# Introduction

The current document describes the design, implementation and validation of a predictor system, based on logistic regression

## Document summary

The document is structured as follows.

* Predictor architecture overview
* Dataset description
  + Raw input dataset reference
  + Preprocessed input dataset: windowed event tables
  + Output dataset description
* Data separation
* Model selection
  + Initial variable selection
  + Model selection
  + Model refinement
* Prediction
  + Validation error rate
  + Test error rate
* Future work
* Conclusions

# Predictor architecture overview

The predictor is composed of the following modules

* Data preprocessing module
  + Generation - Event table
  + Generation - Windowed event table
  + Separation - training and test subsets
* Model selection module
  + Model selection - variable subset
* Prediction module
  + Predictor - Logistic Regression
  + Performance evaluation

# Dataset description

The dataset family used to validate this predictor is SCF-1 - a list of log files from Santander Consumer Finance. A detailed description of this dataset family is provided in document "SCF-1-01-Dataset-Description.docx".

For the initial experiment, we will use data ranging from 01/02/2014 to 11/03/2014, with a total of 39 sampled days. Future experiments will extend this data with samples using a new data format, starting on 12/03/2014, for a total of ~100 sampled days.

The version of the SCF-1 dataset that will be used as system input is SCF-1-4d, a simplified event table that removes secondary fields and keeps only the date, event type, and event node for each event. In addition, non-failure events are filtered from this table. These filtered events, as well as the following fields are left out of the current analysis, but could be considered in later iterations:

* severity
* CreatedBy
* clearedon
* clearedBy
* msg

## input dataset

The initial dataset, SCF-1-4d, is a table with the following columns

* date
* node
* type

Some statistics on the data

* events. 3178
* duration: 39 days, from 01/02/2014 to 11/03/2014
* nodes: 27 (out of a total of 38)
* event types: 10 (out of a total of 37, 27 non-failure event types left out)

The dataset is stored in /Prediction-3-1-data/prediction-3-1-events-allnodes.csv. In R, it is stored as the variable 'event\_table\_1'.

>evTable <-read.csv(file.path(visualizationFolder,"events-allnodes.csv")) %.%

mutate(date=ymd\_hms(date)) %.% tbl\_df() %.%

select(date,type, node)

>quickStr(evTable )

'data.frame': 3178 obs. of 3 variables:

$ date: POSIXct, format: "2014-02-01 00:21:30" "2014-02-01 02:05:30" "2014-02-01 05:44:13" ...

$ node: Factor w/ 27 levels "aswpeuid01.sceu.corp",..: 26 27 27 27 26 27 26 26 ...

$ type: Factor w/ 10 levels "BLANK","t.0x10d35",..: 7 7 7 7 7 7 7 7 ...

> summary(eventTable1)

date node type

Min. :2014-02-01 00:21:30 lbpeuin01.sceu.corp :1402 t.0x3b70012:1556

1st Qu.:2014-02-19 19:00:42 lbpeuin02.sceu.corp :1377 t.0x3b70014: 874

Median :2014-03-04 01:51:17 lbpeuid01.sceu.corp : 153 BLANK : 661

Mean :2014-02-27 14:34:45 lbpeuid02.sceu.corp : 153 t.0x220001 : 73

3rd Qu.:2014-03-05 19:03:46 dswpeuin01.sceu.corp: 24 t.0x3b70019: 4

Max. :2014-03-11 23:59:10 dswpeuin02.sceu.corp: 18 t.0x10d35 : 3

(Other) : 51 (Other) : 7

## Preprocessed input dataset: windowed event tables

### Processing

We define the following function to count messages of a certain type per time window (see utils/eventsPerSample.R)

require(plyr)

require(dplyr)

require(lubridate)

require(ggplot2)

require(reshape2)

#usage

#getEventsInWindow(date,.data,windowSize=300, direction="backward")

getEventsInWindow<-function(windowOrigin,data,windowSize=300,direction="forward", offset=0)

{

# filter: events before or after window origin

data.filtered<-data.frame()

# check window origin offset

if(is.null(offset) || offset<0){offset<-0}

if(offset>windowSize){offset<-windowSize}

if(direction=="forward")

{data.filtered<-data %.%

filter (date>windowOrigin+offset & date <= windowOrigin+windowSize)}else

if(direction=="backward")

{data.filtered<-data %.%

filter (date<=windowOrigin-offset & date >= windowOrigin-windowSize)}else

if(direction=="both")

{data.filtered<-data %.%

filter (date<=windowOrigin+windowSize & date >= windowOrigin-windowSize)}

# if empty window, return row marking this date as having no events in window

if(nrow(data.filtered)==0) {

return(data.frame(date=windowOrigin, type="NO.EVENTS", evInWin=TRUE))}

#if (countEvents) result <- data.filtered %.%

#group\_by(type) %.% summarise(minDist1=n())

result<-data.filtered %.% group\_by(type) %.% summarise(evInWin=n()>0) %.%

mutate(date=windowOrigin) %.% select(date,type,evInWin)

}

applyGetEventsInWindow<-function(data,windowSize=300,direction="forward", offset=0)

{

tmp<-ldply(unique(data$date),

getEventsInWindow,

data,

windowSize=windowSize,

direction=direction,

offset=offset)

dcast(tmp, date ~ type, fun.aggregate= any) %.% tbl\_df()

}

Function getEventsInWindow takes a date 'date' and an event data frame '.data', and returns a table indicating which event types are included in the data frame, within a given time window relative to that date. For example:

> tmp<-getEventsInWindow(evTable$date[1200], evTable, direction="backward")

> tmp

Source: local data frame [2 x 3]

date type evInWin

1 2014-03-04 02:19:49 t.0x3b70012 TRUE

2 2014-03-04 02:19:49 t.0x3b70014 TRUE

The function getEventsInWindow takes the following parameters

* windowOrigin - A POSIXct date, to be used as origin for the data processing
* .data - a data frame with a 'date' column, and a 'type' column
* direction - a string, indicating the position of the event window relative to the origin: "backward", "forward", or "both".
* windowSize - a number, indicating the number of seconds between the windowOrigin and the end of the time window.
* offset - a number, indicating the seconds between the windowOrigin and the start of the time window.

Function applyGetEventsInWindow applies the previous function over all dates in the data frame, returning a data frame with a row for each date, and a column for each event type, indicating whether an event of that type is included within a given time window, relative to the row date:

> tmp<-applyGetEventsInWindow(evTable)

Source: local data frame [6 x 12]

date NO.EVENTS BLANK t.0x10d35 t.0x10daa t.0x10f03 t.0x210027 t.0x220001

1 2014-02-01 00:21:30 TRUE FALSE FALSE FALSE FALSE FALSE FALSE

2 2014-02-01 02:05:30 TRUE FALSE FALSE FALSE FALSE FALSE FALSE

3 2014-02-01 05:44:13 TRUE FALSE FALSE FALSE FALSE FALSE FALSE

4 2014-02-01 08:12:01 TRUE FALSE FALSE FALSE FALSE FALSE FALSE

5 2014-02-01 08:28:47 TRUE FALSE FALSE FALSE FALSE FALSE FALSE

6 2014-02-01 09:10:25 TRUE FALSE FALSE FALSE FALSE FALSE FALSE

Variables not shown: t.0x3b70012 (lgl), t.0x3b70014 (lgl), t.0x3b70019 (lgl), t.0xc40003

(lgl)

This function takes the following parameters (defined the same as the function above)

* data
* windowSize
* direction
* offset

Using these functions, we generate a series of input tables for our predictor, stored in the object 'winEvTable'.

* allnodes
  + back.5m
  + back.30m
  + fw.5m
  + fw.5.30m
* byNode
  + node1
    - backward.5m
    - backward.30m
    - forward.5m
    - forward.30m

Note: the tables under 'byNode' are a work in progress

The code to generate these tables is shown below

require(plyr)

require(dplyr)

require(lubridate)

# folder to load utility functions

utilsPath<- "C:/Users/capelastegui/workspace/OFP/utils"

# folder to load data

baseDataPath <- "C:/Users/capelastegui/workspace/OFP/SCF/SCF-1/1-Data"

# Subfolder to load output

savePath <- file.path(baseDataPath,"4-VisualizationTables")

# load utility functions

source(file.path(utilsPath,"eventsPerSample.R"))

source(file.path(utilsPath,"plyr.nested.R"))

#load data from file

# Option 1: use visualization tables processing script

# source('C:/Users/capelastegui/workspace/OFP/SCF/SCF-1/2-R/preprocess/SCF-1-4-visualization-tables.R')

evTable <-read.csv(file.path(visualizationFolder,"events-allnodes.csv")) %.%

mutate(date=ymd\_hms(date)) %.% tbl\_df() %.%

select(date,type, node)

# Option 2: We can also generate table from SCF eventTable

# events\_allnodes: table with events(date,node,type) for all system nodes

# events\_allnodes<-eventTable %.% filter(severity!="BLANK") %.% select (date,node,type) %.% refactor()

# getWinEvTables generates a list of tables listing, for each event, the observed events in a time window

# see utils/eventsPerSample

# By default, generate the following tables

# - b.5m : For each event, list events in between t, t-5min

# - b.30m: Same, between t, t-30min

# - f.5m : Same, between t, t+5min

# - f.5.30m : Same, between t+5min, t+30min

getWinEvTables <-function(data, # An event table

backward=list(b.5m=5\*60,b.30m=30\*60), # List of window limits for backward windows

forward=list(f.5m=5\*60,f.5.30m=30\*60), # List of window limits for forward windows

offset.back=list(), # List of window offsets for backward windows

offset.fw=list(f.5.30m=5\*60)) # List of window offsets for forward windows

{

# generate tables with 'backward' windows

back<-llply.parallel.multilist(list.ref=backward,

list.multi=list(windowSize=backward, offset=offset.back),

n=1,

data=data,

.fun=function(sizeOffsetList,data)

{applyGetEventsInWindow(data,

windowSize=sizeOffsetList$windowSize,

offset=sizeOffsetList$offset,

direction="backward")}

)

# append windowSize, offset as attributes of 'back' list

attr(back,"windowSize")<-backward

attr(back,"offset")<-offset.back

# generate tables with 'forward' windows

fw<-llply.parallel.multilist(list.ref=forward,

list.multi=list(windowSize=forward, offset=offset.fw),

n=1,

data,

.fun=function(sizeOffsetList,data)

{applyGetEventsInWindow(data,

windowSize=sizeOffsetList$windowSize,

offset=sizeOffsetList$offset,

direction="forward")}

)

# append windowSize, offset as attributes of 'fw' list

attr(fw,"windowSize")<-forward

attr(fw,"offset")<-offset.fw

return ( list(back=back,fw=fw) )

}

winEvTable <-getWinEvTables(evTable)

## Output dataset description

# Data separation

## Separation 1: data from 01/02 to 11/03

For the first experiment, we choose a straightforward scheme to split between training and test data

* Train: data from 01/02 to 28/02
* Test: data from 01/03 to 11/03

# separation of training and test data

date.trainStart <- dmy("1-2-14")

date.trainEnd <- dmy\_hms("2-3-14 23:59:59")

trainInterval <- date.trainStart %--% date.trainEnd

winEvTable.train <- llply(winEvTable,

function(tableList)

{llply(tableList,

function(table){table %.% filter (date %within% trainInterval)}

)

})

winEvTable.test <- llply(winEvTable,

function(tableList)

{llply(tableList,

function(table){table %.% filter (!date %within% trainInterval)}

)

})

Data for the training and test set are stored in the following R variables

* trainTables
* testTables

# Model selection

## Initial model selection

### Model Selection 1: event types by node

In this experiment, we use as base predictors:

* For each node, a boolean variable indicating the presence of events within a past observation window before the present event.
  + this is measured for each observed event
  + An additional variable indicates when there are no events in the observation window for that event

Note: in this version, the past observation window includes the present event.

Likewise, we use as predicted variables:

* a boolean variable indicating the presence of an event of a given (type,node) combination within a future observation window after the present event

Note: No model checking has been performed at this point.

To simplify, in this experiment we will consider only:

* A single observation window
* A single prediction window

In terms of R variables, the model uses:

* From **trainTables**
  + an observation window table, representing all predictors

trainTables$back[[i]]

* + A column from a prediction window table, representing the predicted variable

trainTables$fw[[i]][,-1][[j]]

The code for generating all tables is shown below:

# we use 'trainTables' variable from pred-1-LogR-1-preprocess.R

# For each observation window table in trainTables$back,

## for each prediction window table in trainTables$fw,

### for each column (other than date) in the prediction window table,

### generate a table appending the obs. win. table to the column,

# and store all tables in a tree of lists.

#

# Example: An individual operation from the loop

# predictionTable <- cbind(trainTables$back[[1]],target=trainTables$fw[[1]][,-1][[1]])

# see plyr.nested::llply.ab

source('C:/Users/capelastegui/workspace/OFP/utils/plyr.nested.R')

predTables<-llply.ab(trainTables$back, trainTables$fw,

.fun2=function(a,b)

{

llply (a %.% select(-date),b,.fun=function(a1,b)

{cbind(b,target=a1) %.% tbl\_df()}

)}

)

predTables.test<-llply.ab(testTables$back, testTables$fw,

.fun2=function(a,b)

{

llply (a %.% select(-date),b,.fun=function(a1,b)

{cbind(b,target=a1) %.% tbl\_df()}

)}

)

In the current experiment, we have the following structure of prediction tables:

* b.5m
  + f.5m: list of 11 **prediction tables**
    - each table predicts 1 event type (including NO.EVENTS)
    - Each table has
      * a 'date' column
      * 10 predictor columns, one per event type
      * a 'target' column, the predicted variable
  + f.30m: list of 11 prediction tables, as above
* b.30m
  + f.5m: list of 11 prediction tables, as above
  + f.30m: list of 11 prediction tables, as above

For each prediction table, we build a logistic regression model of the form

glm(target ~ . - date, .data)

#### Note: code bug

The initial version of this code had a bug that reversed the 'backward' and 'forward' elements of the generated prediction tables. As a consequence, predictors generated from this data should show incorrect results.

### Model Selection 2: event types, plus interactions (order 2), by node

### Model Selection - Future work

Other options to consider

* Inter-node interactions
* Variable counts
* Combination of multiple windows.

## Model Refinement

### Model Refinement 1: No model refinement

In this experiment, we perform no model refinement step - all variables from the original model are used to build our model.

### Model Refinement 2: Stepwise for all training set

### Model Refinement 3: Stepwise with crossvalidation

# Prediction

## Training error rate

### Experiment 1

This experiment uses the following parameters

* Data separation 1 (see 4.1)
* Model selection 1 (see 5.1.1)
* Model refinement 1 (see 5.2.1)

Here are the observed results:

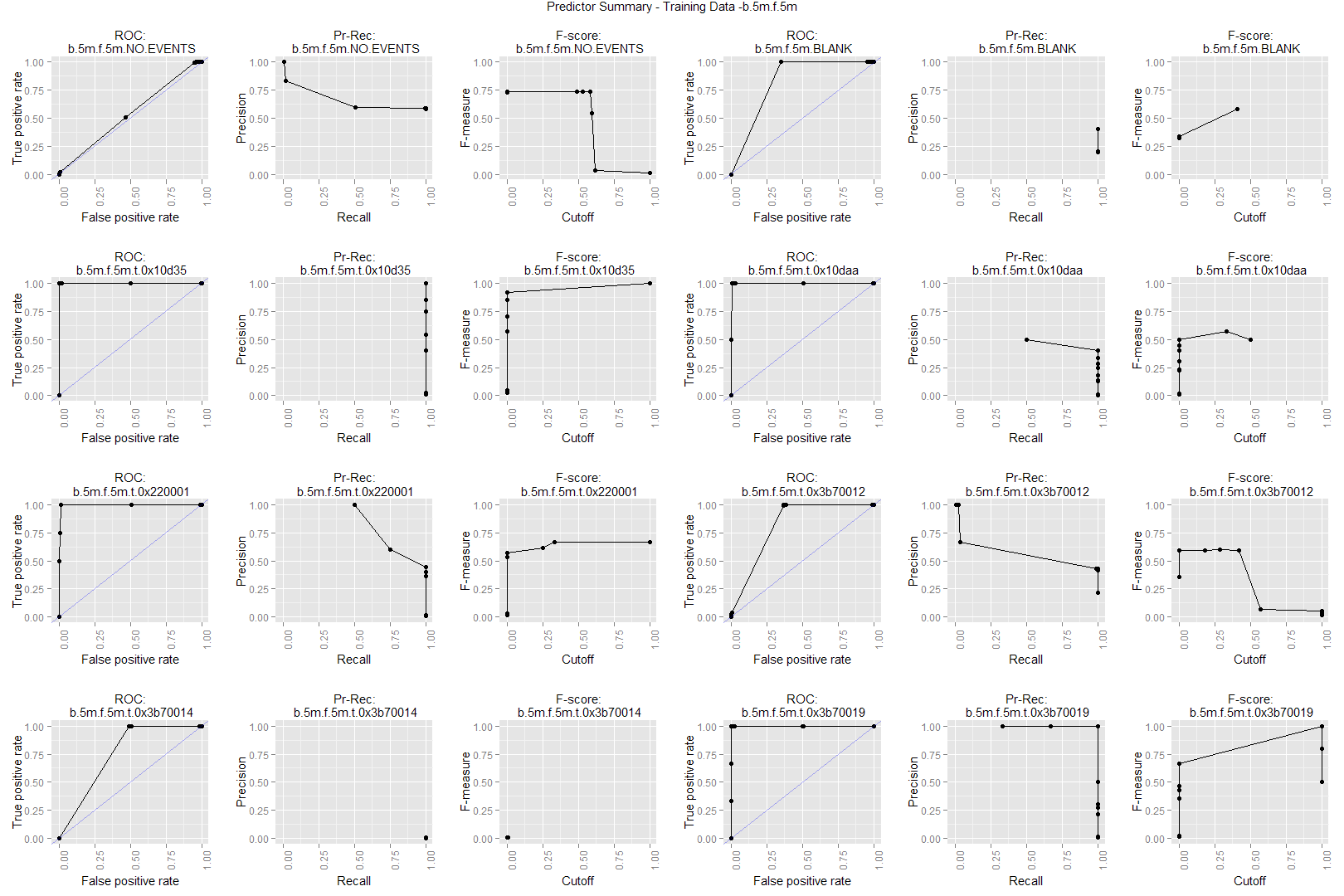


Figure 1 - Training, experiment 1, b5 f5

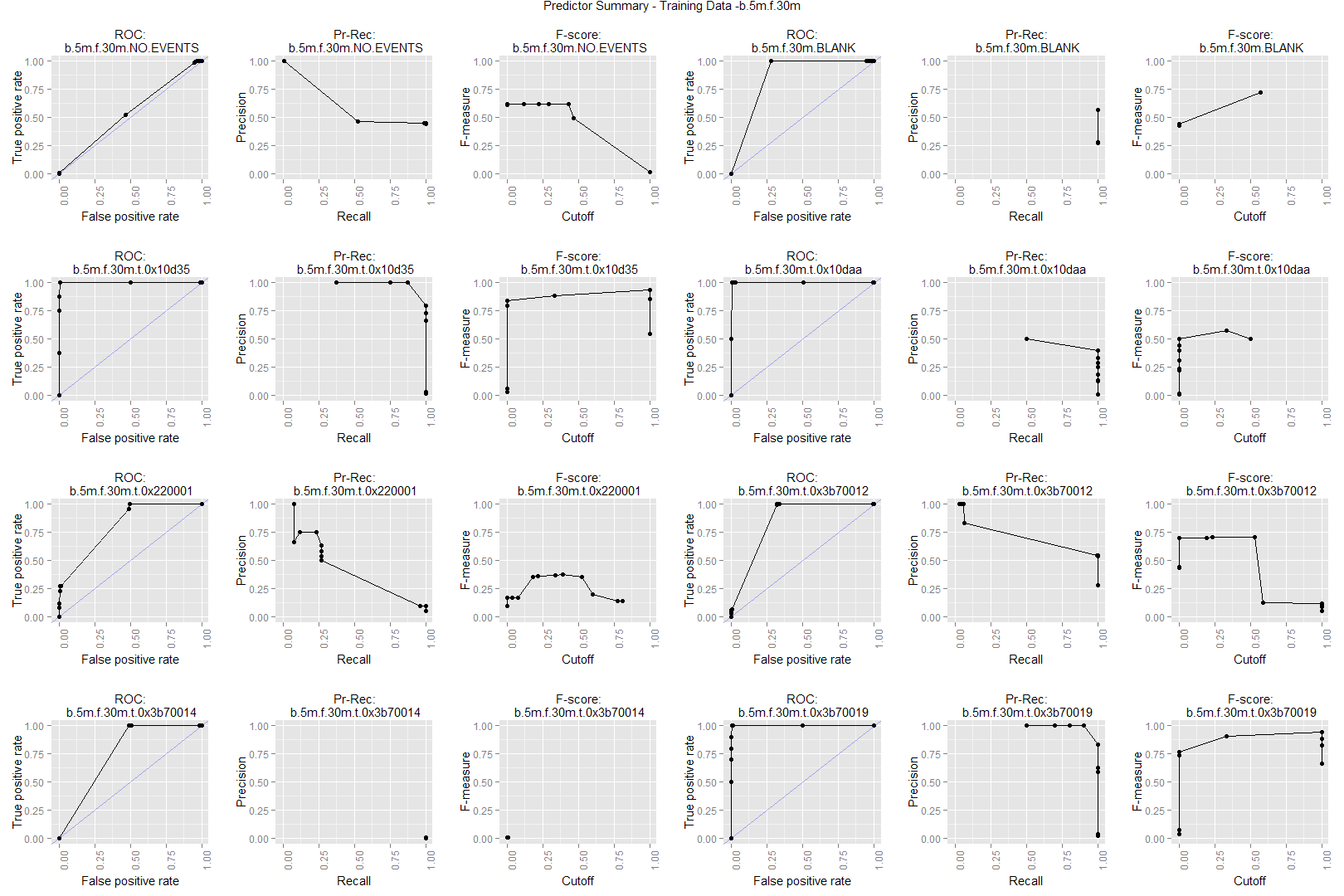


Figure 2 - Training, experiment 1, b5 f30

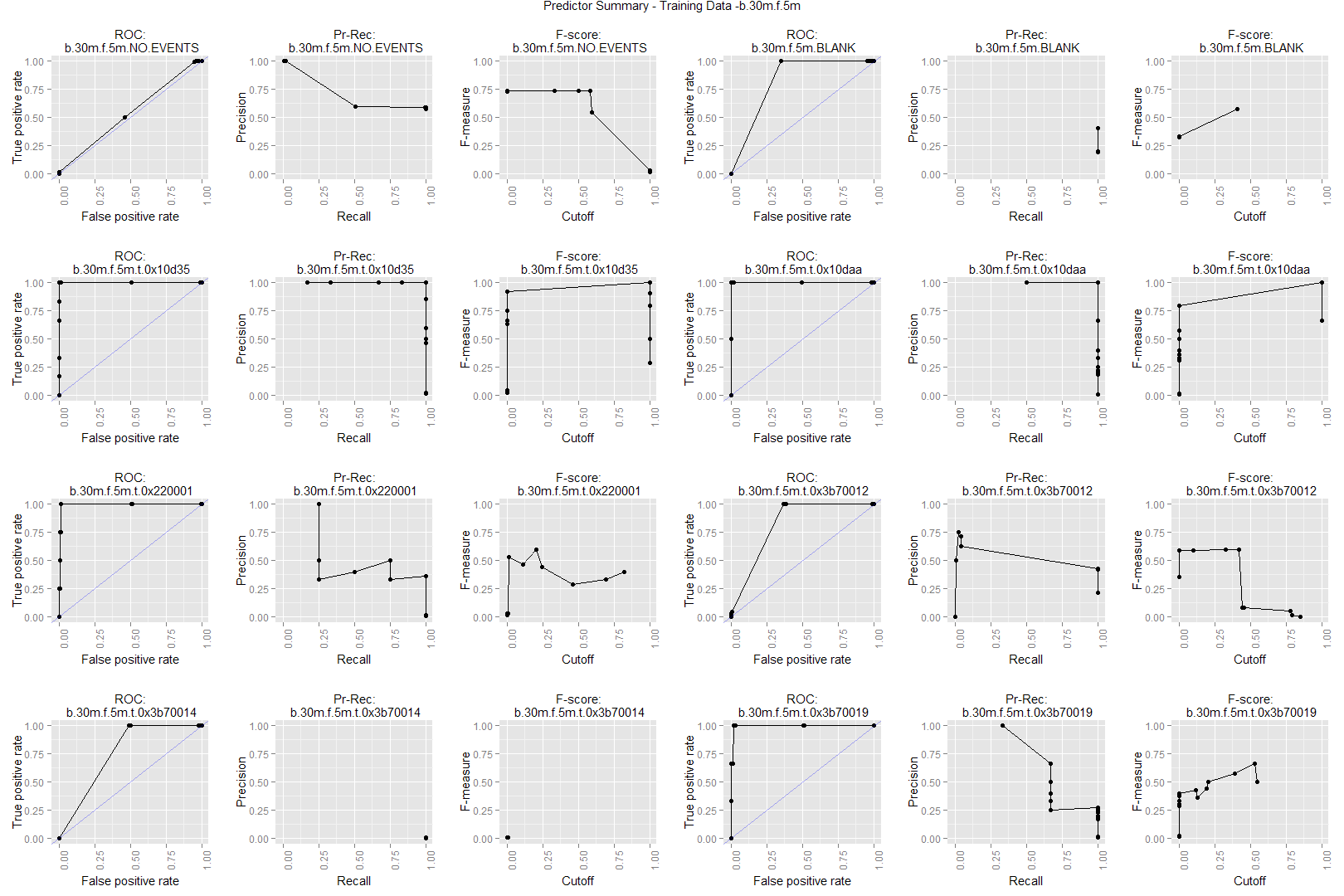


Figure 3 - Training, experiment 1, b30 f5

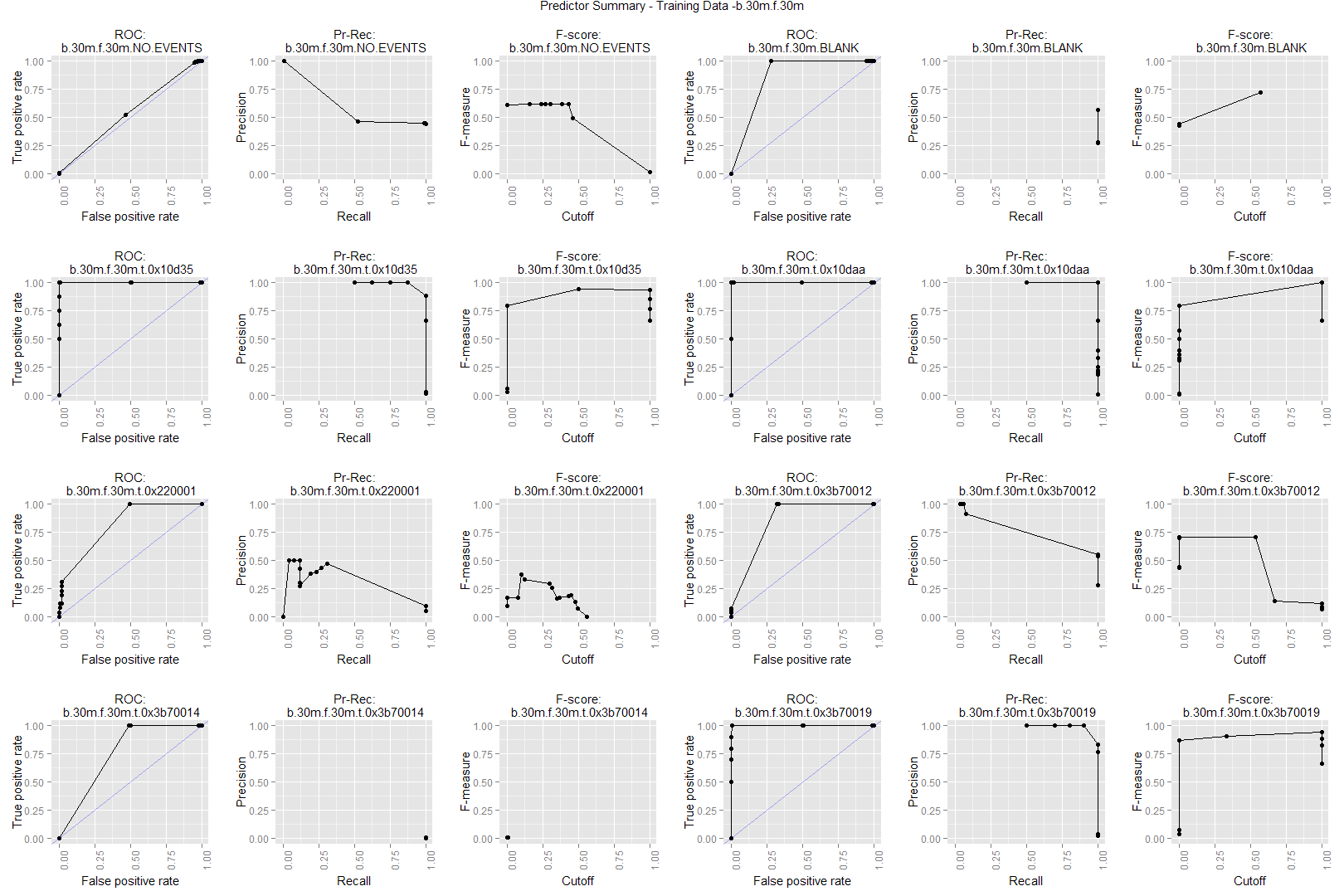


Figure 4 - Training, experiment 1, b30 f30

## Validation error rate

### Experiment 1

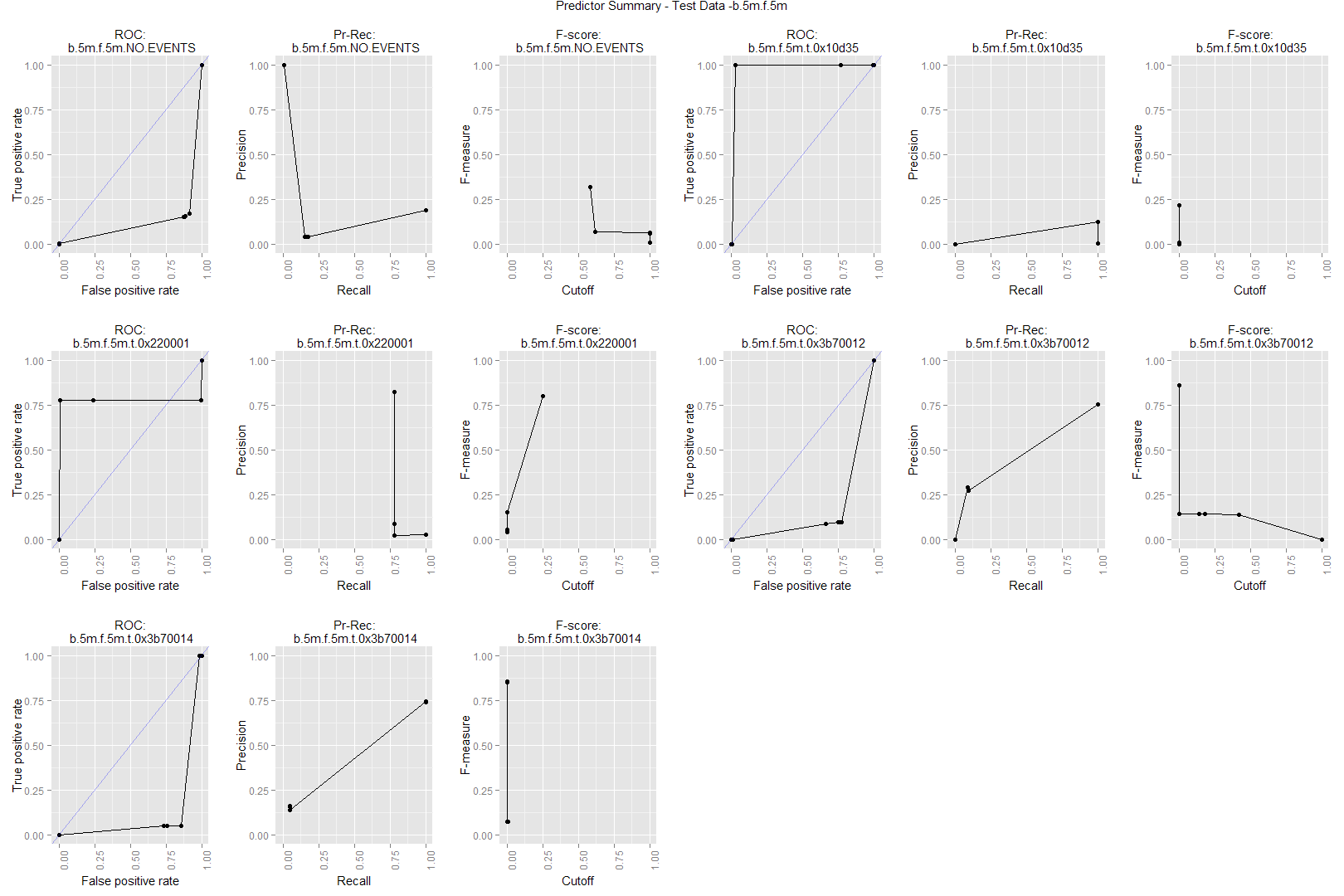


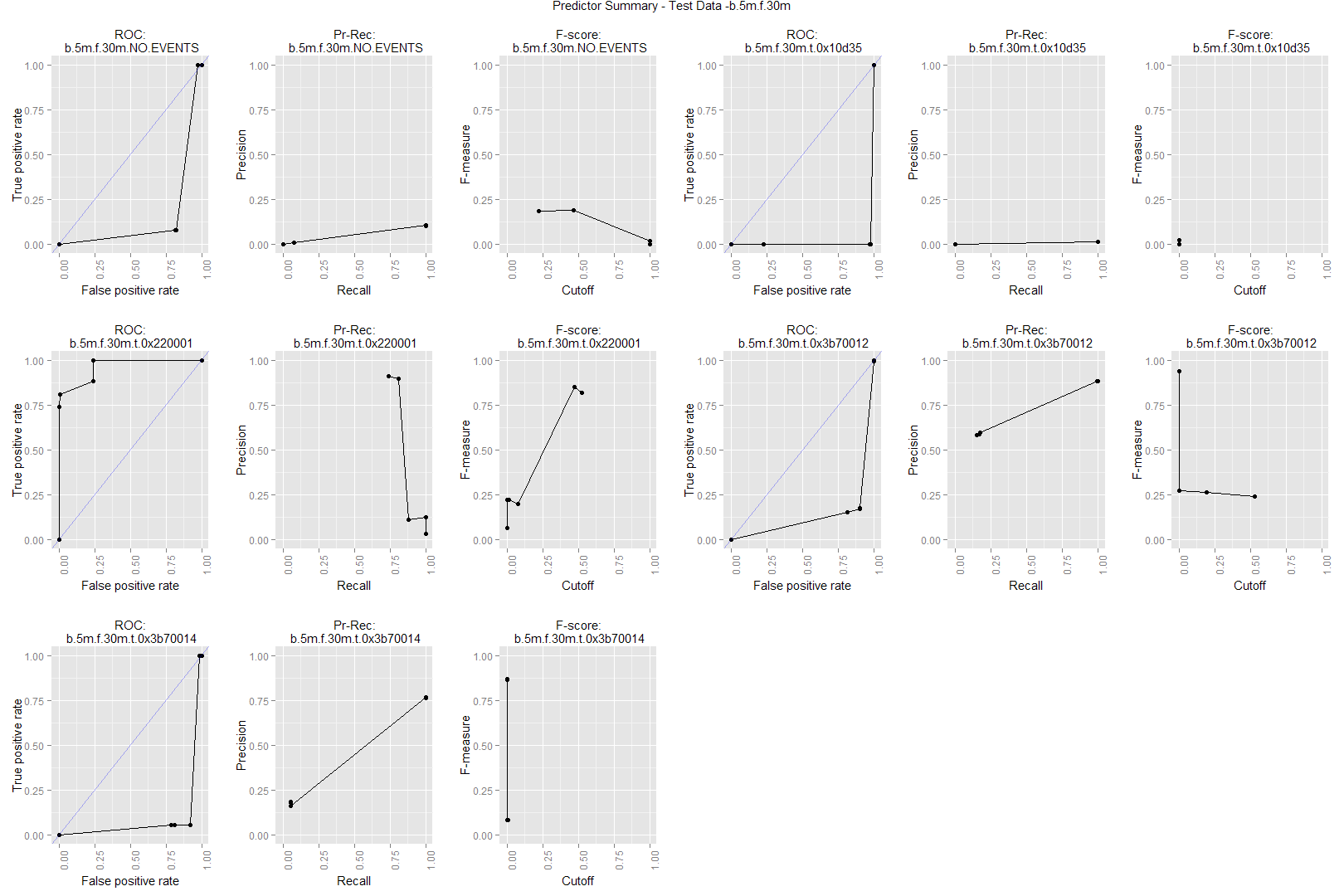
Figure 5 Test - Experiment 1 - b5,f5

Figure 6 Test - Experiment 1 - b5, f30

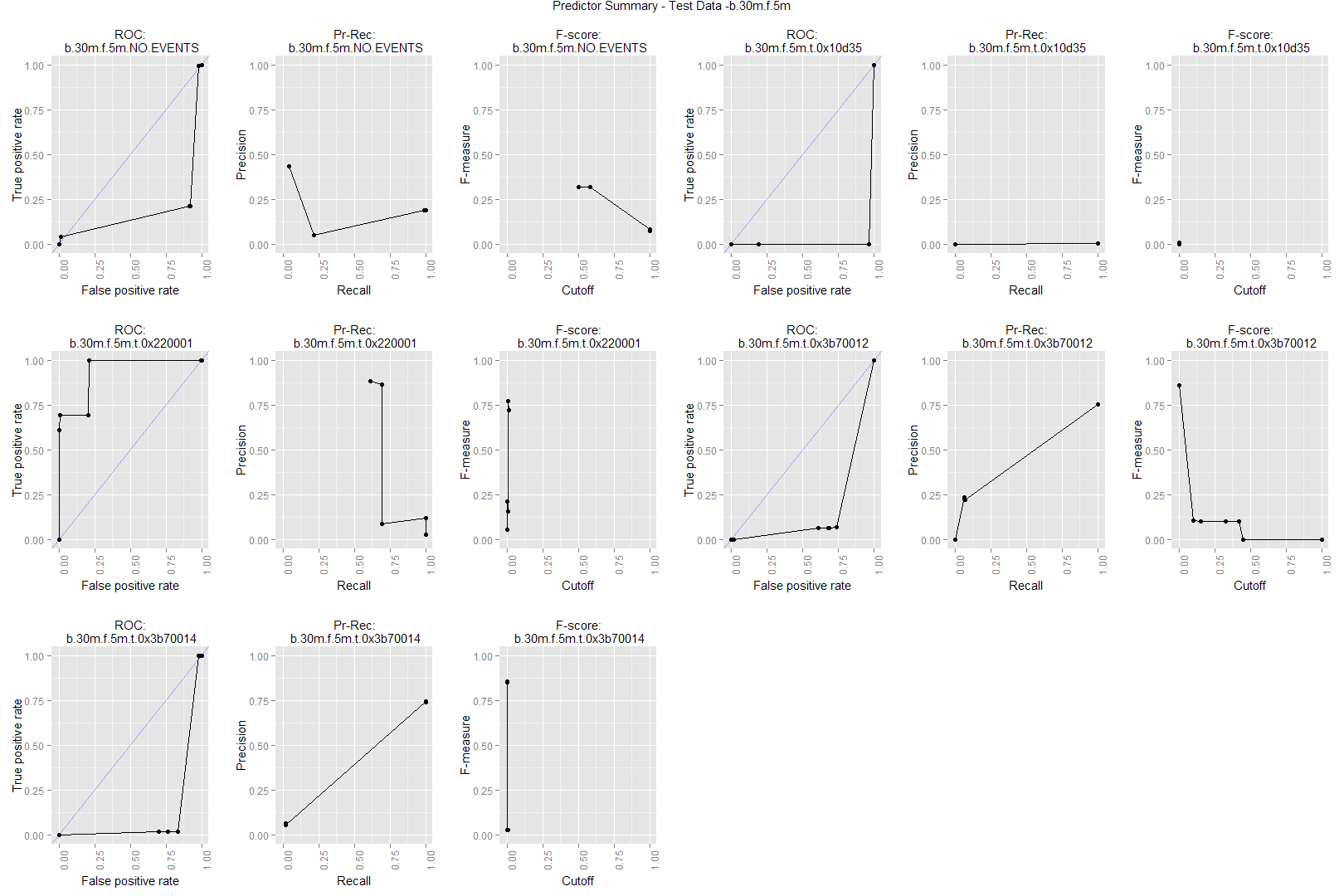


Figure 7 - Test - Experiment 1 - b30, f5

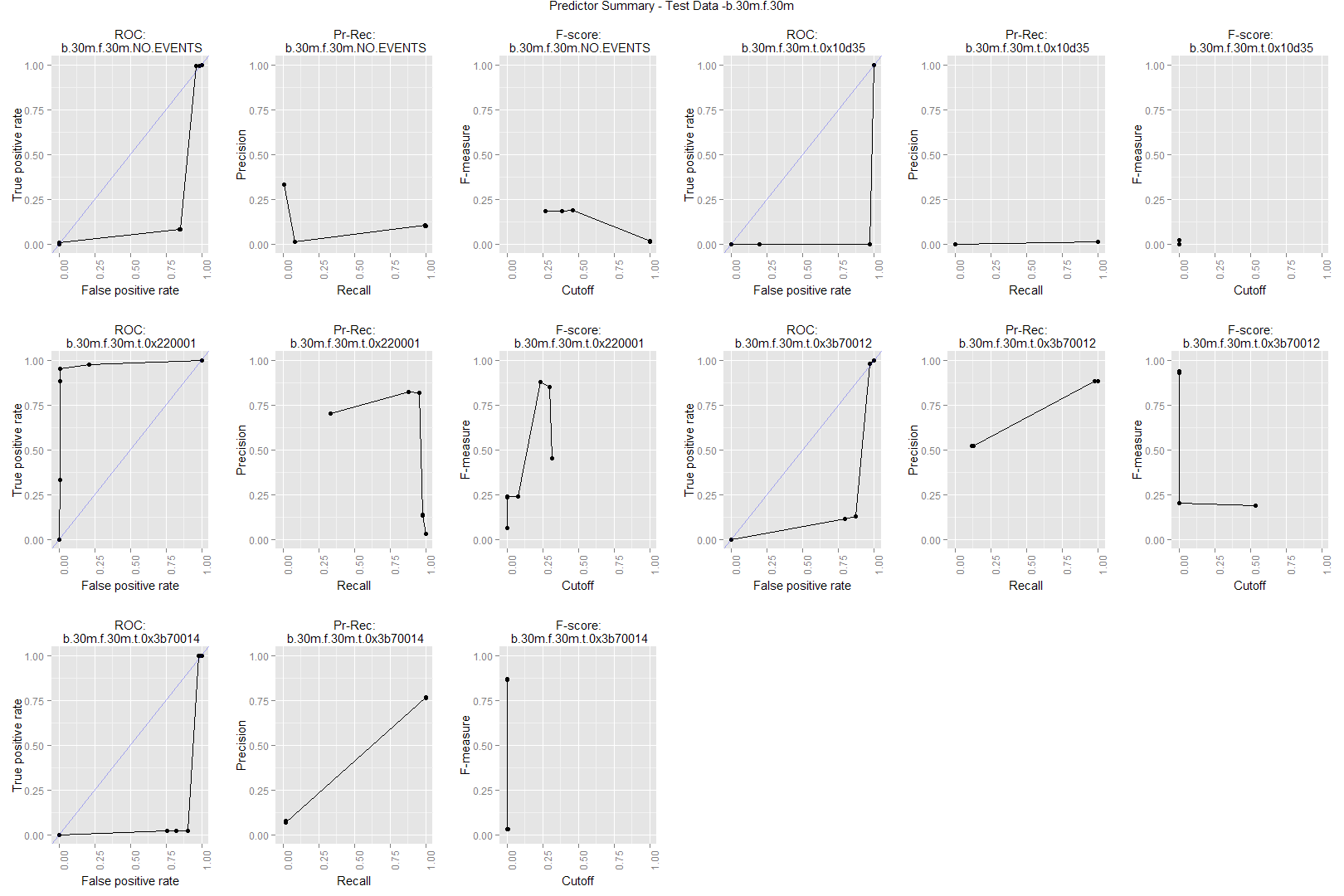


Figure 8 - Test - Experiment 1 - b30, f30

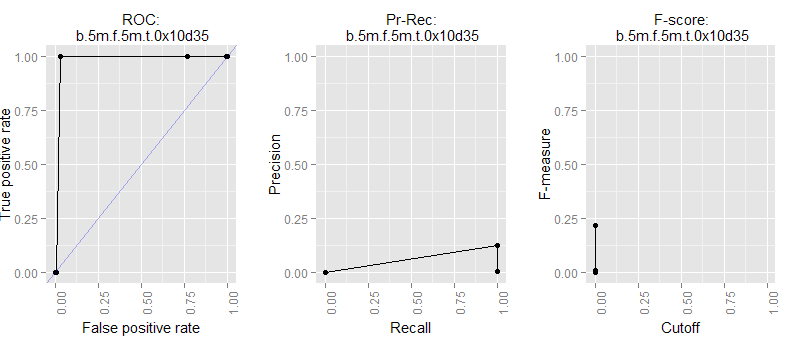
Comments about the results:

* we couldn't evaluate the performance of all predictors, because there weren't 2 classes (TRUE/FALSE) for some of them in the test data.
* check that test data is properly generated: is time threshold <= or <?
* find diagrams to represent event presence vs predictions
  + x= time?
  + contingency table
* Most predictors are terrible!
  + ROC well below random predictor
  + cutoff of either 0 or 1

TODO

* We need clear guidelines on how to evaluate predictors based on these diagrams, e.g.:
  + how much F-score is good enough
  + Is f-score worthless if cutoff is 0 or 1?
  + How to interpret: precision when recall=1
    - recall=1 when precision is low is not always useful
      * tons of false positives, it's what you get if always predict a 1
      * interesting if precision >> minimum precision you get when always predicting a 1
  + How to interpret: recall when precision=1
    - precision=1 and recall>0 is useful
      * prediction is always right, you catch some errors
  + Check theory on ROC, to better discuss it

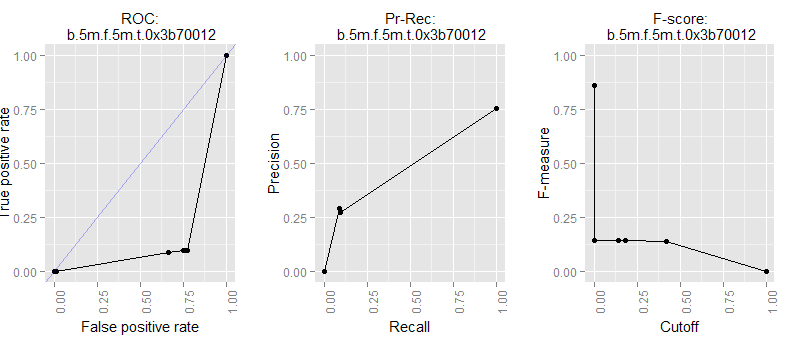
Examples:



Is this useful?

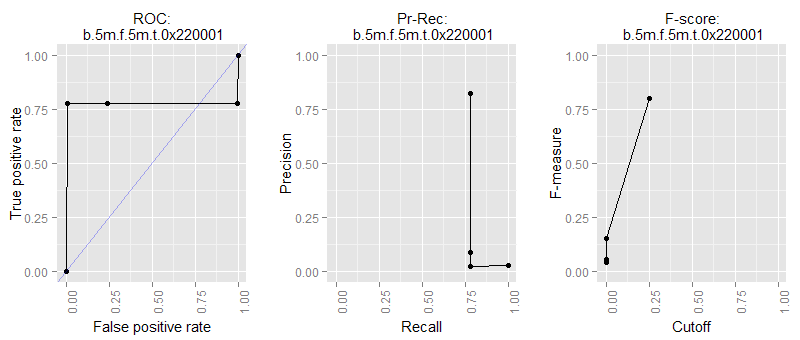
* We get all errors, but an error warning is only 12% reliable
* Threshold is very close to 0?
* Compare with "always predict TRUE" strategy

By contrast:



This one:

* has much better precision-recall than the prior example
* But is terrible compared to random predictor
* Why? - see with specific examples
* **What would be the F-score of a random predictor here? can I plot that?**

****

Finally, this one is rather good.

* Note that F-score, precision-recall is still a bit below example 2!

#### Misc. comments

TODO:

We need to pass test data to predictor.

It appears that the easiest way to do so is.

* Have predictionTables include both training and test data
* define a subset vector to split training and test data
* call 'lm' with the whole dataset, and subset vector

//Two common methods are the AUC (area under the curve) and the Log Likelihood.

<http://stats.stackexchange.com/questions/14803/presenting-logistic-model-fit-graphically>

A good reference for the kind of graphs you seem to want is John Fox, "Effect Displays for Multinomial and Proportional-Odds Logit Models," in Sociological Methodology (2006). See the citations to his earlier work. He implements these techniques in R and S-Plus in his book that accompanies his text on linear regression.

example lm

<http://cbio.ensmp.fr/~jvert/svn/tutorials/practical/linearregression/linearregression.R>

## Test error rate

# Future work

## Model extension

The following elements could be added to our model

* More data
* Filtered event fields
  + **severity**
  + CreatedBy
  + clearedon
  + clearedBy
  + msg
* Synthetic event fields
  + date\_wday
  + date\_mday
  + date\_h
  + cleared\_delay // time to clear event
  + node\_type // whether a firewall, dns, lb, or other
  + node\_group // group of node by location, or other criteria
  + number of event types in window
  + number of total events in window
* Synthetic window info
  + xxx\_count // count (minus 1) of event of a type/node. Useful for event bursts
* Non-failure events // thise are filtered out in the current model
* Resource summaries
  + xxx\_highload // boolean, whether a high load of resource x in window

## Window generation

We are counting events in multiple window sizes to provide input for the predictor.

Currently, these windows are of the form:

* w1: [-5\*60, 0)
* w2: [-30\*60,0]

We would like to be able to use non-overlapping windows, to have uncorrelated variables:

* w1: [-5\*60, 0)
* w2: [-30\*60,-5\*60]

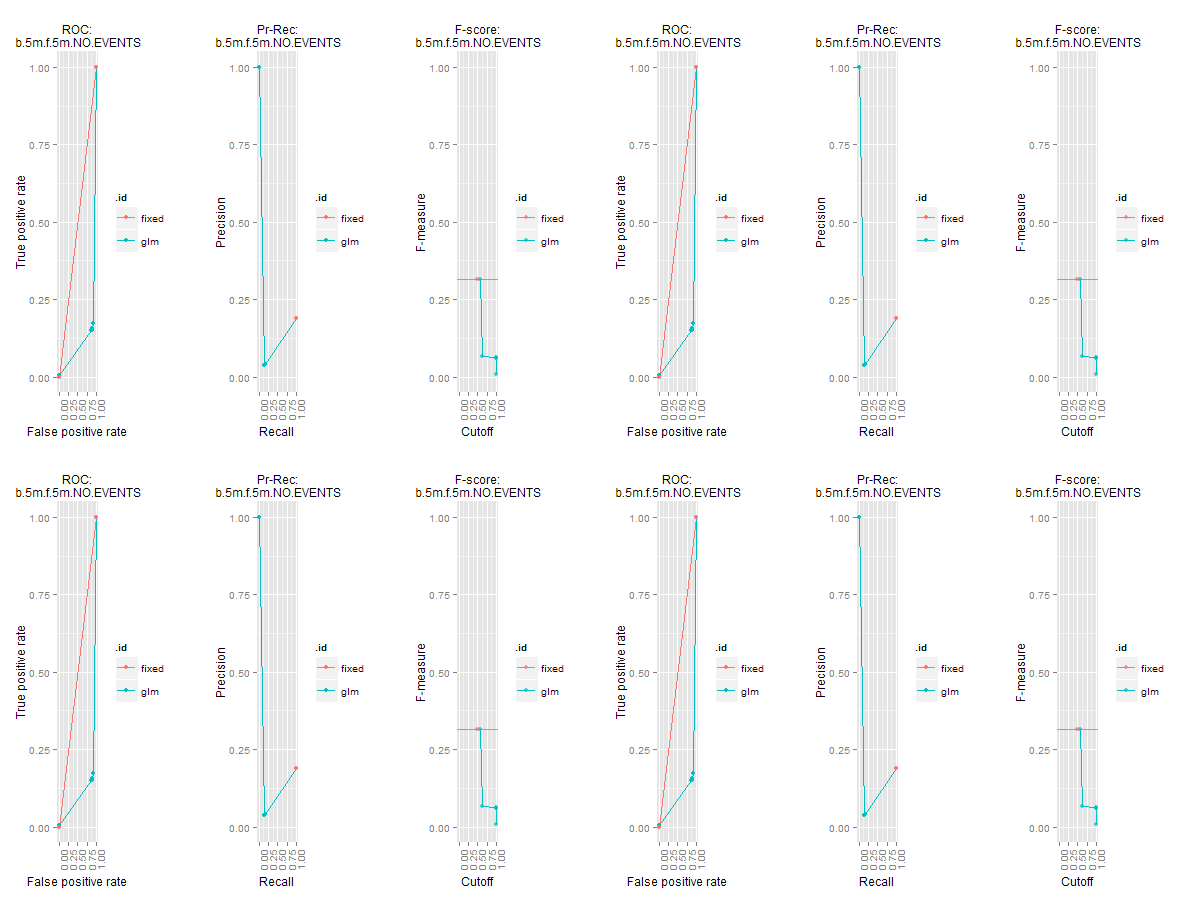
# Conclusions

# Appendix: misc. discussion

#### WIP: plotting stuff

I want to arrange groups of plots, hierarchically, as a grid

Issue: poorly fitted dimensions



Now: not have multiple legends

arrangeGrob can specify legend

<http://stackoverflow.com/questions/13649473/add-a-common-legend-for-combined-ggplots>

#https://github.com/hadley/ggplot2/wiki/Share-a-legend-between-two-ggplot2-graphs

g\_legend<-function(a.gplot){

tmp <- ggplot\_gtable(ggplot\_build(a.gplot))

leg <- which(sapply(tmp$grobs, function(x) x$name) == "guide-box")

legend <- tmp$grobs[[leg]]

return(legend)}

mylegend<-g\_legend(p1)

p3 <- grid.arrange(arrangeGrob(p1 + theme(legend.position="none"),

p2 + theme(legend.position="none"),

nrow=1),

mylegend, nrow=2,heights=c(10, 1))

#### error absolute units

Error in UseMethod("absolute.units") :

no applicable method for 'absolute.units' applied to an object of class "unit"

#### removing space between images

http://www.janeshdevkota.com/blog/removing-spaces-between-images-in-grid-arrange/

p1 theme(legend.position="none",

axis.text.x=element\_blank(),

axis.ticks.x=element\_blank(),

plot.margin=unit(c(1,1,-0.5,1), "cm"))

p2 theme(legend.position="none",

plot.margin=unit(c(-0.5,1,1,1), "cm"))

grid.arrange(p1,p2)

#### adding offset to window generation

getEventsInWindow<-function(windowOrigin,.data,windowSize=300,direction="forward", offset=0)

applyGetEventsInWindow<-function(.data,windowSize=300,direction="forward", offset=0)

getWinEvTables <-function(.data,

backward=list(b.5m=5\*60,b.30m=30\*60),

forward=list(f.5m=5\*60,f.30m=30\*60),

offset.back=list(),

offset.fw=list(f.30m=5\*60))

{

back<-llply(backward,

function(windowSize,.data)

{applyGetEventsInWindow(.data,windowSize=windowSize, direction="backward")},

.data)

attr(back,"windowSize")<-backward

fw<-llply(forward,

function(windowSize,.data)

{applyGetEventsInWindow(.data,windowSize=windowSize, direction="forward")},

.data)

attr(fw,"windowSize")<-forward

list(back=back,fw=fw)

}

#change to:

{

back<-llply.parallel.multilist(backward,

list(windowSize=backward,offset=offset.back)

function(windowSize,.data)

{applyGetEventsInWindow(.data,windowSize=windowSize, direction="backward")},

.data)

attr(back,"windowSize")<-backward

fw<-llply(forward,

function(windowSize,.data)

{applyGetEventsInWindow(.data,windowSize=windowSize, direction="forward")},

.data)

attr(fw,"windowSize")<-forward

list(back=back,fw=fw)

}

#### Evaluating train-test datasets

> table(evTable.train$type)

BLANK t.0x10d35 t.0x10daa t.0x10f03 t.0x210027 t.0x220001 t.0x3b70012

661 2 1 1 0 14 316

t.0x3b70014 t.0x3b70019 t.0xc40003

2 4 0

> table(evTable.test$type)

BLANK t.0x10d35 t.0x10daa t.0x10f03 t.0x210027 t.0x220001 t.0x3b70012

0 1 0 2 2 59 1240

t.0x3b70014 t.0x3b70019 t.0xc40003

872 0 1

Issue: Many event types (including the largest one, BLANK) have no presence in either the train dataset or the test dataset.

Note: Add to model label

* Train samples
* Test samples

> llply.n(winEvTable.train,2,function(df){summary(df[,-1])})

$back

$back$b.5m

BLANK t.0x10d35 t.0x10daa t.0x10f03 t.0x210027

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:278 FALSE:524 FALSE:528 FALSE:530 FALSE:531

TRUE :253 TRUE :7 TRUE :3 TRUE :1 NA's :0

NA's :0 NA's :0 NA's :0 NA's :0

t.0x220001 t.0x3b70012 t.0x3b70014 t.0x3b70019 t.0xc40003

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:521 FALSE:268 FALSE:530 FALSE:518 FALSE:531

TRUE :10 TRUE :263 TRUE :1 TRUE :13 NA's :0

NA's :0 NA's :0 NA's :0 NA's :0

$back$b.30m

BLANK t.0x10d35 t.0x10daa t.0x10f03 t.0x210027

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:278 FALSE:523 FALSE:527 FALSE:530 FALSE:531

TRUE :253 TRUE :8 TRUE :4 TRUE :1 NA's :0

NA's :0 NA's :0 NA's :0 NA's :0

t.0x220001 t.0x3b70012 t.0x3b70014 t.0x3b70019 t.0xc40003

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:516 FALSE:262 FALSE:530 FALSE:518 FALSE:531

TRUE :15 TRUE :269 TRUE :1 TRUE :13 NA's :0

NA's :0 NA's :0 NA's :0 NA's :0

$fw

$fw$f.5m

NO.EVENTS BLANK t.0x10d35 t.0x10daa t.0x10f03

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:224 FALSE:428 FALSE:525 FALSE:529 FALSE:531

TRUE :307 TRUE :103 TRUE :6 TRUE :2 NA's :0

NA's :0 NA's :0 NA's :0 NA's :0

t.0x210027 t.0x220001 t.0x3b70012 t.0x3b70014 t.0x3b70019

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:531 FALSE:527 FALSE:417 FALSE:530 FALSE:528

NA's :0 TRUE :4 TRUE :114 TRUE :1 TRUE :3

NA's :0 NA's :0 NA's :0 NA's :0

t.0xc40003

Mode :logical

FALSE:531

NA's :0

$fw$f.30m

NO.EVENTS BLANK t.0x10d35 t.0x10f03 t.0x210027

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:187 FALSE:417 FALSE:529 FALSE:531 FALSE:531

TRUE :344 TRUE :114 TRUE :2 NA's :0 NA's :0

NA's :0 NA's :0 NA's :0

t.0x220001 t.0x3b70012 t.0x3b70014 t.0x3b70019 t.0xc40003

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:509 FALSE:477 FALSE:531 FALSE:523 FALSE:531

TRUE :22 TRUE :54 NA's :0 TRUE :8 NA's :0

NA's :0 NA's :0 NA's :0

> llply.n(winEvTable.test ,2,function(df){summary(df[,-1])})

$back

$back$b.5m

BLANK t.0x10d35 t.0x10daa t.0x10f03 t.0x210027

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:1297 FALSE:1292 FALSE:1297 FALSE:1295 FALSE:1290

NA's :0 TRUE :5 NA's :0 TRUE :2 TRUE :7

NA's :0 NA's :0 NA's :0

t.0x220001 t.0x3b70012 t.0x3b70014 t.0x3b70019 t.0xc40003

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:1251 FALSE:40 FALSE:342 FALSE:1297 FALSE:1290

TRUE :46 TRUE :1257 TRUE :955 NA's :0 TRUE :7

NA's :0 NA's :0 NA's :0 NA's :0

$back$b.30m

BLANK t.0x10d35 t.0x10daa t.0x10f03 t.0x210027

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:1297 FALSE:1290 FALSE:1297 FALSE:1294 FALSE:1289

NA's :0 TRUE :7 NA's :0 TRUE :3 TRUE :8

NA's :0 NA's :0 NA's :0

t.0x220001 t.0x3b70012 t.0x3b70014 t.0x3b70019 t.0xc40003

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:1245 FALSE:31 FALSE:304 FALSE:1297 FALSE:1289

TRUE :52 TRUE :1266 TRUE :993 NA's :0 TRUE :8

NA's :0 NA's :0 NA's :0 NA's :0

$fw

$fw$f.5m

NO.EVENTS BLANK t.0x10d35 t.0x10daa t.0x10f03

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:1053 FALSE:1297 FALSE:1292 FALSE:1297 FALSE:1288

TRUE :244 NA's :0 TRUE :5 NA's :0 TRUE :9

NA's :0 NA's :0 NA's :0

t.0x210027 t.0x220001 t.0x3b70012 t.0x3b70014 t.0x3b70019

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:1295 FALSE:1261 FALSE:321 FALSE:337 FALSE:1297

TRUE :2 TRUE :36 TRUE :976 TRUE :960 NA's :0

NA's :0 NA's :0 NA's :0 NA's :0

t.0xc40003

Mode :logical

FALSE:1296

TRUE :1

NA's :0

$fw$f.30m

NO.EVENTS BLANK t.0x10d35 t.0x10f03 t.0x210027

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:1099 FALSE:1297 FALSE:1288 FALSE:1278 FALSE:1296

TRUE :198 NA's :0 TRUE :9 TRUE :19 TRUE :1

NA's :0 NA's :0 NA's :0 NA's :0

t.0x220001 t.0x3b70012 t.0x3b70014 t.0x3b70019 t.0xc40003

Mode :logical Mode :logical Mode :logical Mode :logical Mode :logical

FALSE:1274 FALSE:213 FALSE:390 FALSE:1297 FALSE:1296

TRUE :23 TRUE :1084 TRUE :907 NA's :0 TRUE :1

NA's :0 NA's :0 NA's :0 NA's :0

#### Processing Issue: samples with no events in window

I want to trace these too

* Solution: on empty event windows, instead of returning empty data frame, return a row with (date, type=NO.EVENTS, evInWin=TRUE).
* Now we can keep track of events without precedent.

#### Processing Issue: performance differences

We have tweaked the function to work better, but we have not identified the reason behind the perfomance differences.

* Current version (¬ 6s to process 3k events)

getEventsInWindow<-function(index,.data,windowSize=300,direction="forward")

{

windowOrigin<-.data$date[index]

# filter out current event

# filter: events before or after window origin

#TODO: check if need to remove = from >=

if(direction=="forward")

{.data.filtered<-.data %.% filter (date>windowOrigin & date <= windowOrigin+windowSize)}else

if(direction=="backward")

{.data.filtered<-.data %.% filter (date<windowOrigin & date >= windowOrigin-windowSize)}else

if(direction=="both")

{.data.filtered<-.data %.% filter (date<=windowOrigin+windowSize & date >= windowOrigin-windowSize)}else

{return(data.frame())}

# if empty window, return empty row

if(nrow(.data.filtered)==0) {return(data.frame())}

#if (countEvents) result <- .data.filtered %.% group\_by(type) %.% summarise(minDist1=n())

result<-.data.filtered %.% group\_by(type) %.% summarise(evInWin=n()>0) %.% mutate(date=windowOrigin) %.% select(date,type,evInWin)

}

* Previous version (~30s to process 3k events)

getEventsInWindow<-function(index,.data,windowSize=300,direction="forward")

{

windowOrigin<-.data$date[index]

# filter out current event

.data.filtered<-.data[-index,]

# filter: events before or after window origin

#TODO: check if need to remove = from >=

if(direction=="forward"){.data.filtered<-.data.filtered %.% filter (date>=windowOrigin)

}else if(direction=="backward"){.data.filtered<-.data.filtered %.% filter (date<=windowOrigin) }

# if empty window, return empty table - dplyr crashes with empty tables

if(nrow(.data.filtered)==0) {return(data.frame())}

#filter: events within window

.data.filtered<-.data.filtered %.% filter(abs(windowOrigin%--%date)<windowSize)

# if empty window, return empty row

if(nrow(.data.filtered)==0) {return(data.frame())}

#if (countEvents) result <- .data.filtered %.% group\_by(type) %.% summarise(minDist1=n())

result<-.data.filtered %.% group\_by(type) %.% summarise(evInWin=n()>0) %.% mutate(date=windowOrigin) %.% select(date,type,evInWin)

}

It appears that generating lubridate::interval's before filtering is very inefficient. Also, this version performs two filtering steps, rather than one.