25, along with the corresponding values of h recommended by Montgomery 2020 to achieve an ARL_0 of 370 observations.

Since we are applying the method multiple times to each calf, we will do this without plotting. This is done simply by using the *runTabularCusumUnivariate* and saving the output as a data frame. We can then check

the performance using the alarms for the lower limit, upper limit, and any alarms. The standards for performance assessment are the same as previously explained when presenting MMAs for Shewhart Control Charts.

```
# Define the names of the standardized forecast errors
ut.names <- c("ut_consumption_liters", "ut_visitswent", "ut_visits", "ut_Drinki
ngSpeed")

# Make output data frames
performance.all <- data.frame()
performance.all.aggregated <- data.frame()

# Define the k and corresponding h values to use
ks <- c(0.25, 0.50, 0.75, 1.00, 1.25, 1.5)
hs <- c(8.01, 4.77, 3.34, 2.52, 1.99, 1.61)

# Iterate over the ut names
for(name in ut.names){

  # Iterate over the k and corresponding h values
  for(k in ks){
    h <- hs[which(ks == k)]

    # Iterate over the calves, and save the output along with the input
    res_new <- data.frame()
    for(calf in unique(res$calf.herd)){
      calf.set <- subset(res, res$calf.herd == calf)
      # Apply the V-mask to this column of standardized forecast errors using t
hese settings
      TabularCusum.out <- runTabularCusumUnivariate(ut = calf.set[,name], slack
= k, decision = h)
      # Turn the output list into a data frame
      TabularCusum.out <- as.data.frame(TabularCusum.out)
      # cbind the output on to the calf.set data frame
      calf.set <- cbind(calf.set, TabularCusum.out)
      # Add this to the new res_new data frame
      res_new <- rbind(res_new, calf.set)
    }

    # For each type of alarm, assess the performance
    for(alarmType in c("upperAlarms", "lowerAlarms", "anyAlarms" )){
      Pred.all <- res_new[,alarmType]

      # Assess the naive performance on this ut-name with these settings
      performance <- getPerformance(observations = res$SickOrHealthy, alarms =
Pred.all)
      performance <- cbind('Name'=name, 'k'=k, 'h'=h, 'alarmType'=alarmType, pe
rformance)
      performance.all <- rbind(performance.all, performance)
```

```
    }


  }
}

# Remove columns with TP, FP, TN, and FN to make the tables better fit the page
performance.all <- performance.all[,-c(5,6,7,8)]
performance.all.aggregated <- performance.all.aggregated[,-c(5,6,7,8)]
```

We can now look at the set of parameters resulting in the best performance.

```
best.i <- which(performance.all$MMA == max(performance.all$MMA))
print(performance.all[best.i,])

##                Name    k    h   alarmType Sensitivity Specificity  MMA
## 61 ut_DrinkingSpeed 0.75 3.34 upperAlarms        0.07        0.97 0.52
## 63 ut_DrinkingSpeed 0.75 3.34   anyAlarms        0.07        0.97 0.52
## 64 ut_DrinkingSpeed 1.00 2.52 upperAlarms        0.06        0.98 0.52
## 66 ut_DrinkingSpeed 1.00 2.52   anyAlarms        0.06        0.98 0.52
##    MMA 95 % CI
## 61 0.513-0.527
## 63 0.513-0.527
## 64 0.513-0.527
## 66 0.513-0.527
```

6.3 Conclusions about the method

Tabular CUSUMs provide a lightweight yet powerful complement to state-space monitoring of DLMs. Implemented in a handful of lines in R, they add a clearly interpretable alarm layer that safeguards real-time forecasting pipelines against unnoticed structural change.

It can be an advantage to define your slack and decision limit values to expect an ARL_0 which corresponds to the expected length of your time series.

6.4 References

- Lucas, J.M., & Crosier, R.B. (1982). Fast Initial Response for CUSUM Quality-Control Schemes. *Technometrics,* 24(3), 199–205.
- Montgomery, D.C. (2020). *Introduction to Statistical Quality Control* (9th ed.). Wiley.
- West, M., & Harrison, J. (1997). *Bayesian Forecasting and Dynamic Models* (2nd ed.). Springer.

7 Cholesky decomposition and the Malahanobis distance

7.1 Introduction to the method

Dynamic Linear Models (DLMs) supply, at every time point t, a vector of one–step-ahead forecast errors $e_t = y_t - \hat{y}_t$ and its $p \times p$ forecast-error covariance matrix Q_t. Because in multivariate models the elements of e_t are **correlated** and can have different units, a threshold that makes sense for one of the variables variable s not sufficient for online monitoring. The method summarised here transforms the error vector to a single scalar that, under the model assumptions, follows a χ^2 distribution. Comparing that statistic with a pre-chosen quantile yields an intuitive "green / red" decision rule.

7.2 Method details

7.2.1 Forecast error distribution

Assume the DLM is correctly specified so that

$$e_t \sim \mathcal{N}(\mathbf{0}, Q_t), \qquad Q_t \succ 0.$$

7.2.2 Whitening transformation

Every positive-definite matrix admits a *Cholesky factorisation*

$$Q_t = R_t^{\top} R_t,$$

where R_t is upper-triangular. Multiplying the error vector by R_t^{-1} **decorrelates and rescales** it:

$$u_t = R_t^{-1} e_t \sim \mathcal{N}(\mathbf{0}, I_p).$$

The following figure illustrates this for whitening transformation for the correlation between the forecast errors for the variables *visitwent* and *DrinkingSpeed*. Before the Cholesky decomposition, the two variables are mutually co-dependent, as seen by the elliptical distribution, while after Cholesky decomposition the two variables are mutually independent, as seen by the circular distribution.

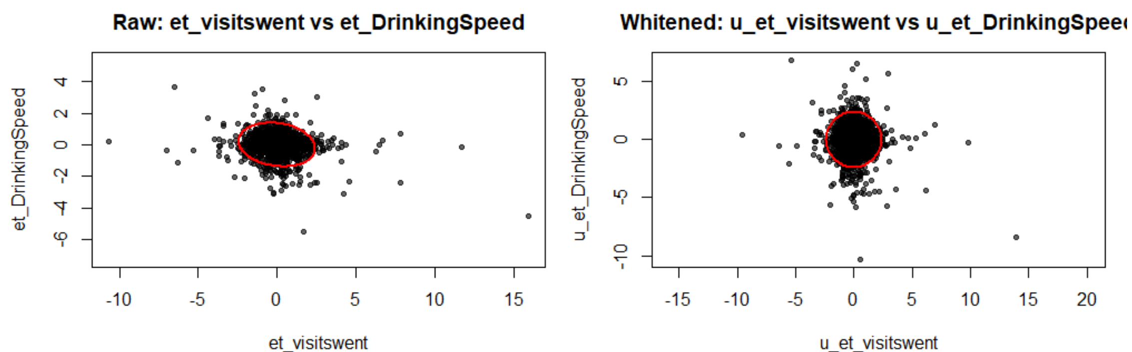


Illustration of the whitening transformation for the correlation between the forecast errors for the variables visitwent and DrinkingSpeed

7.2.3 Mahalanobis distance

Because u_t has identity covariance, its squared Euclidean norm

$$d_t^2 = u_t^\top u_t = e_t^\top Q_t^{-1} e_t$$

obeys the *Chi-square* law

$$d_t^2 \ \sim \ \chi_p^2.$$

A one-sided test at confidence level $1 - \alpha$ therefore flags an alarm whenever

$$d_t^2 > q_{\chi_p^2}(1 - \alpha),$$

with $q_{\chi_p^2}$ the upper quantile function.

7.2.4 Optional dashboard scaling

If several series with different dimensions p are displayed side-by-side, it is convenient to rescale d_t^2 so that **all** of them share the *same* control limit corresponding to the largest dimension $p_{\max}$:

$$\tilde{d}_t^2 = d_t^2 \frac{q_{\chi_{p_{\max}}^2}(1 - \alpha)}{q_{\chi_p^2}(1 - \alpha)}.$$

The statistic $\tilde{d}_t^2$ is returned by the R helper function `get.MahalanobisDistance()` described next.

7.3 R implementation

7.3.1 Convenience function for monitoring

The following helper, `get.MahalanobisDistance()`, bundles the steps from the *Method* section—symmetrising the covariance matrix, whitening the error vector, computing the Mahalanobis distance, rescaling to a common χ² control limit, and extracting an upper-tail probability.

```
#' Calculate Mahalanobis distance and χ² p-value for DLM forecast errors
#'
#' @param et Numeric vector of forecast errors (length p).
#' @param Qt Forecast-error covariance matrix (p × p, positive-definite).
#' @param p_max Integer giving the maximum dimension to which distances
#'        should be rescaled (defaults to length(et)).
#' @param alpha Significance level for the control limit (defaults to 0.05).
#'
#' @return A list with components:
#'   * d2          - Mahalanobis distance (squared)
#'   * d2_scaled   - Distance rescaled to `p_max` degrees of freedom
#'   * p_value     - Upper-tail χ² probability (`df = p_max`)
#'   * ucl         - Control limit `qchisq(1-alpha, df = p_max)`
#' @examples
#' res <- get.MahalanobisDistance(et, Qt)
#' res$p_value < 0.05  # alarm?
#' @export
get.MahalanobisDistance <- function(et, Qt, p_max = length(et), alpha = 0.05) {
  stopifnot(is.numeric(et), is.matrix(Qt),
            length(et) == nrow(Qt), nrow(Qt) == ncol(Qt))

  # 1. Force exact symmetry
```

```
  Qt <- (Qt + t(Qt)) / 2

  # 2. Upper-triangular Cholesky factor, Qt = Rᵀ R
  R <- chol(Qt)

  # 3. Whitening: u = R⁻¹ e (efficient back-substitution)
  u <- backsolve(R, et, transpose = TRUE)

  # 4. Mahalanobis distance
  d2 <- sum(u ^ 2)

  # 5. Rescale for common dashboard limit
  d2_scaled <- d2 * qchisq(1 - alpha, df = p_max) /
                     qchisq(1 - alpha, df = length(et))

  # 6. Tail probability
  p_val <- pchisq(d2_scaled, df = p_max, lower.tail = FALSE)

  list(
    d2          = d2,
    d2_scaled = d2_scaled,
    p_value   = p_val,
    ucl         = qchisq(1 - alpha, df = p_max)
  )
}
```

7.3.2 Batch processing across time

The *runDLM* function can be adjusted to apply the Mahalanobis distance calculations at each time step, so we can obtain multivariate-level forecast errors at every step, to be used in place of the several et columns generated. To do this, we need to call the *get.MahalanobisDistance* function at each time step, as well as adding new storage lists for the four outputs of the *get.MahalanobisDistance* function.

The following code chunk shows how to update the *runDLM* function; the comments *# <– THIS IS NEW!* shows where lines have been added.

```
## A function for running the DLM
runDLM <- function(Data,
                   mu0,
                   C0,
                   V,
                   W=NA,
                   adjust.W=FALSE,
                   delta=0.95,
                   relevant.names,
                   Spline.list=NA,
                   time.var,
                   stratify.by=NA){

  n <- nrow(Data)
```

```
  Yt.list <- list()
  at.list <- list()      # Define the lists
  Rt.list <- list()
  ft.list <- list()
  Qt.list <- list()
  At.list <- list()
  et.list <- list()
  ut.list <- list()
  mt.list <- list()
  Ct.list <- list()
  Ft.list <- list()
  VSE.list <- list()
  Gt.list <- list()
  d2.list <- list()  #  <-- THIS IS NEW!
  d2_scaled.list <- list()  #  <-- THIS IS NEW!
  p_value.list <- list()  #  <-- THIS IS NEW!
  ucl.list <- list()  #  <-- THIS IS NEW!

  mt <- mu0             # Prior Distribution
  Ct <- C0
  # Make sure Ct is symmetrical
  Ct <- (Ct + t(Ct))/2
  Ct <<- Ct

  #When we have an multivariate DLM with trend, both mu0 and C0 are not numbers
, but matrices. Thereby, remember to apply %*% in all cases.

  for(i in (1:n)){

    # Define relevant variables as global variables
    # - it's not pretty, but it makes it easier to use custom get.Gt and get.Ft
functions
    Data_ <<- Data
    i <<- i
    time.var <<- time.var
    stratify.by <<- stratify.by
    Spline.list <<- Spline.list
    relevant.names <<- relevant.names

    # Get V-sum-element, to be used in the EM-algorithm
    VSE <- getVSumElement(Data, i)

    # Get the observation vector
    # Yt <- getYt(Data, i, relevant.names)
    Yt <- t(as.matrix(Data[i,relevant.names]))
    colnames(Yt) <- NULL

    # Get the observational variance (Vt), including only values related to obs
erved variables
    # Vt <- getVt(Data, i, V, relevant.names)
    Vt <- V
```

```
    Vt <<- Vt

    # Define Wt
    if(identical(W, NA)){
      Wt <- ((1-delta)/delta) * Ct
      Wt <- (Wt + t(Wt))/2
    }else{
      Wt <- W
    }
    Wt <<- Wt

    # Make the DLM more adaptive in the beginning
    if(adjust.W == TRUE & i < 5){
      Wt <- Wt * 20000
    }

    # Get Gt - independent of the current Yt
    Gt <- get.Gt()
    Gt <<- Gt

    # Get Ft given the current Yt
    # - only rows related to observed variables are included
    Ft <- get.Ft()
    Ft <<- Ft

    # Run the Kalman filter - only if we observe at least one variable!
    mt <<- mt
    at <- Gt %*%  mt                           # Prior mean
    Rt <- Gt %*%  Ct %*% t(Gt) + Wt        #! I have changes it to G'     # Pri
or Variance
    Rt <<- Rt

    ft <- t(Ft) %*% at                       # One-step Forecast mean
    Qt <- t(Ft) %*% Rt %*%  Ft + Vt        # One-step Forecast variance

    At <- Rt %*% Ft %*% solve(Qt)          # Adaptative Coef. matrix
    et <- Yt  - ft                         # one-step forecast error
    ut <- et / sqrt(diag(Qt))              #Standardized forecast error

    # - handle the missing values
    et.A <- et
    et.A[which(is.na(et.A))] <- 0

    # - update the parameter vector and variance matrix
    mt <- at + At %*% et.A                # Filtered mean
    Ct <- Rt - At  %*% Qt %*% t(At)       # Filtered variance

    # Make sure Ct is symmetrical
    Ct <- (Ct + t(Ct))/2
    Ct <<- Ct
```

```
    # Calculate the Mahalanobis distance <-- THIS IS NEW!
    Mahalanobis.out <- get.MahalanobisDistance(et, Qt, p_max = length(et), alph
a = 0.05)
    d2 <- Mahalanobis.out$d2
    d2_scaled <- Mahalanobis.out$d2_scaled
    p_value <- Mahalanobis.out$p_value
    ucl <- Mahalanobis.out$ucl

    # Save the values in lists
    Yt.list[[i]] <- Yt
    at.list[[i]] <- at
    Rt.list[[i]] <- Rt
    ft.list[[i]] <- ft
    Qt.list[[i]] <- Qt
    At.list[[i]] <- At
    et.list[[i]] <- et
    ut.list[[i]] <- ut
    mt.list[[i]] <- mt
    Ct.list[[i]] <- Ct
    Ft.list[[i]] <- t(Ft)
    VSE.list[[i]] <- VSE
    Gt.list[[i]] <- Gt
    d2.list[[i]] <- d2 #  <-- THIS IS NEW!
    d2_scaled.list[[i]] <- d2_scaled  #  <-- THIS IS NEW!
    p_value.list[[i]] <- p_value  #  <-- THIS IS NEW!
    ucl.list[[i]] <- ucl  #  <-- THIS IS NEW!

  }


  return(list(
    Yt=Yt.list,
    at=at.list,
    Rt=Rt.list,
    ft=ft.list,
    Qt=Qt.list,
    At=At.list,
    et=et.list,
    ut=ut.list,
    mt=mt.list,
    Ct=Ct.list,
    F=Ft.list,
    vse=VSE.list,
    Gt.list=Gt.list,
    d2.list,  #  <-- THIS IS NEW!
    d2_scaled,  #  <-- THIS IS NEW!
    p_value <- p_value,  #  <-- THIS IS NEW!
    ucl[[i]] <- ucl  #  <-- THIS IS NEW!
  ))
}
```

7.4 Illustrative Example

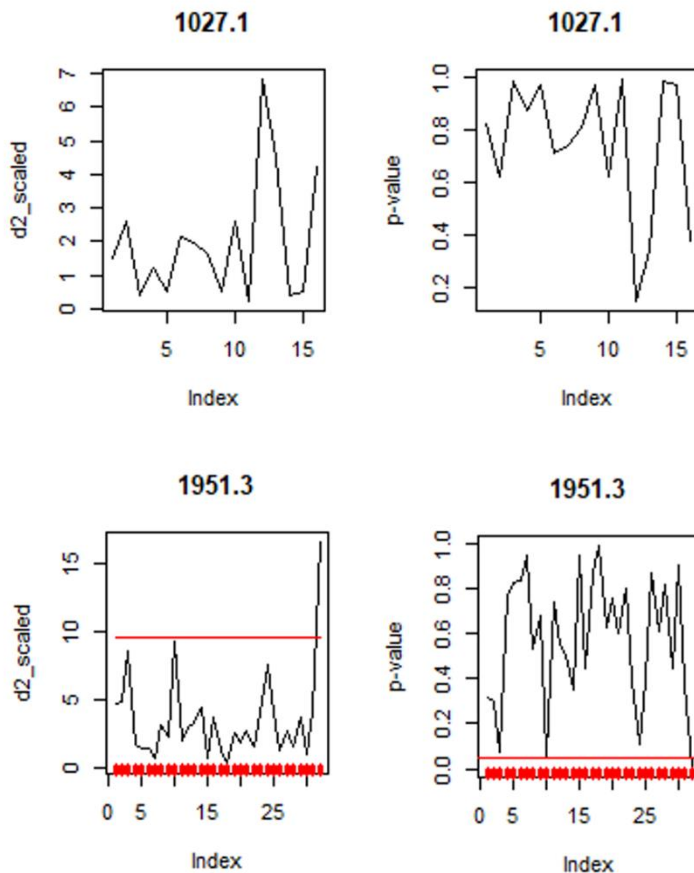
Here we will look at the data set *DLM_preprocessed_data__NEW_WithoutActivity.RDS*, in wich the variables “consumption_liters”, “visitswent”, “visits”, and “DrinkingSpeed” have been filtered using a multivariate DLM.

Furthermore, we will source the script *DLM functions - DECIDE DLM summer school.R*.

```
res <- readRDS("DLM_preprocessed_data__NEW_WithoutActivity.RDS")
source('DLM functions - DECIDE DLM summer school.R')
```

We can randomly select a healthy and a sick calf and look at the outputs from the Mahalnanobis function.

```
# Select a random healthy calf
set.seed(42)
healthy.calves <- unique(res$calf.herd[which(res$AnySickness == 0)])
healthy.calves <- sample(x = healthy.calves, size = 1)

# Select a random sick calf
set.seed(42)
sick.calves <- unique(res$calf.herd[which(res$AnySickness == 1)])
sick.calves <- sample(x = sick.calves, size = 1)

# Plot the selected calves
calves <- c(healthy.calves, sick.calves)

par(mfrow=c(2,3))
for(calf in calves){
  calf.set <- subset(res, res$calf.herd == calf)
  plot(calf.set$d2_scaled, type='l', main=calf, ylab='d2_scaled')
  lines(calf.set$ucl, col='red')
  rug(which(calf.set$AnySickness == 1), col='red', lwd=3)

  plot(calf.set$p_value, type='l', main=calf, ylab='p-value')
  abline(h=0.05, col='red')
  rug(which(calf.set$AnySickness == 1), col='red', lwd=3)
}
```



The first two plots show the healthy calf (1027.1), while other two show the sick calf (1951.1). The horizontal red lines represent the alarm threshold for each metric.

Notice that the upper control limit, *ucl*, which is produced as an output from the *get.MahalanobisDistance* function only applies to *d2_scale*, not *d2*. That being said, so long as none of the variables which the DLM was applied to are missing, the values of *d2* and *d2_scaled* will be identical.

7.4.1 Performance assesment

We can now assess the performance of using the outputs from the *get.MahalanobisDistance* function to raise alarms, in terms of detecting sick calves. For this purpose, we will need to source the scirpt *FUNCTION COLLECTION - RANDOM FORESTS.R*, which contains functions for performance assesment.

```
source('FUNCTION COLLECTION - RANDOM FORESTS.R')
```

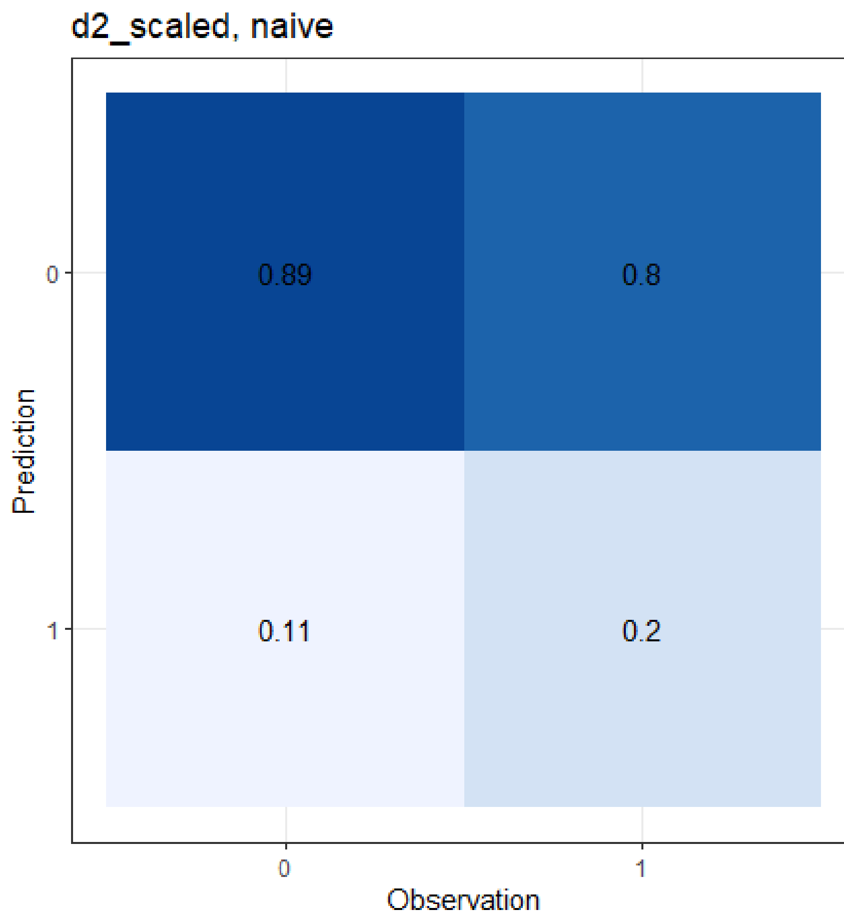
7.4.1.1 d2_scaled

First, we will consider the *d2_scaled* values. We can add a column called "Pred" (predicted values by the DLM?) to the *res* data frame and set *Pred* to 1 if *d2_scaled* is at or above the upper control limit, and 0 otherwise.

```
res$Pred <- 0
res$Pred[which(res$d2_scaled >= res$ucl)] <- 1
```

Now, we can make a data frame called "obs.n.pred", where "Obs" is set equal to the value for variables *res$AnySickness**, and "Pred" is set equal to the value in variable *res$Pred.* These specific column names are needed for the confusion.matrix* function to work. We can now call the *confusion.matrix* function.

```
obs.n.pred <- as.data.frame(cbind('Obs'=res$AnySickness, 'Pred'=res$Pred))
confusion.matrix(obs.n.pred, Title="d2_scaled, naive")
```

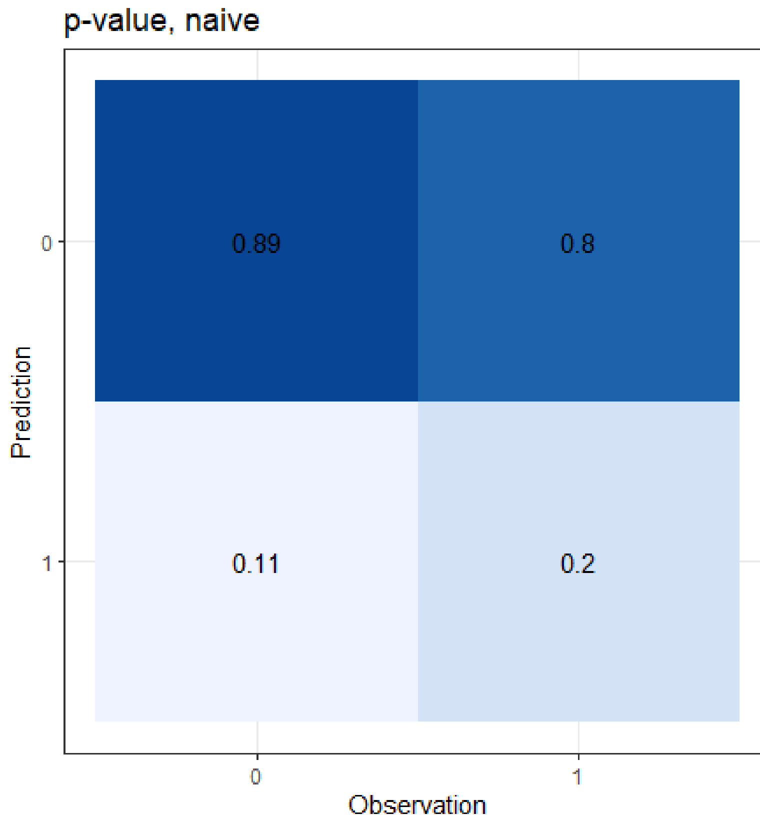


We see that the specificity is 89 % and the sensitivity is 20 %.

7.4.1.2 p-value

We now repeat the same steps for the *p-value* values, except that here an alarm is raised if the *p-value* is below 0.05.

```
# Make the pred column
res$Pred <- 0
res$Pred[which(res$p_value < 0.05)] <- 1

# Apply the confusion.matrix function
obs.n.pred <- as.data.frame(cbind('Obs'=res$AnySickness, 'Pred'=res$Pred))
confusion.matrix(obs.n.pred, Title='p-value, naive')
```



Notice that these values are exactly identical to those we saw for *d2_scaled*, as would be expected.

7.5 Discussion of the method

The whitening–Mahalanobis approach converts a correlated error vector into a single χ^2 statistic. This enables *simple, interpretable* control charts for multivariate forecast monitoring and can be implemented in a few lines of R code.

Equation 5 shows that under correct model specification, the Mahalanobis distance provides a direct, distribution-free mapping from $\mathcal{N}(\mathbf{0}, Q_t)$ to a χ^2 distribution. In practice you should:

- verify that Q_t is positive-definite (or apply a near-PD repair),
- choose α according to the desired false-alarm rate (e.g. 5 %), and
- consider robust alternatives (e.g. t-based distances) if heavy tails are suspected.

It is worth noting that although the performances achieved with this data set were not impressive, other studies have shown that the Mahalanobis distance method can be useful on other data sets, e.g., when using milk yield and composition data to predict the onset of mastitis (Jensen et al., 2017).

7.6 List of references

Jensen, D. B., Toft, N., & Kristensen, A. R. (2017). A multivariate dynamic linear model for early warnings of diarrhea and pen fouling in slaughter pigs. *Computers and Electronics in Agriculture*. https://doi.org/10.1016/j.compag.2016.12.018

8 Using DLM outputs as inputs for a Random Forest

8.1 Introduction to the example

In this set of examples, we demonstrate the use random forests to detect when calves are sick (including respiratory infections), based on the output of a DLM combined with non-filtered variables form the original dataset that indicate if calves are sick or healthy. Thus, we use DLM to pre-process the data, and we use the random forest as a secondary model to make the final prediction.

We will assess the performance of the random forest by means of per-herd cross-validation, where each herd will iteratively be held out and used for validation, while the remaining herds will be used for training.

The random forest does not work with missing values. Therefore, before training, we need to create a version of the training set where the target and input variables do not contain “NA”.

8.2 Setting up your working environment

Here we will read in the data and prepare them to be used for training and testing our random forest model.

8.2.1 Load libraries

Use the *library* function to load the necessary packages.

```
library(randomForest)

library(pROC)

library(splitstackshape)
```

Use the *source* function to source the script called “FUNCTION COLLECTION - RANDOM FORESTS.R”. This script contains custom functions, used in the following example.

```
source("FUNCTION COLLECTION - RANDOM FORESTS.R")
```

8.2.2 Data preparations

We read in the file **DLM_preprocessed_data__NEW_WithoutActivity.RDS.**

The column “SickOrHealthy” shows us which observations belong to a “Sick” (1) vs a “Healthy” (0) calf. We can use the *table* function to see the distribution between classes of each column in the “SickOrHealthy” column.

```
# Get the data
Data <- readRDS('DLM_preprocessed_data__NEW_WithoutActivity.RDS')

# See the distribution of each health metric
table(Data[,"SickOrHealthy"])

##
##    0    1
##  850 1922
```

As seen in the result, there are 850 calves classified as "Healthy", and 1922 classified as "Sick".

We wish to do classification, which means the target value needs to be operated by R as a factor-type variable. However, right now the values in the "SickOrHealthy" column are numerical. We use the *as.factor* function to transform the "SickOrHealthy" column into a factor variable.

```
# Turn the target variable into a factor
Data$SickOrHealthy <- as.factor(Data$SickOrHealthy )
```

8.2.3 Defining variables

First, we will define the column names of the meta data (*meta.names*, used for identifying the individual calf and the herd it came from), the columns of the raw data which the DLM was applied to (*raw.names*), and the column names of the other variables which might be relevant for disease detection, but which were not suitable for DLM (*other_relevant.names*).

```
meta.names <- c(
  "Herd",
  "calf.herd",
  "date"
)

# Columns the DLM was applied to
raw.names <- c(
  "consumption_liters",
  "visitswent",
  "visits",
  "DrinkingSpeed"
)

# Define a list of names that are not suitable for DLM but might also be releva
nt
other_relevant.names <- c(
  "DayOfYear",
  "age_days",
  "DaySinceHousingDate",
  "visits_with_breakoff",
  "visitswoent"
)
```

Now we wish to define the various outputs from the DLM we wish to use as inputs for the random forest.

First, we get the column names which relate to the standardized forecasts of the DLM. Use the *colnames* function combined with the *grep* function to select the the column names which contain the pattern "ut_" from the *Data*; call the resulting vector "ut.names". Print *ut.names* to the console to make sure they are correct.

```
  # Get standardized forecast error names
  ut.names <- colnames(Data)[grep(pattern = 'ut_', x = colnames(Data))]
  print(ut.names)
```

```
## [1] "ut_consumption_liters" "ut_visitswent"         "ut_visits"
## [4] "ut_DrinkingSpeed"
```

Next, we get the column names which relate to the filtered means of the DLM. Use the *colnames* function combined with the *grep* function to select the the column names which contain the pattern "mt_" from the *Data*; call the resulting vector "mt.names". Print *mt.names* to the console to make sure they are correct.

```
  # get filtered data names
  mt.names <- colnames(Data)[grep(pattern = 'mt_', x = colnames(Data))]
  print(mt.names)

## [1] "mt_consumption_liters"   "mt_d.consumption_liters"
## [3] "mt_visitswent"           "mt_d.visitswent"
## [5] "mt_visits"               "mt_d.visits"
## [7] "mt_DrinkingSpeed"        "mt_d.DrinkingSpeed"
```

8.3 Training and validating the random forest

In this example, we show how to make a random forest which takes as input the standardized forecast errors and filtered means from the DLM, as well as other potentially relevant variables that were not fit to be modelled in a DLM , like the age in days, number of days since housing or number of visits without entitlement.

Four separate random forests will be made in a per-herd cross-validation scheme. All random forest models in this example will be made with default settings for *ntree*, *maxnodes*, *nodesize*, etc., for the sake of simplicity.

8.3.1 Make an out.all data frame

Before anything else, we need to create an empty data frame, in which we will store the performance values achieved in the per-herd cross validation.

Use the *data.frame* function to create an empty data frame called "out.all".

```
  out.all <- data.frame()
```

8.3.2 Assess model using all available input data and per-herd cross-validation

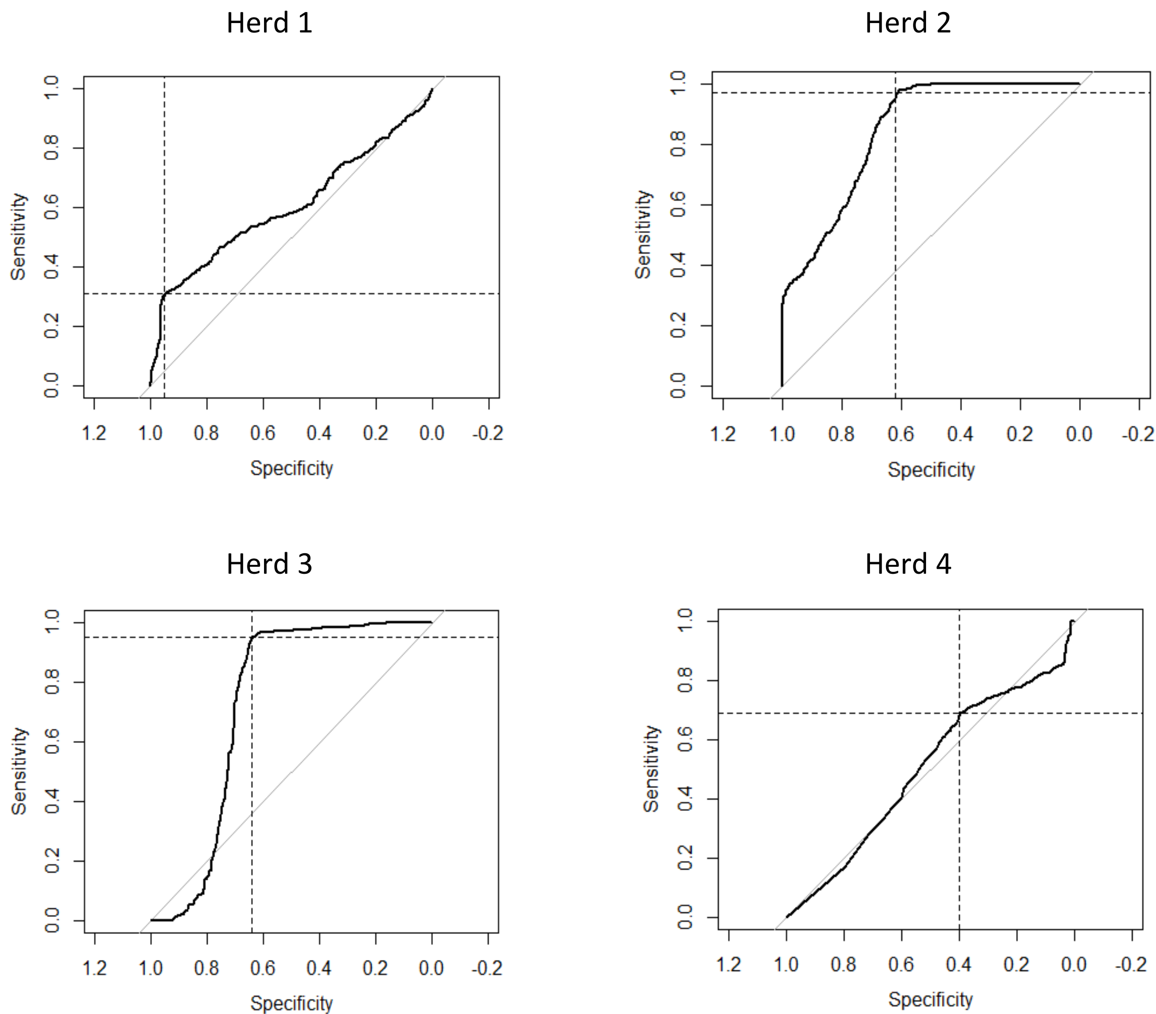
Then, we make a vector called "relevant.names", containing "mt" (filtered DLM means), "ut" (standardized forecast errors of DLMs), and "other", representing the three different types of input variables we defined earlier.

We use the following code to iteratively hold out one herd as the test set and make a training set without missing values from the remaining herds. Additionally, in this code we define the formula to be used for the random forest using the *relevant.names* vector, train the random forest on the training set, and apply the trained model to the test set. The script also evaluates the performance in terms of the area under the ROC curve which, like the MMA used earlier, provides a combined assessment of the Se and Sp. For each test herd, the performance gets added to the *out.all* data frame.

After the *for*-loop, we print the *out.all* to the console.

```
# Make the formula
Formula <- as.formula(paste('SickOrHealthy ~ ' , relevant.names_F) )
```

```
#Print it to console
print(Formula)

## SickOrHealthy ~ mt_consumption_liters + mt_d.consumption_liters +
##     mt_visitswent + mt_d.visitswent + mt_visits + mt_d.visits +
##     mt_DrinkingSpeed + mt_d.DrinkingSpeed + ut_consumption_liters +
##     ut_visitswent + ut_visits + ut_DrinkingSpeed + DayOfYear +
##     age_days + DaySinceHousingDate + visits_with_breakoff + visitswoent

# Train random forest model in a 10-fold cross-validation
for(herd in unique(Data$Herd)){

  # Print fold to see how far along we are
  print(herd)

  # Create the training and test set for this fold
  training.set <- subset(Data, Data$Herd != herd)
  test.set <- subset(Data, Data$Herd == herd)

  # Make a new data set with no missing values in the relevant columns
  training.set_A <- training.set[,c('SickOrHealthy', relevant.names)]
  training.set_A <- na.omit(training.set_A)

  # Train the random forest
  RF <- randomForest(formula = Formula,
                     data = training.set_A
                    )

  # Make predictions on the test set of this fold
  pred <- predict(object = RF,
                  newdata = test.set,
                  type = 'prob')

  # Extract the column of pred named "1", this column gives us the estimated pr
obability for mastitis
  pred <- pred[,'1']

  # Use the apply.roc function to get performance values
  out <- apply.roc(obs.test = test.set$SickOrHealthy,
                   pred.test = pred)

  # Add a column identifying the herd and a column identifying the inputs
  out <- cbind('Herd'=herd, out, 'Inputs'="mt_ut_other")

  # Add out to out.all
  out.all <- rbind(out.all, out)
}
```




```
out.model <- subset(out.all, out.all$Inputs == "mt_ut_other")

print(out.model)

##    Herd  AUC    AUC.CI MMA.max   Se   Sp Threshold      Inputs
## 17    1  0.6 0.57-0.63    0.63 0.31 0.95     0.564 mt_ut_other
## 18    2 0.84 0.82-0.86   0.795 0.97 0.62     0.179 mt_ut_other
## 19    3 0.72  0.7-0.74   0.795 0.95 0.64     0.377 mt_ut_other
## 20    4  0.5 0.48-0.52   0.545 0.69  0.4     0.939 mt_ut_other
```

We see that the AUC is significantly greater than 50 %, meaning the model performs significantly better than random guessing, for all herds except Herd 4. The best MMAs, depending on the alarm threshold, are between 54 % and 79.5 %.

Furthermore, we could try various combinations of including and omitting the different types of inputs in the *relevant.names* vector, but that would go beyond the scope of this example.

9 Overall conclusion

This deliverable shows that there is no single “silver-bullet” alarm algorithm: each layer we tested—classical statistical control charts, multivariate distances and machine-learning classifiers—comes with distinct trade-offs in ease of deployment, interpretability and diagnostic performance. A brief comparison is given below to guide future field implementations.

Method	Main strengths	Main weaknesses
Shewhart chart + Montgomery rules	• Easiest to explain to farmers and integrators • Very low computational cost • Excellent specificity when tuned	• Relies on independence and Normality of residuals • Requires per-stream limits once for Phase II monitoring
V-mask on CUSUM	• Detects small, sustained drifts better than Shewhart • Highly visual: the “V” overlay helps users grasp why an alarm fired	• Alarm comes “dist” days after the event, so corrective action is delayed • Tuning (angle, dist) is data-specific • Same Normality/independence assumptions as CUSUM
Tabular CUSUM	• Parametric design (k, h) lets users target specific ARLs • The only univariate chart that (in theory) minimises average detection delay for a chosen shift size	• Choosing k/h is non-trivial when series are short • False alarms increase if residuals are autocorrelated
Mahalanobis distance (whitening + χ^2)	• Handles correlated multivariate errors in a single statistic • Distribution-free threshold once Q_t is positive-definite	• Requires stable covariance estimates; heavy tails degrade calibration
Random Forest on DLM outputs	• Captures non-linear patterns and variable interactions • Easily extended with new sensor modalities	• Requires labelled events in the training data • Less transparent than control-chart rules

In these examples, the highest performances by far were achieved with the Random Forest method, which suggests that (at least for these data) the ability to include additional relevant variables in addition to the DLM outputs improves the detectability of diseases in calves. We lack, however, an external validation for the method, where the model is tested on another population at another time period.