Using the *TRONCO* package

Marco Antoniotti* Giulio Caravagna* Luca De Sano* Alex Graudenzi*
Giancarlo Mauri* Bud Mishra† Daniele Ramazzotti*

September 22, 2016

Overview. The *TRONCO* (TR*anslational* ONCO*logy*) *R* package collects algorithms to infer progression models via the approach of Suppes-Bayes Causal Network, both from an ensemble of tumors (cross-sectional samples) and within an individual patient (multi-region or single-cell samples). The package provides parallel implementation of algorithms that process binary matrices where each row represents a tumor sample and each column a single-nucleotide or a structural variant driving the progression; a 0/1 value models the absence/presence of that alteration in the sample. The tool can import data from plain, MAF or GISTIC format files, and can fetch it from the cBioPortal for cancer genomics. Functions for data manipulation and visualization are provided, as well as functions to import/export such data to other bioinformatics tools for, e.g, clustering or detection of mutually exclusive alterations. Inferred models can be visualized and tested for their confidence via bootstrap and cross-validation. *TRONCO* is used for the implementation of the Pipeline for Cancer Inference.

In this vignette, we will give an overview of the package by presenting some of the functions that could be most commonly used to arrange a data-analysis pipeline, along with their parameters to customize TRONCO's functioning. Advanced example case studies are available at the tool webpage

Version.	2.4.2 (June 2016)
Contact.	tronco@disco.unimib.it
Bugs report.	https://github.com/BIMIB-DISCo/TRONCO
Website.	https://sites.google.com/site/troncopackage

*Dipartimento di Informatica Sistemistica e Comunicazione, Università degli Studi Milano Bicocca Milano, Italy.
†Courant Institute of Mathematical Sciences, New York University, New York, USA.

Contents

1 Changelog

2.4.2 Latest development version. Implements a noise model and finalizes a series of algorithms reconstructing Suppes-Bayes Causal Network as maximum spanning trees.

2.4 Current stable version. New statistics available for model confidence via cross-validation routines. New algorithms based on Minimum Spanning Tree extraction.

2.0 Released in summer 2015 on our GitHUB, replaced the *Bioconductor* version in autumn 2015. This version is parallel, includes also the CAPRI algorithm, supports common GISTIC and MAF input formats, supports TCGA samples editing and queries to the cBio portal. This version has new plotting capabilities, and a general from-scratch design. It is not compatible with previous releases.

1.0 released in mid 2014, includes CAPRESE algorithm. It is now outdated and no more maintained;

2 Algorithms and useful links

Acronym	**Extended name**	**App.**	**Reference**
CAPRESE	Cancer Progression Extraction with Single Edges	Ind	PLoS ONE, 9(10):e108358, 2014.
CAPRI	Cancer Progression Inference	Ens	Bioinformatics 31(18), 3016-3016, 2015.
MST (Edmond)	Directed Minimum Spanning Tree with Mutual Information	Ind	In preparation.
MST (Gabow)	Partially Directed Minimum Spanning Tree with Mutual Information	Ind	In preparation.
MST (Chow Liu)	Undirected Minimum Spanning Tree with Likelihood-Fit	Ind	In preparation.
MST (Prim)	Undirected Minimum Spanning Tree with Mutual Information	Ind	In preparation.

Legend. Ens.: ensemble-level with cross-sectional data; Ind.: individual-level with single-cell or multi-region data.

External links to resources related to TRONCO.

- TRONCO was introduced in Bioinformatics. 2016 Feb 9. pii: btw035.
- TRONCO since version **2.3** is used to implement the **Pipeline For Cancer Inference (PiCnIc)** described in Caravagna et al., 2016, under review.
- Case studies featuring Atypical Chronic Myeloid Leukemia, Colorectal Cancer, Clear Cell Renal Cell Carcinoma and others are available at the tool webpage. Code for replication of each of those study is made available through Bioinformatics Milano-Bicocca's Github.

3 Loading data

```
library(TRONCO)
data(aCML)
data(crc_maf)
```

```
data(crc_gistic)
data(crc_plain)
```

Preliminaries. TRONCO transforms input data in a sort of database-alike format, where three main fields are presente: `genotypes` which contains the genomic signatures of the input samples, `annotations` which provides an index to the events present in the data and `types`, a field mapping type of events (e.g., mutations, CNAs, etc.) to colors for display visualization. Other annotations are generated when a dataset is augmented with some metadata. A TRONCO object shall be edited by using TRONCO functions, to avoid to create inconsistencies in its internal representation. Function `is.compliant` can be used to test if a TRONCO object is consistent; the function is called by any TRONCO function before returning a modified object, so to ensure that consistency is preserved – `is.compliant` will raise an error if this is not the case.

TRONCO supports the import of data from 3 formats. The Mutation Annotation Format (*MAF*) is a tab-delimited file containing somatic and/or germline mutation annotations; the *GISTIC* format for copy number alterations as defined by TCGA and a custom boolean matrix format where the user can directly specify the mutational profiles to be importend. Through some data included in the package we will show how to load your datasets in TRONCO.

`aCML` a TRONCO object that represents the *atypical Chronic Myeloid Leukemia* dataset by Piazza *et al.* (Nat. Gen. 2013 45(1):18-24).

`crc_maf` a shortened version of the *colorectal cancer mutation data* made available by the TCGA consortium within the COADREAD project[1]

`crc_gistic` from the same TCGA project, we also provide a shortened version of the focal CNAs in the GISTIC format where 1 represents a low level gain, 2 a high level gain, -1 a heterozygous loss of a gene and -2 its homozygous loss.

`crc_plain` a custom boolean matrix where rows are samples, and columns represent events – in this case alterations in a certain gene. Notice with this format one could also custom types of alterations, for instance wider chromosomal aberrations or, in principle, epigenetic states (over-expression, methylated regions, etc.) that are persistent across tumor evolution.

Whatever is dataset created as explained in the next sections, it can be annotated by adding a mnemonic description of the data, which will be used as plot titles when possible. Function `annotate.description` raises a warning if the dataset was previously annotated.

aCML = annotate.description(aCML, 'aCML data (Bioinf.)')

3.1 Mutations annotated in a MAF format

We use the function `import.MAF` to import a dataset in MAF format, in this case the following TCGA dataset

```
head(crc_maf[, 1:10])
```

```
##      Hugo_Symbol Entrez_Gene_Id       Center NCBI_Build Chromosome Start_position
## 27          TP53           7157 hgsc.bcm.edu         36         17        7519131
## 246       FBXW7          55294 hgsc.bcm.edu         36          4      153466817
## 623         APC            324 hgsc.bcm.edu         36          5      112192485
## 649         TP53           7157 hgsc.bcm.edu         36         17        7518937
## 928       FBXW7          55294 hgsc.bcm.edu         36          4      153468834
## 1390        TP53           7157 hgsc.bcm.edu         36         17        7517864
##      End_position Strand Variant_Classification Variant_Type
## 27        7519131      +      Missense_Mutation          SNP
## 246     153466817      +      Nonsense_Mutation          SNP
## 623     112192485      +      Nonsense_Mutation          SNP
## 649       7518937      +      Nonsense_Mutation          SNP
## 928     153468834      +      Missense_Mutation          SNP
## 1390      7517864      +      Missense_Mutation          SNP
```

A default importation is done without adding parameters to `import.MAF`. In this case, all mutations per gene will be considered equivalent, regardless of the type that is annotated in the MAF. Also, all genes will be imported, and all

[1] See https://tcga-data.nci.nih.gov/docs/publications/coadread_2012/ and our PicNiC case study (§**??**) for the real analysis of such data.

samples.

```
dataset_maf = import.MAF(crc_maf)

## *** Importing from dataframe
## Loading MAF dataframe ...DONE
## *** Mutations names: using Hugo_Symbol
## *** Using full MAF: #entries  17
## *** MAF report: TCGA=TRUE
## Type of annotated mutations:
## [1] "Missense_Mutation" "Nonsense_Mutation"
## *** [merge.mutation.types = T] Mutations will be merged and annotated as 'Mutation'
## Number of samples: 9
## [TCGA = TRUE] Number of TCGA patients: 9
## Number of annotated mutations: 17
## Mutations annotated with "Valid" flag (%): 71
## Number of genes (Hugo_Symbol): 6
## Starting conversion from MAF to 0/1 mutation profiles (1 = mutation) :9 x 6
## .................
## Starting conversion from MAF to TRONCO data type.
```

See §**??** to understand how to visualize a TRONCO dataset. In the above case – where we see that mutations are annotated as `Missense_Mutation` or `Nonsense_Mutation`, if a gene in a sample has both, these will be merged to a unique `Mutation` type. In this case a pair gene name with `Mutation` will be what we call an "event" in our dataset – e.g., APC `Mutation`.

If one would like to have two distinct events in the dataset, i.e., APC `Missense_Mutation` and APC `Nonsense_Mutation`, parameter `merge.mutation.types` should be set to false in the call to `import.MAF`.

```
dataset_maf = import.MAF(crc_maf, merge.mutation.types = FALSE)

## *** Importing from dataframe
## Loading MAF dataframe ...DONE
## *** Mutations names: using Hugo_Symbol
## *** Using full MAF: #entries  17
## *** MAF report: TCGA=TRUE
## Type of annotated mutations:
## [1] "Missense_Mutation" "Nonsense_Mutation"
## *** [merge.mutation.types = F] Mutations will be distinguished by type
## Number of samples: 9
## [TCGA = TRUE] Number of TCGA patients: 9
## Number of annotated mutations: 17
## Mutations annotated with "Valid" flag (%): 71
## Number of genes (Hugo_Symbol): 6
## Starting conversion from MAF to 0/1 mutation profiles (1 = mutation) :
## .................
```

Sometimes, we might want to filter out some of the entries in a MAF – maybe restricting the type of genes, mutations or sample that we want to process. If one defines `filter.fun` as a function that returns `TRUE` only for those entries which shall be considered, he gets a filter process which is applied to each row of the MAF file prior to transforming that into a TRONCO dataset. In this example we select only mutations annotated to APC – we access that through the `Hugo_Symbol` flag of a MAF.

```
dataset_maf = import.MAF(crc_maf, filter.fun = function(x){ x['Hugo_Symbol'] == 'APC'} )

## *** Importing from dataframe
## Loading MAF dataframe ...DONE
## *** Mutations names: using Hugo_Symbol
## *** Filtering full MAF: #entries  17
## *** Using reduced MAF: #entries  3
## *** MAF report: TCGA=TRUE
## Type of annotated mutations:
```

```
## [1] "Nonsense_Mutation"
## *** [merge.mutation.types = T] Mutations will be merged and annotated as 'Mutation'
## Number of samples: 3
## [TCGA = TRUE] Number of TCGA patients: 3
## Number of annotated mutations: 3
## Mutations annotated with "Valid" flag (%): 33
## Number of genes (Hugo_Symbol): 1
## Starting conversion from MAF to 0/1 mutation profiles (1 = mutation) :3 x 1
## ...
## Starting conversion from MAF to TRONCO data type.
```

It is also sometimes convenient – especially when working with data collected from a single individual patient – to distinguish the type of mutations and their position in a gene, or if they are somehow annotated to COSMIC or other databases. For instance, we might want to want to use the `MA.protein.change` annotation in the MAF file to get composite names such as TP53.R175H, TP53.R213, TP53.R267W etc. This can be done by setting `paste.to.Hugo_Symbol` to have the relevant name of the MAF annotation

```
dataset_maf = import.MAF(crc_maf,
    merge.mutation.types = FALSE,
    paste.to.Hugo_Symbol = c('MA.protein.change'))
```

```
## *** Importing from dataframe
## Loading MAF dataframe ...DONE
## *** Mutations names: augmenting Hugo_Symbol with values:  MA.protein.change
## *** Using full MAF: #entries  17
## *** MAF report: TCGA=TRUE
## Type of annotated mutations:
## [1] "Missense_Mutation" "Nonsense_Mutation"
## *** [merge.mutation.types = F] Mutations will be distinguished by type
## Number of samples: 9
## [TCGA = TRUE] Number of TCGA patients: 9
## Number of annotated mutations: 17
## Mutations annotated with "Valid" flag (%): 71
## Number of genes (Hugo_Symbol): 16
## Starting conversion from MAF to 0/1 mutation profiles (1 = mutation) :
## .................
```

TRONCO supports custom MAF files, where possibly not all the standard annotations are present, via `irregular = TRUE`.

3.2 Copy Number Variants annotated in the GISTIC format

We use the function `import.GISTIC` to import a dataset in GISTIC format, in this case from

```
crc_gistic
```

```
##              NRAS CTNNB1 FBXW7 APC KRAS TP53
## TCGA-A6-2670   -1      0     0  -1    1   -1
## TCGA-A6-2672    0      0     0   0    0    0
## TCGA-A6-2674    0      0     0   0    0    0
## TCGA-A6-2676    0      0     0   0    0    0
## TCGA-A6-2677    0      0     0   0    0   -1
## TCGA-A6-2678    0      0     0   0    0   -1
## TCGA-A6-2683   -1     -1    -1  -1    0   -1
## TCGA-A6-3807    0     -1     0   0   -1   -1
## TCGA-AA-3516    0      0     0   0    0    0
```

In its default execution all the data annotated in the file is imported. But in principle it is possible to avoid to import some genes or samples; in this case it is sufficient to use parameters `filter.genes` and `filter.samples` for this function.

```
dataset_gistic = import.GISTIC(crc_gistic)
```

```
## *** Using full GISTIC: #dim  9  x  6
## *** GISTIC input format conversion started.
## Converting input data to character for import speedup.
## Creating  24 events for 6 genes
## Extracting "Homozygous Loss" events (GISTIC = -2)
## Extracting "Heterozygous Loss" events (GISTIC = -1)
## Extracting "Low-level Gain" events (GISTIC = +1)
## Extracting "High-level Gain" events (GISTIC = +2)
## Transforming events in TRONCO data types .....
## *** Binding events for 4 datasets.
## *** Data extracted, returning only events observed in at least one sample
##  Number of events: n = 7
##  Number of genes: |G| = 6
##  Number of samples: m = 9
```

3.3 Custom alterations annotated in a boolean matrix

One can annotate its custom type of alterations in a boolean matrix such as `crc_plain`

```
crc_plain
```

```
##                 TP53 FBXW7 APC CTNNB1 NRAS KRAS
## TCGA-AA-3517-01    1     0   0      0    0    0
## TCGA-AA-3518-01    0     1   0      0    0    0
## TCGA-AA-3519-01    1     0   1      0    0    0
## TCGA-AA-3520-01    1     0   0      0    0    1
## TCGA-AA-3521-01    0     0   1      0    0    1
```

In this case, function `import.genotypes` will convert the matrix to a TRONCO object where events' names and samples codes will be set from column and row names of the matrix. If this is not possible, these will be generated from templates. By default, the `event.type` is set to `variant` but one can specify a custom name for the alteration that is reported in the matrix

```
dataset_plain = import.genotypes(crc_plain, event.type='myVariant')
```

3.4 Downloading data from the cBio portal for cancer genomics

TRONCO uses the R interface to cBio to query data from the portal. All type of data can be downloaded from the portal, which includes MAF/GISTIC data for a lot of different cancer studies. An example of interaction with the portal is archived at the tool's webpage.

Here, we show how to download lung cancer data somatic mutations for genes TP53, KRAS and PIK3CA, from the lung cancer project run by TCGA, which is archived as `luad_tcga_pub` at cBio. If some of the parameters to `cbio.query` are missing the function will become interactive by showing a list of possible data available at the portal.

```
data = cbio.query(
    genes=c('TP53', 'KRAS', 'PIK3CA'),
    cbio.study = 'luad_tcga_pub',
    cbio.dataset = 'luad_tcga_pub_cnaseq',
    cbio.profile = 'luad_tcga_pub_mutations')
```

4 Data visualisation

All examples in this section will be done with the the aCML dataset as reference.

4.1 Summary report for a dataset and boolean queries

We use the function `view` to get a short summary of a dataset that we loaded in TRONCO; this function reports on the number of samples and events, plus some meta information that could be displayed graphically.

```
view(aCML)
```

```
## Description: CAPRI - Bionformatics aCML data.
## -- TRONCO Dataset: n=64, m=31, |G|=23, patterns=0.
## Events (types): Ins/Del, Missense point, Nonsense Ins/Del, Nonsense point.
## Colors (plot): #7FC97F, #4483B0, #FDC086, #fab3d8.
## Events (5 shown):
##   gene 4 : Ins/Del TET2
##   gene 5 : Ins/Del EZH2
##   gene 6 : Ins/Del CBL
##   gene 7 : Ins/Del ASXL1
##   gene 29 : Missense point SETBP1
## Genotypes (5 shown):
```

4.2 Creating views with the "as" functions

Several functions are available to create views over a dataset, with a set of parameter which can constraint the view – as in the SELECT/JOIN approaches in databases. In the following examples we show their execution with the default parameters, but shorten their output to make this document readable.

The main "as" functions are here documented. `as.genotypes`, that we can use to get the matrix of "genotypes" that we imported.

```
as.genotypes(aCML)[1:10,5:10]
```

```
##            gene 29 gene 30 gene 31 gene 32 gene 33 gene 34
## patient 1        1       0       0       0       0       0
## patient 2        1       0       0       0       0       1
## patient 3        1       1       0       0       0       0
## patient 4        1       0       0       0       0       1
## patient 5        1       0       0       0       0       0
## patient 6        1       0       0       0       0       0
## patient 7        1       0       0       0       0       0
## patient 8        1       0       0       0       0       0
## patient 9        1       0       0       0       0       0
## patient 10       1       0       0       0       0       0
```

Differently, `as.events` and `as.events.in.samples`, that show tables with the events that we are processing in all dataset or in a specific sample that we want to examine.

```
as.events(aCML)[1:5, ]
```

```
##         type             event
## gene 4  "Ins/Del"        "TET2"
## gene 5  "Ins/Del"        "EZH2"
## gene 6  "Ins/Del"        "CBL"
## gene 7  "Ins/Del"        "ASXL1"
## gene 29 "Missense point" "SETBP1"
```

```
as.events.in.sample(aCML, sample = 'patient 2')
```

```
##         type             event
## gene 29 "Missense point" "SETBP1"
## gene 34 "Missense point" "CBL"
## gene 91 "Nonsense point" "ASXL1"
```

Concerning genes, `as.genes` shows the mnemonic names of the genes (or chromosomes, cytobands, etc.) that we included in our dataset.

```
as.genes(aCML)[1:8]
```

```
## [1] "TET2"   "EZH2"   "CBL"    "ASXL1"  "SETBP1" "NRAS"   "KRAS"   "IDH2"
```

And `as.types` shows the types of alterations (e.g., mutations, amplifications, etc.) that we have find in our dataset, and function `as.colors` shows the list of the colors which are associated to each type.

```
as.types(aCML)
```

```
## [1] "Ins/Del"          "Missense point"   "Nonsense Ins/Del" "Nonsense point"
```

```
as.colors(aCML)
```

```
##          Ins/Del   Missense point Nonsense Ins/Del   Nonsense point
##        "#7FC97F"        "#4483B0"        "#FDC086"        "#fab3d8"
```

A function `as.gene` can be used to display the alterations of a specific gene across the samples

```
head(as.gene(aCML, genes='SETBP1'))
```

```
##           Missense point SETBP1
## patient 1                      1
## patient 2                      1
## patient 3                      1
## patient 4                      1
## patient 5                      1
## patient 6                      1
```

Views over samples can be created as well. `as.samples` and `which.samples` list all the samples in the data, or return a list of samples that harbour a certain alteration. The former is

```
as.samples(aCML)[1:10]
```

```
##  [1] "patient 1"  "patient 2"  "patient 3"  "patient 4"  "patient 5"  "patient 6"
##  [7] "patient 7"  "patient 8"  "patient 9"  "patient 10"
```

and the latter is

```
which.samples(aCML, gene='TET2', type='Nonsense point')
```

```
## [1] "patient 12" "patient 13" "patient 26" "patient 29" "patient 40" "patient 57"
## [7] "patient 62"
```

A slightly different function, which manipulates the data, is `as.alterations`, which transforms a dataset with events of different type to events of a unique type, labeled "Alteration".

```
dataset = as.alterations(aCML)
```

```
## *** Aggregating events of type(s) { Ins/Del, Missense point, Nonsense Ins/Del, Nonsense point }
## in a unique event with label " Alteration ".
## Dropping event types Ins/Del, Missense point, Nonsense Ins/Del, Nonsense point for 23 genes.
## ......................
## *** Binding events for 2 datasets.
```

```
view(dataset)
```

```
## -- TRONCO Dataset: n=64, m=23, |G|=23, patterns=0.
## Events (types): Alteration.
## Colors (plot): khaki.
## Events (5 shown):
##   G1 : Alteration TET2
##   G2 : Alteration EZH2
##   G3 : Alteration CBL
##   G4 : Alteration ASXL1
```

```
##   G5 : Alteration SETBP1
## Genotypes (5 shown):
```

When samples are enriched with stage information function `as.stages` can be used to create a view over such table. Views over patterns can be created as well – see Model Inference with CAPRI.

4.3 Dataset size

A set of functions allow to get the number of genes, events, samples, types and patterns in a dataset.

```
ngenes(aCML)
## [1] 23
nevents(aCML)
## [1] 31
nsamples(aCML)
## [1] 64
ntypes(aCML)
## [1] 4
npatterns(aCML)
## [1] 0
```

4.4 Oncoprints

Oncoprints are the most effective data-visualization functions in TRONCO. These are heatmaps where rows represent variants, and columns samples (*the reverse* of the input format required by TRONCO), and are annotated and displayed/sorted to enhance which samples have which mutations etc.

By default `oncoprint` will try to sort samples and events to enhance exclusivity patterns among the events.

```
oncoprint(aCML)
## *** Oncoprint for "CAPRI - Bionformatics aCML data"
## with attributes: stage = FALSE, hits = TRUE
## Sorting samples ordering to enhance exclusivity patterns.
## Setting automatic row font (exponential scaling): 8.1
```

But the sorting mechanism is bypassed if one wants to cluster samples or events, or if one wants to split samples by cluster (not shown). In the clustering case, the ordering is given by the dendrograms. In this case we also show the annotation of some groups of events via parameter `gene.annot`.

```
oncoprint(aCML,
    legend = FALSE,
    samples.cluster = TRUE,
    gene.annot = list(one = list('NRAS', 'SETBP1'), two = list('EZH2', 'TET2')),
    gene.annot.color = 'Set2',
    genes.cluster = TRUE)
```

Oncoprints can be annotated; a special type of annotation is given by stage data. As this is not available for the aCML dataset, we create it randomly, just for the sake of showing how the oncoprint is enriched with this information. This is the random stage map that we create – if some samples had no stage a NA would be added automatically.

```
stages = c(rep('stage 1', 32), rep('stage 2', 32))
stages = as.matrix(stages)
rownames(stages) = as.samples(aCML)
dataset = annotate.stages(aCML, stages = stages)
```

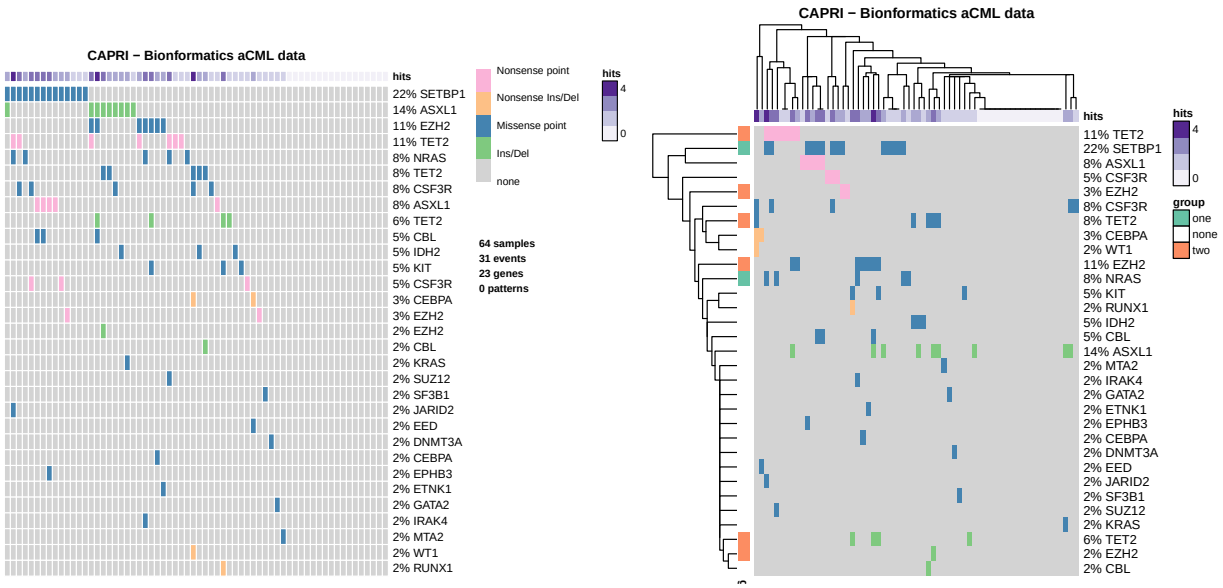

Figure 1: Two different calls to `oncoprint` with aCML data. This plot gives a graphical visualization of the events that are in the dataset – with a color per event type – but in left it sorts samples to enhance exclusivity patterns among the events, while in right it clusters samples/events.

```
has.stages(aCML)
```

```
## [1] FALSE
```

```
head(as.stages(dataset))
```

```
##             stage
## patient 1 stage 1
## patient 2 stage 1
## patient 3 stage 1
## patient 4 stage 1
## patient 5 stage 1
## patient 6 stage 1
```

The `as.stages` function can now be used to create a view over stages.

```
head(as.stages(dataset))
```

```
##             stage
## patient 1 stage 1
## patient 2 stage 1
## patient 3 stage 1
## patient 4 stage 1
## patient 5 stage 1
## patient 6 stage 1
```

After that the data is annotated via `annotate.stages` function, we can again plot an oncoprint – which this time will detect that the dataset has also stages associated, and will diplay those

```
oncoprint(dataset, legend = FALSE)
```

If one is willing to display samples grouped according to some variable, for instance after a sample clustering task, he can use `group.samples` parameter of `oncoprint` and that will override the mutual exclusivity ordering. Here, we make the trick of using the stages as if they were such clustering result.

```
oncoprint(dataset, group.samples = as.stages(dataset))
```

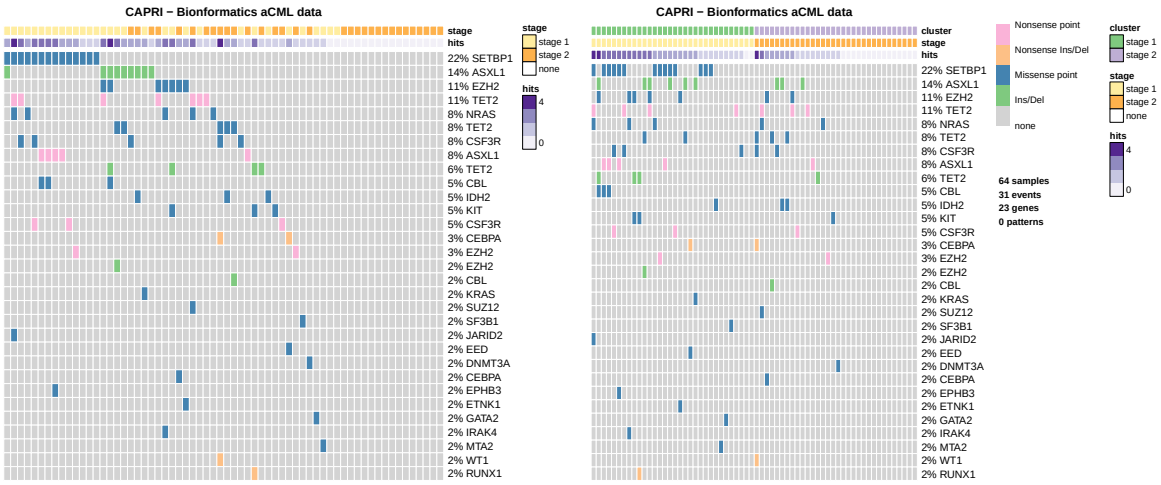

Figure 2: Example `oncoprint` output for aCML data with randomly annotated stages, in left, and samples clustered by group assignment in right – for simplicity the group variable is again the stage annotation.

4.5 Groups visualization (e.g., pathways)

TRONCO provides functions to visualize groups of events, which in this case are called pathways – though this could be any group that one would like to define. Aggregation happens with the same rational as the `as.alterations` function, namely by merging the events in the group.

We make an example of a pathway called `MyPATHWAY` involving genes SETBP1, EZH2 and WT1; we want it to be colored in red, and we want to have the genotype of each event to be maintened in the dataset. We proceed as follows (R's output is omitted).

```
pathway = as.pathway(aCML,
    pathway.genes = c('SETBP1', 'EZH2', 'WT1'),
    pathway.name = 'MyPATHWAY',
    pathway.color = 'red',
    aggregate.pathway = FALSE)
```

```
## *** Extracting events for pathway:  MyPATHWAY .
## *** Events selection: #events =  31 , #types =  4 Filters freq|in|out = { FALSE ,  TRUE ,  FALSE }
## [filter.in] Genes hold:  SETBP1, EZH2, WT1  ...  [ 3 / 3  found].
## Selected  5  events, returning.
## Pathway extracted succesfully.
## *** Binding events for 2 datasets.
```

Which we then visualize with an oncoprint

```
oncoprint(pathway, title = 'Custom pathway',  font.row = 8, cellheight = 15, cellwidth = 4)
```

In TRONCO there is also a function which creates the pathway view and the corresponding oncoprint to multiple pathways, when these are given as a list. We make here a simple example of two custom pathways.

```
pathway.visualization(aCML,
    pathways=list(P1 = c('TET2', 'IRAK4'),  P2=c('SETBP1', 'KIT')),
    aggregate.pathways=FALSE,
    font.row = 8)
```

```
## Annotating pathways with RColorBrewer color palette Set2 .
## *** Processing pathways: P1, P2
##
## [PATHWAY "P1"] TET2, IRAK4
## *** Extracting events for pathway:  P1 .
## *** Events selection: #events =  31 , #types =  4 Filters freq|in|out = { FALSE ,  TRUE ,  FALSE }
```

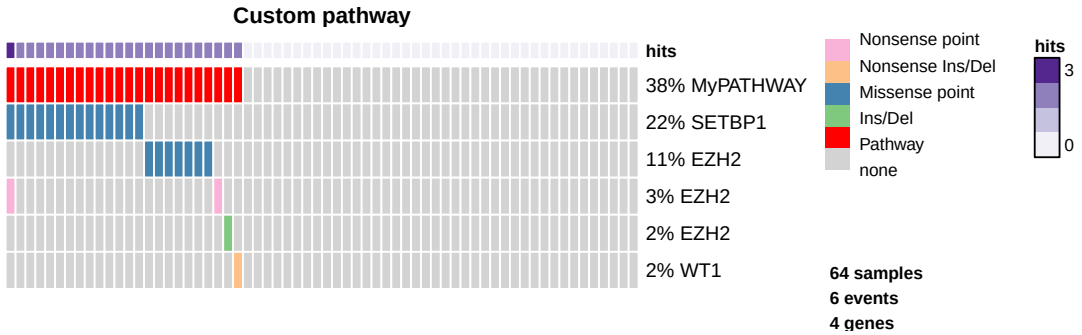

Figure 3: `oncoprint` output of a custom pathway called `MyPATHWAY` involving genes SETBP1, EZH2 and WT1; the genotype of each event is shown.

```
## [filter.in] Genes hold:  TET2, IRAK4  ...  [ 2 / 2  found].
## Selected  4  events, returning.
## Pathway extracted succesfully.
## *** Binding events for 2 datasets.
##
##
## [PATHWAY "P2"] SETBP1, KIT
## *** Extracting events for pathway:  P2 .
## *** Events selection: #events =  31 , #types =  4 Filters freq|in|out = { FALSE ,  TRUE ,  FALSE }
## [filter.in] Genes hold:  SETBP1, KIT  ...  [ 2 / 2  found].
## Selected  2  events, returning.
## Pathway extracted succesfully.
## *** Binding events for 2 datasets.
## *** Binding events for 2 datasets.
## *** Oncoprint for "Pathways: P1, P2"
## with attributes: stage = FALSE, hits = TRUE
## Sorting samples ordering to enhance exclusivity patterns.
## NULL
```

If we had to visualize just the signature of the pathway, we could set `aggregate.pathways=T`.

```
pathway.visualization(aCML,
    pathways=list(P1 = c('TET2', 'IRAK4'),  P2=c('SETBP1', 'KIT')),
    aggregate.pathways = TRUE,
    font.row = 8)
```

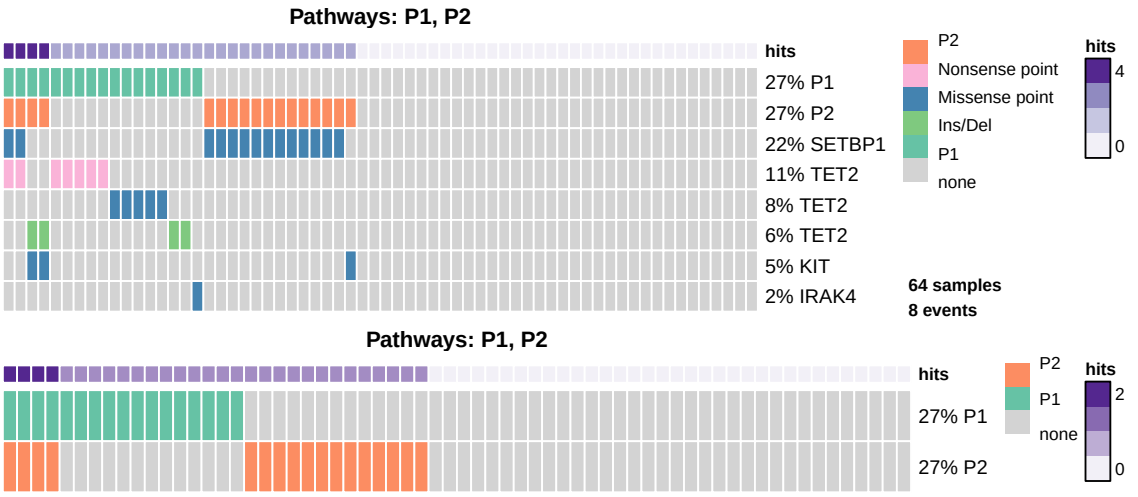

Figure 4: `oncoprint` output of a custom pair of pathways, with events shown in top and hidden in bottom.

5 Data manipulation

All examples in this section will be done with the the aCML dataset as reference.

5.1 Modifying events and samples

TRONCO provides functions for renaming the events that were included in a dataset, or the type associated to a set of events (e.g., a "Mutation" could be renamed to a "Missense Mutation").

```
dataset = rename.gene(aCML, 'TET2', 'new name')
dataset = rename.type(dataset, 'Ins/Del', 'new type')
as.events(dataset, type = 'new type')
```

```
##        type       event
## gene 4 "new type" "new name"
## gene 5 "new type" "EZH2"
## gene 6 "new type" "CBL"
## gene 7 "new type" "ASXL1"
```

and return a modified TRONCO object. More complex operations are also possible. For instance, two events with the same signature – i.e., appearing in the same samples – can be joined to a new event (see also Data Consolidation in Model Inference) with the same signature and a new name.

```
dataset = join.events(aCML,
    'gene 4',
    'gene 88',
    new.event='test',
    new.type='banana',
    event.color='yellow')
```

```
## *** Binding events for 2 datasets.
```

where in this case we also created a new event type, with its own color.

In a similar way we can decide to join all the events of two distinct types, in this case if a gene x has signatures for both type of events, he will get a unique signature with an alteration present if it is either of the second *or* the second type

```
dataset = join.types(dataset, 'Nonsense point', 'Nonsense Ins/Del')
```

```
## *** Aggregating events of type(s) { Nonsense point, Nonsense Ins/Del }
```

```
## in a unique event with label " new.type ".
## Dropping event types Nonsense point, Nonsense Ins/Del for 6 genes.
## ......
## *** Binding events for 2 datasets.
```

```
as.types(dataset)
```

```
## [1] "Ins/Del"        "Missense point" "banana"         "new.type"
```

TRONCO also provides functions for deleting specific events, samples or types.

```
dataset = delete.gene(aCML, gene = 'TET2')
dataset = delete.event(dataset, gene = 'ASXL1', type = 'Ins/Del')
dataset = delete.samples(dataset, samples = c('patient 5', 'patient 6'))
dataset = delete.type(dataset, type = 'Missense point')
view(dataset)
```

```
## Description: CAPRI - Bionformatics aCML data.
## -- TRONCO Dataset: n=62, m=8, |G|=7, patterns=0.
## Events (types): Ins/Del, Nonsense Ins/Del, Nonsense point.
## Colors (plot): #7FC97F, #FDC086, #fab3d8.
## Events (5 shown):
##   gene 5 : Ins/Del EZH2
##   gene 6 : Ins/Del CBL
##   gene 66 : Nonsense Ins/Del WT1
##   gene 69 : Nonsense Ins/Del RUNX1
##   gene 77 : Nonsense Ins/Del CEBPA
## Genotypes (5 shown):
```

5.2 Modifying patterns

TRONCO provides functions to edit patterns, pretty much as for any other type of events. Patterns however have a special denotation and are supported only by CAPRI algorithm – see Model Reconstruction with CAPRI to see a practical application of that.

5.3 Subsetting a dataset

It is very often the case that we want to subset a dataset by either selecting only some of its samples, or some of its events. Function `samples.selection` returns a dataset with only some selected samples.

```
dataset = samples.selection(aCML, samples = as.samples(aCML)[1:3])
view(dataset)
```

```
## Description: CAPRI - Bionformatics aCML data.
## -- TRONCO Dataset: n=3, m=31, |G|=23, patterns=0.
## Events (types): Ins/Del, Missense point, Nonsense Ins/Del, Nonsense point.
## Colors (plot): #7FC97F, #4483B0, #FDC086, #fab3d8.
## Events (5 shown):
##   gene 4 : Ins/Del TET2
##   gene 5 : Ins/Del EZH2
##   gene 6 : Ins/Del CBL
##   gene 7 : Ins/Del ASXL1
##   gene 29 : Missense point SETBP1
## Genotypes (5 shown):
```

Function `events.selection`, instead, performs selection according to a filter of events. With this function, we can subset data according to a frequency, and we can force inclusion/exclusion of certain events by specifying their name. For instance, here we pick all events with a minimum frequency of 5%, force exclusion of SETBP1 (all events associated), and inclusion of EZH1 and EZH2.

```
dataset = events.selection(aCML,  filter.freq = .05,
    filter.in.names = c('EZH1','EZH2'),
    filter.out.names = 'SETBP1')
```

```
## *** Events selection: #events =  31 , #types =  4 Filters freq|in|out = { TRUE ,  TRUE ,  TRUE }
## Minimum event frequency:  0.05  ( 3  alterations out of  64  samples).
## ..............................
## Selected  9  events.
##
## [filter.in] Genes hold:  EZH1, EZH2  ...  [ 1 / 2  found].
## [filter.out] Genes dropped:  SETBP1  ...  [ 1 / 1  found].
## Selected  10  events, returning.
```

```
as.events(dataset)
```

```
##         type             event
## gene 4  "Ins/Del"        "TET2"
## gene 5  "Ins/Del"        "EZH2"
## gene 7  "Ins/Del"        "ASXL1"
## gene 30 "Missense point" "NRAS"
## gene 32 "Missense point" "TET2"
## gene 33 "Missense point" "EZH2"
## gene 55 "Missense point" "CSF3R"
## gene 88 "Nonsense point" "TET2"
## gene 89 "Nonsense point" "EZH2"
## gene 91 "Nonsense point" "ASXL1"
```

An example visualization of the data before and after the selection process can be obtained by combining the `gtable` objects returned by `oncoprint`. We here use `gtable = T` to get access to have a GROB table returned, and `silent = T` to avoid that the calls to the function display on the device; the call to `grid.arrange` displays the captured `gtable` objects.

```
library(gridExtra)
grid.arrange(
    oncoprint(as.alterations(aCML, new.color = 'brown3'),
        cellheight = 6, cellwidth = 4, gtable = TRUE,
        silent = TRUE, font.row = 6)$gtable,
    oncoprint(dataset, cellheight = 6, cellwidth = 4,
        gtable = TRUE, silent = TRUE, font.row = 6)$gtable,
    ncol = 1)
```

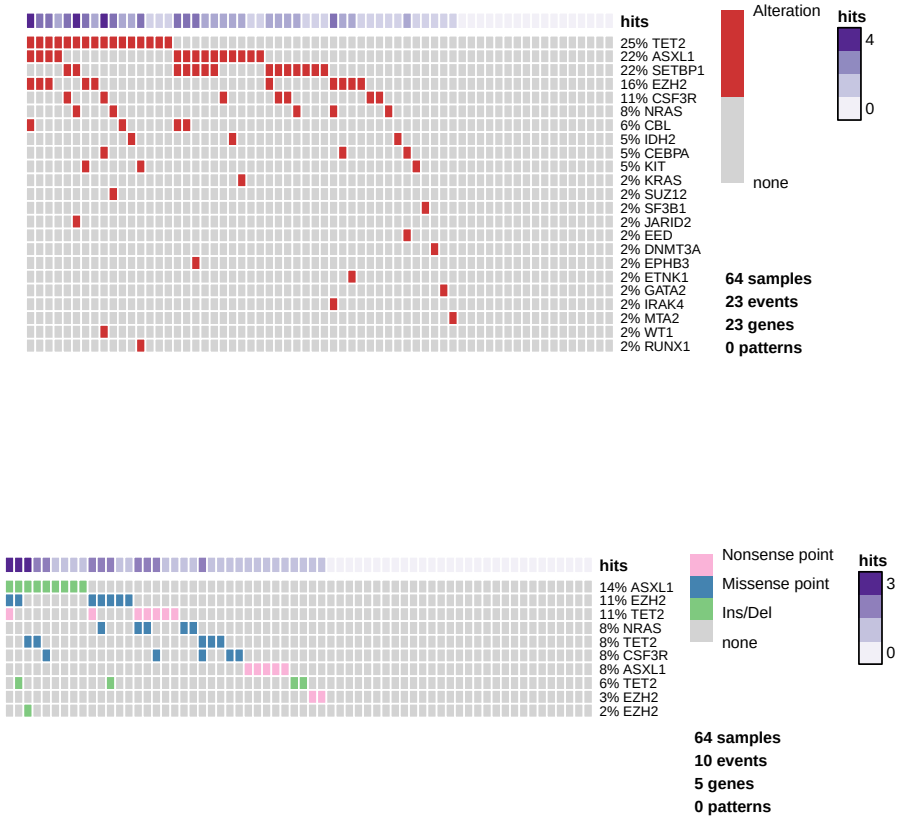

Figure 5: Multiple output from `oncoprint` can be captured as a `gtable` and composed via `grid.arrange` (package *gridExtra*). In this case we show aCML data on top – displayed after the `as.alterations` transformation – versus a selected subdataset of events with a minimum frequency of 5%, force exclusion of SETBP1 (all events associated), and inclusion of EZH1 and EZH2.

6 Model inference

We make use of the most of the functions described above to show how to perform inference with various algorithms; the reader should read first those sections of the vignette to have an explanation of how those functions work. The aCML dataset is used as a test-case for all algorithms, regardless it should be precessed by algorithms to infer ensemble-level progression models.

To replicate the plots of the original paper were the aCML dataset was first analyzed with CAPRI, we can change the colors assigned to each type of event with the function `change.color`.

```
dataset = change.color(aCML, 'Ins/Del', 'dodgerblue4')
dataset = change.color(dataset, 'Missense point', '#7FC97F')
as.colors(dataset)
```

```
##          Ins/Del   Missense point Nonsense Ins/Del   Nonsense point
##    "dodgerblue4"        "#7FC97F"        "#FDC086"        "#fab3d8"
```

Data consolidation. All TRONCO algorithms require an input dataset were events have non-zero/non-one probability, and are all distinguishable. The tool provides a function to return lists of events which do not satisfy these constraint.

```
consolidate.data(dataset)
```

```
## $indistinguishable
```

```
## list()
##
## $zeroes
## list()
##
## $ones
## list()
```

The aCML data has none of the above issues (the call returns empty lists); if this were not the case data manipulation functions can be used to edit a TRONCO object (§**??**).

6.1 CAPRI

In what follows, we show CAPRI's functioning by replicating the aCML case study presented in CAPRI's original paper. Regardless from which types of mutations we include, we select only the genes mutated at least in the 5% of the patients – thus we first use `as.alterations` to have gene-level frequencies, and then we apply there a frequency filter (R's output is omitted).

```
alterations = events.selection(as.alterations(aCML), filter.freq = .05)
```

```
## *** Aggregating events of type(s) { Ins/Del, Missense point, Nonsense Ins/Del, Nonsense point }
## in a unique event with label " Alteration ".
## Dropping event types Ins/Del, Missense point, Nonsense Ins/Del, Nonsense point for 23 genes.
## ......................
## *** Binding events for 2 datasets.
## *** Events selection: #events =  23 , #types =  1 Filters freq|in|out = { TRUE ,  FALSE ,  FALSE }
## Minimum event frequency:  0.05  ( 3  alterations out of  64  samples).
## ......................
## Selected  7  events.
##
## Selected  7  events, returning.
```

To proceed further with the example we create the `dataset` to be used for the inference of the model. From the original dataset we select all the genes whose mutations are occurring at least the 5% of the times, and we get that by the alterations profiles; also we force inclusion of all the events for the genes involved in an hypothesis (those included in variable `gene.hypotheses`, this list is based on the support found in the literature of potential aCML patterns).

```
gene.hypotheses = c('KRAS', 'NRAS', 'IDH1', 'IDH2', 'TET2', 'SF3B1', 'ASXL1')
aCML.clean = events.selection(aCML,
    filter.in.names=c(as.genes(alterations), gene.hypotheses))
```

```
## *** Events selection: #events =  31 , #types =  4 Filters freq|in|out = { FALSE ,  TRUE ,  FALSE }
## [filter.in] Genes hold:  TET2, EZH2, CBL, ASXL1, SETBP1  ...  [ 10 / 14  found].
## Selected  17  events, returning.
```

```
aCML.clean = annotate.description(aCML.clean,
    'CAPRI - Bionformatics aCML data (selected events)')
```

We show a new oncoprint of this latest dataset where we annotate the genes in `gene.hypotheses` in order to identify them. The sample names are also shown.

```
oncoprint(aCML.clean, gene.annot = list(priors = gene.hypotheses), sample.id = TRUE)
```

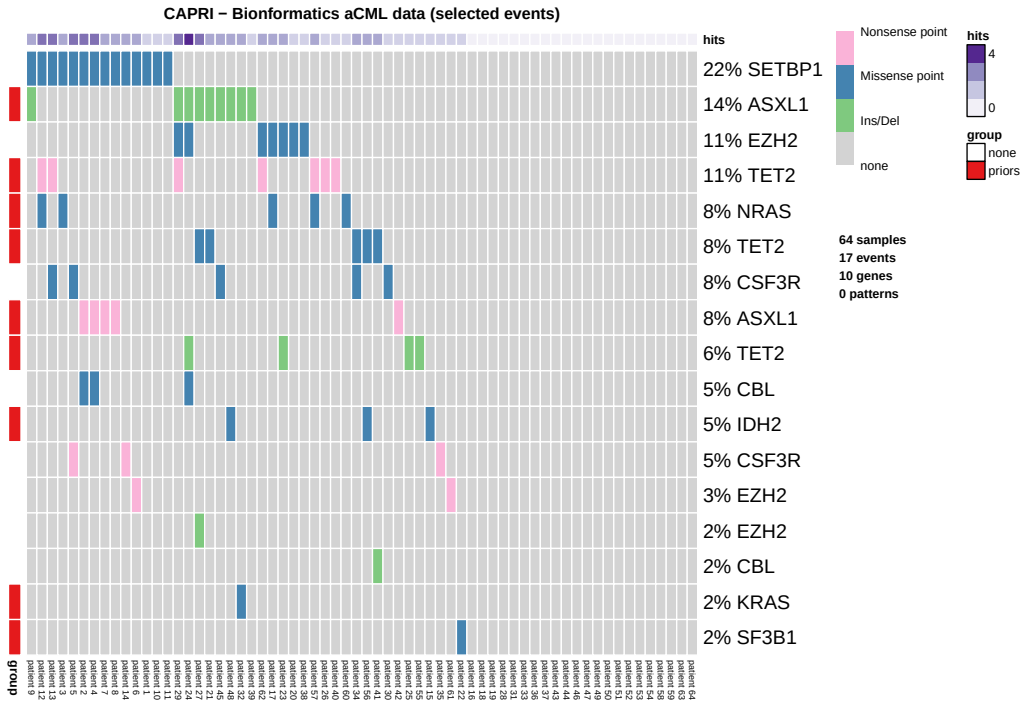

Figure 6: Data selected for aCML reconstruction annotated with the events which are part of a pattern that we will input to CAPRI.

6.1.1 Testable hypotheses via logical formulas (i.e., patterns)

CAPRI is the only algorithm in TRONCO that supports hypotheses-testing of causal structures expressed as logical formulas with AND, OR and XOR operators. An example invented formula could be

(APC:Mutation XOR APC:Deletion) OR CTNNB1:Mutation

where APC mutations and deletions are in disjunctive relation with CTNNB1 mutations; this is done to test if those events could confer equivalent fitness in terms of ensemble-level progression – see the original CAPRI paper and the PiCnIc pipeline for detailed explanations.

Every formula is transformed into a CAPRI "pattern". For every hypothesis it is possible to specify against which possible target event it should be tested, e.g., one might test the above formula against PIK3CA mutations, but not ATM ones. If this is not done, a pattern is tested against all other events in the dataset but those which constitute itself. A pattern tested against one other event is called an hypothesis.

Adding custom hypotheses. We add the hypotheses that are described in CAPRI's manuscript; we start with hard exclusivity (XOR) for NRAS/KRAS mutation,

NRAS:Missense point XOR KRAS:Missense point

tested against all the events in the dataset (default `pattern.effect = *`)

```
aCML.hypo = hypothesis.add(aCML.clean, 'NRAS xor KRAS', XOR('NRAS', 'KRAS'))
```

When a pattern is included, a new column in the dataset is created – whose signature is given by the evaluation of the formula constituting the pattern. We call this operation **lifting of a pattern**, and this shall create not inconsistency in the data – i.e., it shall not duplicate any of the other columns. TRONCO check this; for instance when we try to include a soft exclusivity (OR) pattern for the above genes we get an error (not shown).

```
aCML.hypo = hypothesis.add(aCML.hypo, 'NRAS or KRAS',  OR('NRAS', 'KRAS'))
```

Notice that TRONCO functions can be used to look at their alterations and understand why the OR signature is equivalent to the XOR one – this happens as no samples harbour both mutations.

```
oncoprint(events.selection(aCML.hypo,
    filter.in.names = c('KRAS', 'NRAS')),
    font.row = 8,
    ann.hits = FALSE)
```

We repeated the same analysis as before for other hypotheses and for the same reasons, we will include only the hard exclusivity pattern. In this case we add a two-levels pattern

SF3B1:Missense point XOR (ASXL1:Ins/Del XOR ASXL1:Nonsense point)

since ASXL1 is mutated in two different ways, and no samples harbour both mutation types.

```
aCML.hypo = hypothesis.add(aCML.hypo, 'SF3B1 xor ASXL1', XOR('SF3B1', XOR('ASXL1')),
    '*')
```

Finally, we now do the same for genes TET2 and IDH2. In this case 3 events for the gene TET2 are present, that is "Ins/Del", "Missense point" and "Nonsense point". For this reason, since we are not specifying any subset of such events to be considered, all TET2 alterations are used. Since the events present a perfect hard exclusivity, their patters will be included as a XOR.

```
as.events(aCML.hypo, genes = 'TET2')
```

```
##         type             event
## gene 4  "Ins/Del"        "TET2"
## gene 32 "Missense point" "TET2"
## gene 88 "Nonsense point" "TET2"
```

```
aCML.hypo = hypothesis.add(aCML.hypo,
    'TET2 xor IDH2',
    XOR('TET2', 'IDH2'),
    '*')
aCML.hypo = hypothesis.add(aCML.hypo,
    'TET2 or IDH2',
    OR('TET2', 'IDH2'),
    '*')
```

Which is the following pattern

(TET2:Ins/Del) XOR (TET2:Missense point) XOR (TET2:Nonsense point) XOR (IDH2:Missense point)

which we can visualize via an`oncoprint`.

```
oncoprint(events.selection(aCML.hypo,
    filter.in.names = c('TET2', 'IDH2')),
    font.row = 8,
    ann.hits = FALSE)
```

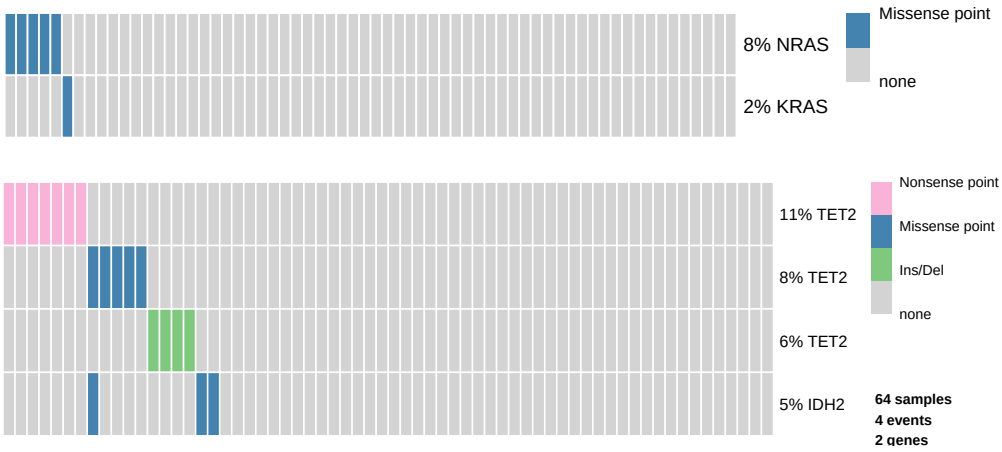

Figure 7: `oncoprint` output to show the perfect (hard) exclusivity among NRAS/KRAS mutations in aCML on top, and the sofy-one among TET2 and IDH2 alterations.

Adding (automatically) hypotheses for homologous events. We consider homologous events those having the same mnemonic name – as of function `as.genes` – but events of different type. For instance, mutations and deletions of the same gene would be considered such (e.g., in the aCML dataset ASXL1 Ins/Del and Nonsense point). It could be a good idea to test such events, in terms of progression fitness, to test is they might be equivalent; we can do that by building a pattern of exclusivity among them. TRONCO has a function to make this automatically which, by default, adds a soft exclusivity OR pattern among them.

```
aCML.hypo = hypothesis.add.homologous(aCML.hypo)

## *** Adding hypotheses for Homologous Patterns
##  Genes: TET2, EZH2, CBL, ASXL1, CSF3R
##  Function: OR
##  Cause: *
##  Effect: *
## .....Hypothesis created for all possible gene patterns.
```

This function added one pattern for each of TET2, EZH2, CBL, ASXL1, CSF3R (unless they created duplicated columns in the dataset), with a connective OR/XOR which is appropriate for the events considered; for instance the TET2 homologous pattern

(TET2:Ins/Del) XOR (TET2:Missense point) XOR (TET2:Nonsense point)

was created with a XOR function, as TET2 appears in perfect exclusivity.

Adding (automatically) hypotheses for a group of genes. The idea behind the previous function is generalized by `hypothesis.add.group`, that add a set of hypotheses that can be combinatorially created out of a group of genes. As such, this function can create an exponential number of hypotheses and should be used with caution as too many hypotheses, with respect to sample size, should not be included.

This function takes, among its inputs, the top-level logical connective, AND/OR/XOR, a minimum/maximum pattern size – to restrict the combinatorial sampling of subgroups –, plus a parameter that can be used to constrain the minimum event frequency. If, among the events included some of them have homologous, these are put automatically nested with the same logic of the `hypothesis.add.group` function.

```
dataset = hypothesis.add.group(aCML.clean, OR, group = c('SETBP1', 'ASXL1', 'CBL'))

## *** Adding Group Hypotheses
##  Group: SETBP1, ASXL1, CBL
##  Function: OR
##  Cause: * ;  Effect: * .
## Genes with multiple events:  ASXL1, CBL
## Generating  4 patterns [size: min = 3  -  max = 3 ].
```

```
## Hypothesis created for all possible patterns.
```

The final dataset that will be given as input to CAPRI is finally shown. Notice the signatures of all the lifted patterns.

```
oncoprint(aCML.hypo, gene.annot = list(priors = gene.hypotheses), sample.id = TRUE,
    font.row=10, font.column=5, cellheight=15, cellwidth=4)
```

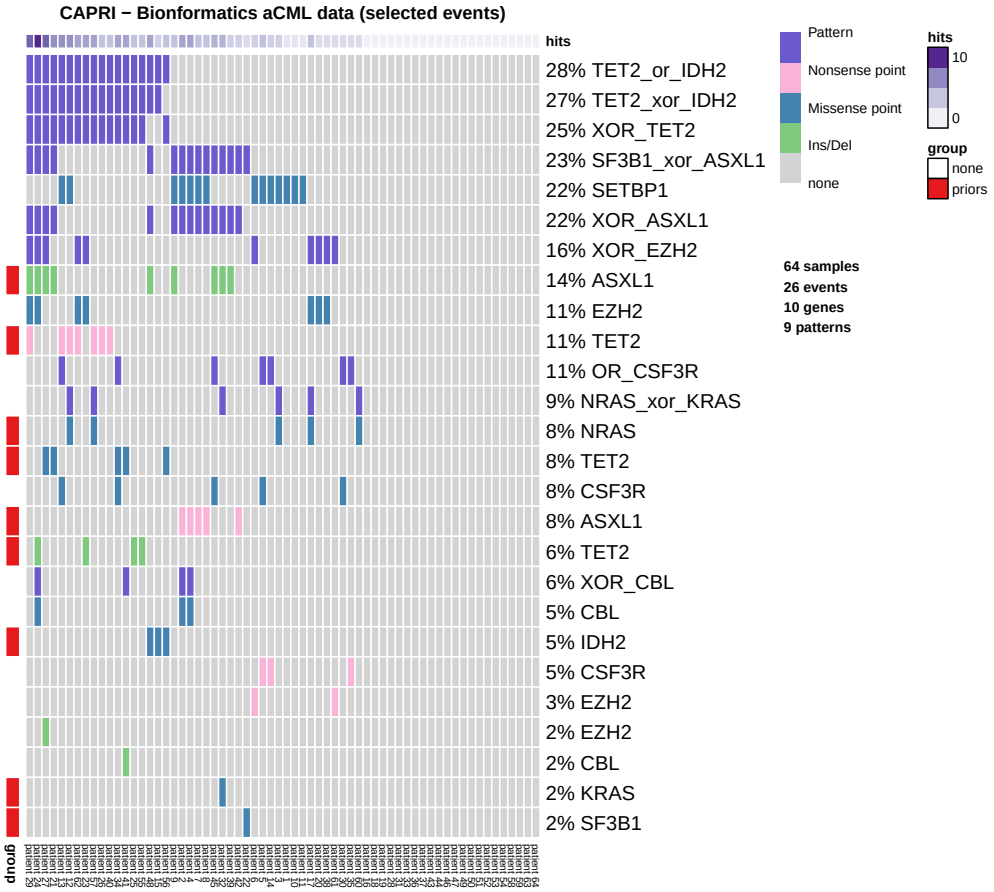

Figure 8: `oncoprint` output of the a dataset that has patterns that could be given as input to CAPRI to retrieve a progression model.

Querying, visualizing and manipulating CAPRI's patterns. We also provide functions to get the number of hypotheses and patterns present in the data.

```
npatterns(dataset)
```

```
## [1] 4
```

```
nhypotheses(dataset)
```

```
## [1] 106
```

We can visualize any pattern or the elements involved in them with the following functions.

```
as.patterns(dataset)
```

```
## [1] "OR_SETBP1_ASXL1"      "OR_SETBP1_CBL"        "OR_ASXL1_CBL"
## [4] "OR_SETBP1_ASXL1_CBL"
```

```
as.events.in.patterns(dataset)
```

```
##         type              event
```

```
## gene 6  "Ins/Del"        "CBL"
## gene 7  "Ins/Del"        "ASXL1"
## gene 29 "Missense point" "SETBP1"
## gene 34 "Missense point" "CBL"
## gene 91 "Nonsense point" "ASXL1"
```

```
as.genes.in.patterns(dataset)
```

```
## [1] "CBL"    "ASXL1"  "SETBP1"
```

```
as.types.in.patterns(dataset)
```

```
## [1] "Ins/Del"        "Missense point" "Nonsense point"
```

Similarily, we can enumerate the hypotheses with the function `as.hypotheses`, and delete certain patterns and hypotheses. Deleting a pattern consists in deleting all of its hypotheses.

```
head(as.hypotheses(dataset))
```

```
##      cause type cause event       effect type      effect event
## [1,] "Pattern"  "OR_SETBP1_ASXL1" "Ins/Del"        "TET2"
## [2,] "Pattern"  "OR_SETBP1_ASXL1" "Ins/Del"        "EZH2"
## [3,] "Pattern"  "OR_SETBP1_ASXL1" "Ins/Del"        "CBL"
## [4,] "Pattern"  "OR_SETBP1_ASXL1" "Missense point" "NRAS"
## [5,] "Pattern"  "OR_SETBP1_ASXL1" "Missense point" "KRAS"
## [6,] "Pattern"  "OR_SETBP1_ASXL1" "Missense point" "TET2"
```

```
dataset = delete.hypothesis(dataset, event = 'TET2')
dataset = delete.pattern(dataset, pattern = 'OR_ASXL1_CBL')
```

How to build a pattern. It is sometimes of help to plot some information about a certain combination of events, and a target – especially to disentangle the proper logical connectives to use when building a pattern. Here, we test genes SETBP1 and ASXL1 versus Missense point mutations of CSF3R, and observe that the majority of observations are mutually exclusive, but almost half of the CSF3R mutated samples with Missense point mutations do not harbout any mutation in SETBP1 and ASXL1.

```
tronco.pattern.plot(aCML,
    group = as.events(aCML, genes=c('SETBP1', 'ASXL1')),
    to = c('CSF3R', 'Missense point'),
    legend.cex=0.8,
    label.cex=1.0)
```

```
## Group:
##         type             event
## gene 7  "Ins/Del"        "ASXL1"
## gene 29 "Missense point" "SETBP1"
## gene 91 "Nonsense point" "ASXL1"
## Group tested against: CSF3R Missense point
## Pattern conditioned to samples: patient 5, patient 13, patient 30, patient 34, patient 45
## Co-occurrence in #samples:  0
## Hard exclusivity in #samples: 1 2 0
## Other observations in #samples: 2
## Soft exclusivity in #samples: 0
##                CSF3R
##   ASXL1            1
## SETBP1             2
##     ASXL1          0
## soft               0
## co-occurrence      0
## other              2
##      CSF3R Missense point
```

```
## [1,]                    3
## [2,]                    0
## [3,]                    0
## [4,]                    2
```

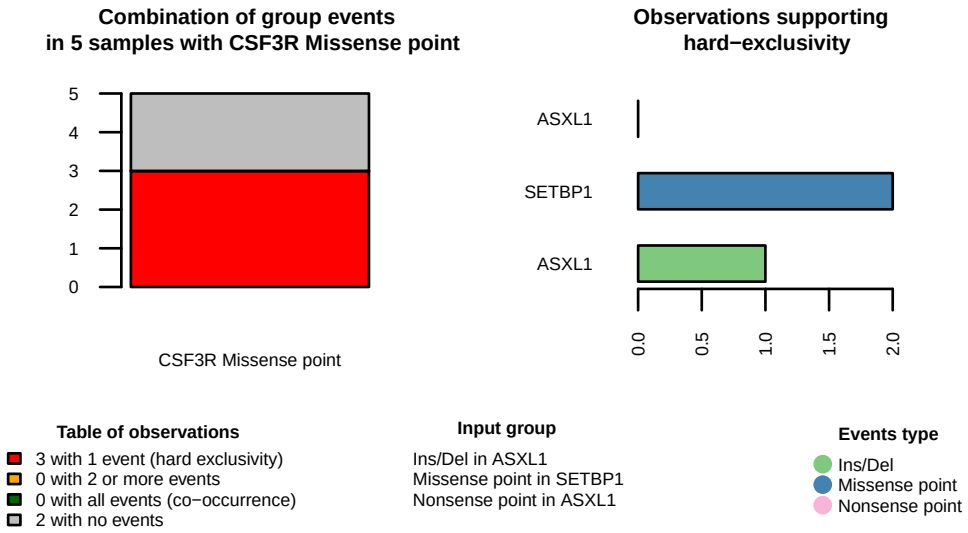

Figure 9: Barplot to show an hypothesis: here we test genes SETBP1 and ASXL1 versus Missense point mutations of CSF3R, which suggests that that pattern does not "capture" all the samples with CSF3R mutations.

It is also possible to create a circle plot where we can observe the contribution of genes SETBP1 and ASXL1 in every match with a Missense point mutations of CSF3R.

```
tronco.pattern.plot(aCML,
    group = as.events(aCML, genes=c('TET2', 'ASXL1')),
    to = c('CSF3R', 'Missense point'),
    legend = 1.0,
    label.cex = 0.8,
    mode='circos')
```

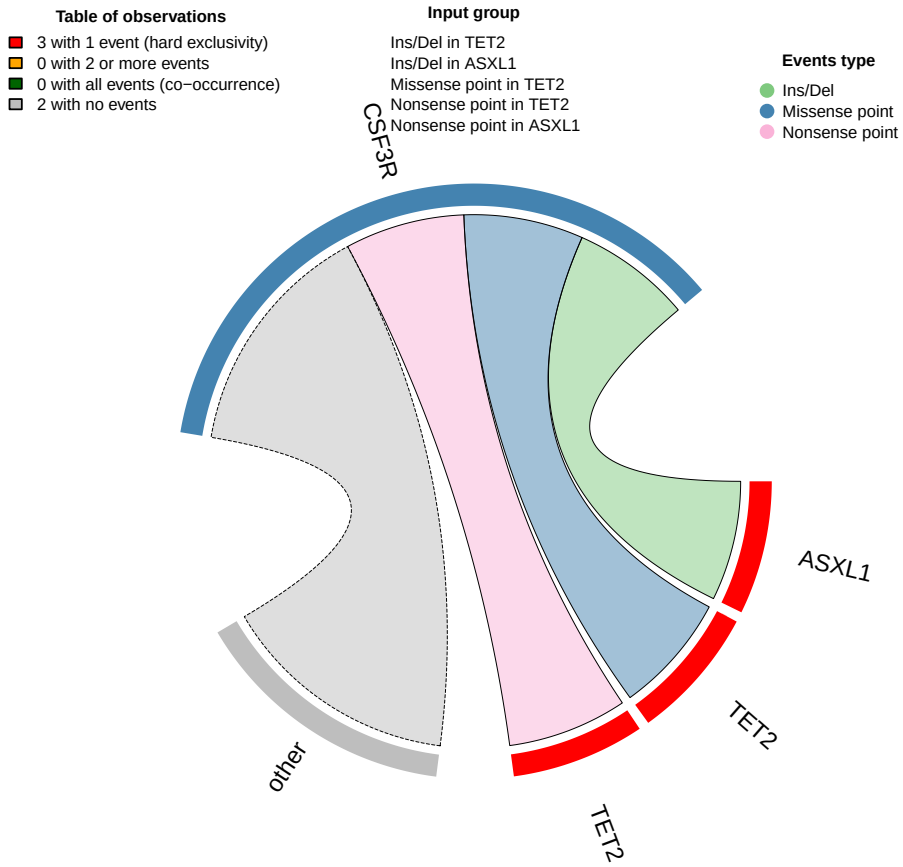

Figure 10: Circos to show an hypothesis: here we test genes SETBP1 and ASXL1 versus Missense point mutations of CSF3R. The combination of this and the previous plots should allow to understand which pattern we shall write in an attempt to capture a potential causality relation between the pattern and the event.

6.1.2 Model reconstruction

We run the inference of the model by CAPRI algorithm with its default parameter: we use both AIC and BIC as regularizators, Hill-climbing as heuristic search of the solutions and exhaustive bootstrap (`nboot` replicates or more for Wilcoxon testing, i.e., more iterations can be performed if samples are rejected), p-value are set at 0.05. We set the seed for the sake of reproducibility.

```
model.capri = tronco.capri(aCML.hypo, boot.seed = 12345, nboot = 5)

## *** Checking input events.
## *** Inferring a progression model with the following settings.
##  Dataset size: n = 64, m = 26.
##  Algorithm: CAPRI with "bic, aic" regularization and "hc" likelihood-fit strategy.
##  Random seed: 12345.
##  Bootstrap iterations (Wilcoxon): 5.
##  exhaustive bootstrap: TRUE.
##  p-value: 0.05.
##  minimum bootstrapped scores: 3.
## *** Bootstraping selective advantage scores (prima facie).
##  ......
##  Evaluating "temporal priority" (Wilcoxon, p-value 0.05)
```

```
##  Evaluating "probability raising" (Wilcoxon, p-value 0.05)
## *** Loop detection found loops to break.
##  Removed 37 edges out of 71 (52%)
## *** Performing likelihood-fit with regularization bic.
## *** Performing likelihood-fit with regularization aic.
## *** Evaluating BIC / AIC / LogLik informations.
## The reconstruction has been successfully completed in 00h:00m:03s
```

```
model.capri = annotate.description(model.capri, 'CAPRI - aCML')
```

6.2 CAPRESE

The CAPRESE algorithm is one of a set of algorithms to reconstruct progression models from data of an individual patient. This algorithm uses a shrinkage-alike estimator combining correlation and probability raising among pair of events. This algorithm shall return a forest of trees, a special case of a Suppes-Bayes Causal Network.

Despite this is derived to infer progression models from individual level data, we use it here to process aCML data (without patterns and with its default parameters). This algorithm has no bootstrap and, as such, is the quickest available in TRONCO.

```
model.caprese = tronco.caprese(aCML.clean)
```

```
## *** Checking input events.
## *** Inferring a progression model with the following settings.
##  Dataset size: n = 64, m = 17.
##  Algorithm: CAPRESE with shrinkage coefficient: 0.5.
## *** Evaluating LogLik informations.
## The reconstruction has been successfully completed in 00h:00m:00s
```

```
model.caprese = annotate.description(model.caprese, 'CAPRESE - aCML')
```

6.3 Directed Minimum Spanning Tree with Mutual Information

This algorithm is meant to extract a forest of trees of progression from data of an individual patient. This algorithm is based on a formulation of the problem in terms of minimum spamming trees and exploits results from Edmonds. We test it to infer a model from aCML data as we did with CAPRESE.

```
model.edmonds = tronco.edmonds(aCML.clean, nboot = 5, boot.seed = 12345)
```

```
## *** Checking input events.
## *** Inferring a progression model with the following settings.
##  Dataset size: n = 64, m = 17.
##  Algorithm: Edmonds with "no_reg" regularization Random seed: 12345.
##  Bootstrap iterations (Wilcoxon): 5.
##  exhaustive bootstrap: TRUE.
##  p-value: 0.05.
##  minimum bootstrapped scores: 3.
## *** Bootstraping selective advantage scores (prima facie).
##  ......
##  Evaluating "temporal priority" (Wilcoxon, p-value 0.05)
##  Evaluating "probability raising" (Wilcoxon, p-value 0.05)
## *** Loop detection found loops to break.
##  Removed 2 edges out of 24 (8%)
## *** Performing likelihood-fit with regularization: no_reg and score: pmi .
## *** Evaluating BIC / AIC / LogLik informations.
## The reconstruction has been successfully completed in 00h:00m:01s
```

```
model.edmonds = annotate.description(model.edmonds, 'MST Edmonds - aCML')
```

6.4 Partially Directed Minimum Spanning Tree with Mutual Information

This algorithm extends the previous one in situations where it is not possible to fully assess a confident time ordering among the nodes, hence leading to a partially directed input. This algorithm adopts Gabow search strategy to evaluate the best directed minimum spanning tree among such undirected components. We test it to infer a model from aCML data as all the other algorithms.

```
model.gabow = tronco.gabow(aCML.clean, nboot = 5, boot.seed = 12345)
```

```
## *** Checking input events.
## *** Inferring a progression model with the following settings.
##  Dataset size: n = 64, m = 17.
##  Algorithm: Gabow with "no_reg" regularization Random seed: 12345.
##  Bootstrap iterations (Wilcoxon): 5.
##  exhaustive bootstrap: TRUE.
##  p-value: 0.05.
##  minimum bootstrapped scores: 3.
## *** Bootstraping selective advantage scores (prima facie).
##  ......
##  Evaluating "temporal priority" (Wilcoxon, p-value 0.05)
##  Evaluating "probability raising" (Wilcoxon, p-value 0.05)
## *** Loop detection found loops to break.
##  Removed 2 edges out of 24 (8%)
## *** Performing likelihood-fit with regularization: no_reg .
## *** Evaluating BIC / AIC / LogLik informations.
## The reconstruction has been successfully completed in 00h:00m:01s
```

```
model.gabow = annotate.description(model.gabow, 'MST Gabow - aCML')
```

6.5 Undirected Minimum Spanning Tree with Likelihood-Fit

This algorithm is meant to extract a progression from data of an individual patient, but it is not constrained to retrieve a tree/forest – i.e., it could retrieve a direct acyclic graph – according to the level of noise and heterogeneity of the input data. This algorithm is based on a formulation of the problem in terms of minimum spamming trees and exploits results from Chow Liu and other variants for likelihood-fit. Thus, this algorithm is executed with potentially multiple regularizator as CAPRI – here we use BIC/AIC.

We test it to aCML data as all the other algorithms.

```
model.chowliu = tronco.chowliu(aCML.clean, nboot = 5, boot.seed = 12345)
```

```
## *** Checking input events.
## *** Inferring a progression model with the following settings.
##  Dataset size: n = 64, m = 17.
##  Algorithm: Chow Liu with "bic, aic" regularization Random seed: 12345.
##  Bootstrap iterations (Wilcoxon): 5.
##  exhaustive bootstrap: TRUE.
##  p-value: 0.05.
##  minimum bootstrapped scores: 3.
## *** Bootstraping selective advantage scores (prima facie).
##  ......
##  Evaluating "temporal priority" (Wilcoxon, p-value 0.05)
##  Evaluating "probability raising" (Wilcoxon, p-value 0.05)
## *** Loop detection found loops to break.
##  Removed 2 edges out of 24 (8%)
## *** Performing likelihood-fit with regularization bic .
## *** Performing likelihood-fit with regularization aic .
## *** Evaluating BIC / AIC / LogLik informations.
## The reconstruction has been successfully completed in 00h:00m:02s
```

```
model.chowliu = annotate.description(model.chowliu, 'MST Chow Liu - aCML')
```

6.6 Undirected Minimum Spanning Tree with Mutual Information

This algorithm is meant to extract a progression from data of an individual patient. As the Chow Liu algorithm, this could retrieve a direct acyclic graph according to the level of noise and heterogeneity of the input data. This algorithm formulatesf the problem in terms of undirected minimum spamming trees and exploits results from Prim, which are a generalization of Edomonds' ones. We test it to aCML data as all the other algorithms.

```
model.prim = tronco.prim(aCML.clean, nboot = 5, boot.seed = 12345)
```

```
## *** Checking input events.
## *** Inferring a progression model with the following settings.
##  Dataset size: n = 64, m = 17.
##  Algorithm: Prim with "no_reg" regularization Random seed: 12345.
##  Bootstrap iterations (Wilcoxon): 5.
##  exhaustive bootstrap: TRUE.
##  p-value: 0.05.
##  minimum bootstrapped scores: 3.
## *** Bootstraping selective advantage scores (prima facie).
##  ......
##  Evaluating "temporal priority" (Wilcoxon, p-value 0.05)
##  Evaluating "probability raising" (Wilcoxon, p-value 0.05)
## *** Loop detection found loops to break.
##  Removed 2 edges out of 24 (8%)
## *** Performing likelihood-fit with regularization: no_reg .
## *** Evaluating BIC / AIC / LogLik informations.
## The reconstruction has been successfully completed in 00h:00m:03s
```

```
model.prim = annotate.description(model.prim, 'MST Prim - aCML data')
```

7 Post-reconstruction

TRONCO provides functions to plot a model, access information about the probabilities used to extract it from data, and two types of confidence measures: those used to infer the model, and those computed a posteriori from it.

Function `view` provides updated information about a model if this is available.

```
view(model.capri)
```

```
## Description: CAPRI - aCML.
## -- TRONCO Dataset: n=64, m=26, |G|=10, patterns=9.
## Events (types): Ins/Del, Missense point, Nonsense point, Pattern.
## Colors (plot): #7FC97F, #4483B0, #fab3d8, slateblue.
## Events (5 shown):
##   gene 4 : Ins/Del TET2
##   gene 5 : Ins/Del EZH2
##   gene 6 : Ins/Del CBL
##   gene 7 : Ins/Del ASXL1
##   gene 29 : Missense point SETBP1
## Genotypes (5 shown):
##
## -- TRONCO Model(s):  CAPRI
## Score optimization via Hill-Climbing.
## BIC, AIC regularizers.
## BIC: score -566.2114 | logLik -501.7487 | 5 selective advantage relations.
## AIC: score -528.5392 | logLik -491.5392 | 11 selective advantage relations.
```

```
## Available confidence measures:
##  Temporal priority | Probability raising | Hypergeometric
```

7.1 Visualizing a reconstructed model

We can plot a model by using function `tronco.plot`. Here, we plot the aCML model inferred by CAPRI with BIC and AIC as a regolarizator. We set some parameters to get a nice plot (scaling etc.), and distinguish the edges detected by the two regularization techniques. The confidence of each edge is shown in terms of temporal priority and probability raising (selective advantage scores) and hypergeometric testing (statistical relevance of the dataset of input). Events are annotated as in the oncoprint, edge p-values above a minium threshold (default 0.05) are red.

```
tronco.plot(model.capri,
    fontsize = 12,
    scale.nodes = 0.6,
    confidence = c('tp', 'pr', 'hg'),
    height.logic = 0.25,
    legend.cex = 0.35,
    pathways = list(priors = gene.hypotheses),
    label.edge.size = 10)
```

```
## *** Expanding hypotheses syntax as graph nodes:
## *** Rendering graphics
## Nodes with no incoming/outgoing edges will not be displayed.
## Annotating nodes with pathway information.
## Annotating pathways with RColorBrewer color palette Set1 .
## Adding confidence information: tp, pr, hg
## RGraphviz object prepared.
## Plotting graph and adding legends.
```

```
## Warning in strwidth(legend, units = "user", cex = cex, font = text.font):  font metrics unknown
for character 0xa
```

```
## Warning in strheight(legend, units = "user", cex = cex):  font metrics unknown for character
0xa
```

```
## Warning in text.default(x, y, ...):  font metrics unknown for character 0xa
```

```
## Warning in text.default(x, y, ...):  font metrics unknown for character 0xa
```

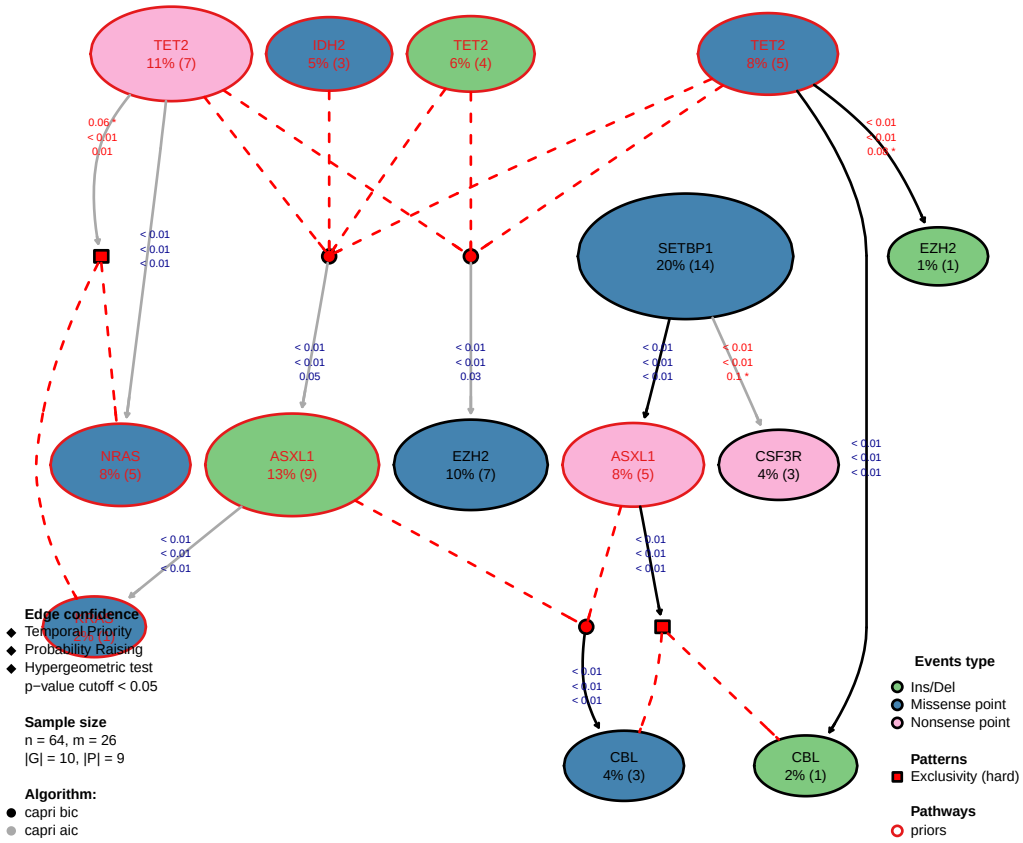

Figure 11: aCML model reconstructed by CAPRI with AIC/BIC as regolarizators; the confidence of each edge is shown both in terms of temporal priority and probability raising (selective advantage scores) and hypergeometric testing (statistical relevance of the dataset of input).

We can also make a multiplot with this function, which in this case we do by showing the models inferred by the other algorithms based on Minimum Spanning Trees.

```
par(mfrow = c(2,2))
tronco.plot(model.caprese, fontsize = 22, scale.nodes = 0.6, legend = FALSE)
tronco.plot(model.edmonds, fontsize = 22, scale.nodes = 0.6, legend = FALSE)
tronco.plot(model.chowliu, fontsize = 22, scale.nodes = 0.6, legend.cex = .7)
```

```
## Warning in strwidth(legend, units = "user", cex = cex, font = text.font):  font metrics unknown
for character 0xa
```

```
## Warning in strheight(legend, units = "user", cex = cex):  font metrics unknown for character
0xa
```

```
## Warning in text.default(x, y, ...):  font metrics unknown for character 0xa
```

```
## Warning in text.default(x, y, ...):  font metrics unknown for character 0xa
```

```
tronco.plot(model.prim, fontsize = 22, scale.nodes = 0.6, legend = FALSE)
```

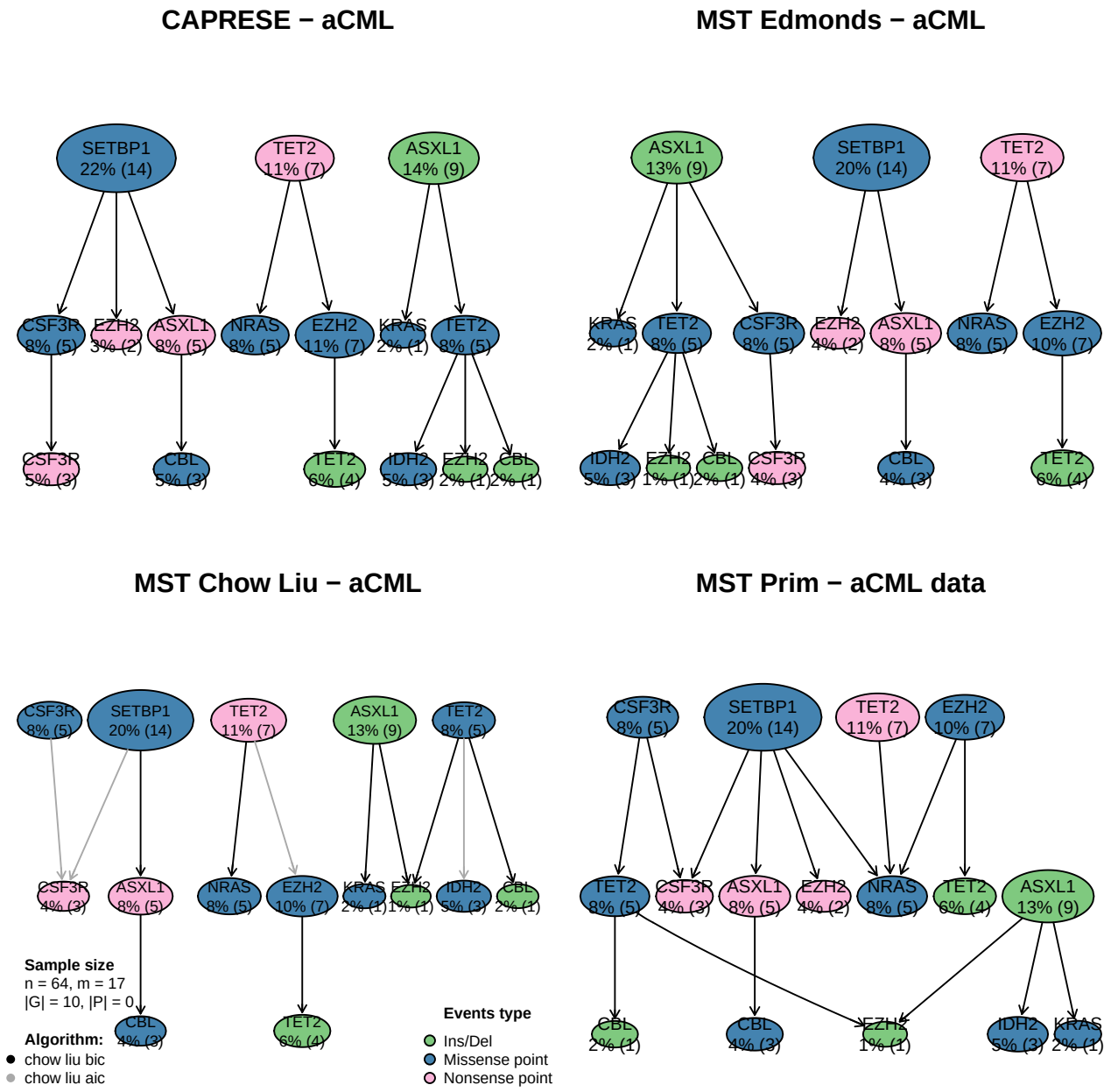

Figure 12: aCML data processed model by algorithms to extract models from individual patients, we show the otput of CAPRESE, and all algorithms based on Minimum Spanning Trees (Edmonds, Chow Liu and Prim). Only the model retrieved by Chow Liu has two different edge colors as it was regularized with two different strategies: AIC and BIC.

7.2 Accessing information within a model (e.g., confidence)

We can visualize a summary of the parameters used for the reconstruction, test if an object has a model or delete it (which shall be done to retrieve the original dataset).

```
as.data.frame(as.parameters(model.capri))
```

```
##   algorithm command regularization do.boot nboot pvalue min.boot min.stat boot.seed
## 1     CAPRI      hc            bic    TRUE     5   0.05        3     TRUE     12345
## 2     CAPRI      hc            aic    TRUE     5   0.05        3     TRUE     12345
##   silent error.rates.epos error.rates.eneg
## 1  FALSE               0                0
```

```
## 2  FALSE                0                0
```

```
has.model(model.capri)
```

```
## [1] TRUE
```

```
dataset = delete.model(model.capri)
```

Model structure. A set of functions can be used to visualize the content of object which contains the reconstructed model. For instance, we can access the adjacency matrix of a model by using `as.adj.matrix` which will return a matrix for each one of the regularizators used – in this case because CAPRI was run with both BIC/AIC.

```
str(as.adj.matrix(model.capri))
```

```
## List of 2
##  $ capri_bic: num [1:26, 1:26] 0 0 0 0 0 0 0 0 0 0 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:26] "gene 4" "gene 5" "gene 6" "gene 7" ...
##   .. ..$ : chr [1:26] "gene 4" "gene 5" "gene 6" "gene 7" ...
##  $ capri_aic: num [1:26, 1:26] 0 0 0 0 0 0 0 0 0 0 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:26] "gene 4" "gene 5" "gene 6" "gene 7" ...
##   .. ..$ : chr [1:26] "gene 4" "gene 5" "gene 6" "gene 7" ...
```

Empirical probabilities. Every model is inferred by estimating the empirical marginal, joint and conditional probabilities for all the events, from input data. These in some cases are estimated by a bootstrap procedure (see the algorithms implemented). TRONCO has functions to extract such table, that could be in turn printed by using external functions for, e.g., heatmap visualization (see below for an example via the *pheatmap* package). We show these functions working with the CAPRI model; in this case the tables are the same for both BIC/AIC structures as they are computed before performing penalized likelihood-fit. The marginal $P(x)$ for x an event in the dataset are obtained by `as.marginal.probs`.

```
marginal.prob = as.marginal.probs(model.capri)
head(marginal.prob$capri_bic)
```

```
##         marginal probability
## gene 4           0.057291667
## gene 5           0.009548611
## gene 6           0.015625000
## gene 7           0.132812500
## gene 29          0.198784722
## gene 30          0.077256944
```

Similarly, the joint $P(x, y)$ for every pair of events in the dataset is given by `as.joint.probs`.

```
joint.prob = as.joint.probs(model.capri, models='capri_bic')
joint.prob$capri_bic[1:3, 1:3]
```

```
##            gene 4      gene 5   gene 6
## gene 4 0.05729167 0.000000000 0.000000
## gene 5 0.00000000 0.009548611 0.000000
## gene 6 0.00000000 0.000000000 0.015625
```

And `as.conditional.probs` finally gives the conditional $P(x \mid y)$ for every edge in the dataset.

```
conditional.prob = as.conditional.probs(model.capri, models='capri_bic')
head(conditional.prob$capri_bic)
```

```
##         conditional probability
## gene 4  1
## gene 5  0.1264368
## gene 6  0.2068966
```

```
## gene 7  1
## gene 29 1
## gene 30 1
```

Confidence measures. Confidence scores can be accessed by function `as.confidence`, which takes as parameter the type of confidence measure that one wants to access to. This will work for either confidence measures assessed before reconstructing the model – if available –, or afterwards.

```
str(as.confidence(model.capri, conf = c('tp', 'pr', 'hg')))
```

```
## List of 3
##  $ hg: num [1:26, 1:26] 1 0.0625 0.0625 0.4632 0.6067 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:26] "gene 4" "gene 5" "gene 6" "gene 7" ...
##   .. ..$ : chr [1:26] "gene 4" "gene 5" "gene 6" "gene 7" ...
##  $ tp: num [1:26, 1:26] 1.00 1.00 1.00 7.66e-07 1.54e-07 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:26] "gene 4" "gene 5" "gene 6" "gene 7" ...
##   .. ..$ : chr [1:26] "gene 4" "gene 5" "gene 6" "gene 7" ...
##  $ pr: num [1:26, 1:26] 1 1 1 0.557 1 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:26] "gene 4" "gene 5" "gene 6" "gene 7" ...
##   .. ..$ : chr [1:26] "gene 4" "gene 5" "gene 6" "gene 7" ...
```

Other functions visualize tables summarizing the statistics for each edge in the model, For instance, if one uses function `as.selective.advantage.relations` the p-values for temporal priority, probability raising and hypergeometric testing, as well as other information about each edge can be accessed, e.g., the number of observations for the upstream and the downstream events.

```
as.selective.advantage.relations(model.capri)
```

```
## $capri_bic
##                 SELECTS             SELECTED OBS.SELECTS OBS.SELECTED TEMPORAL.PRIORITY
## 1 Missense point SETBP1 Nonsense point ASXL1          14            5      3.586517e-07
## 2   Missense point TET2         Ins/Del EZH2           5            1      3.042779e-07
## 3   Missense point TET2          Ins/Del CBL           5            1      5.138164e-07
## 4  Nonsense point ASXL1      Pattern XOR_CBL           5            4      4.969218e-03
## 5     Pattern XOR_ASXL1   Missense point CBL          14            3      1.432328e-07
##   PROBABILITY.RAISING HYPERGEOMETRIC
## 1        1.575005e-07   0.0049513989
## 2        1.981523e-04   0.0781250000
## 3        4.930027e-06   0.0000000000
## 4        1.868612e-04   0.0009364534
## 5        5.704252e-08   0.0087365591
##
## $capri_aic
##                  SELECTS              SELECTED OBS.SELECTS OBS.SELECTED TEMPORAL.PRIORITY
## 1          Ins/Del ASXL1   Missense point KRAS           9            1      1.576141e-07
## 2  Missense point SETBP1  Nonsense point ASXL1          14            5      3.586517e-07
## 3  Missense point SETBP1  Nonsense point CSF3R          14            3      1.605563e-07
## 4    Missense point TET2          Ins/Del EZH2           5            1      3.042779e-07
## 5    Missense point TET2           Ins/Del CBL           5            1      5.138164e-07
## 6    Nonsense point TET2   Missense point NRAS           7            5      2.548590e-03
## 7    Nonsense point TET2 Pattern NRAS_xor_KRAS           7            6      6.056954e-02
## 8   Nonsense point ASXL1       Pattern XOR_CBL           5            4      4.969218e-03
## 9  Pattern TET2_xor_IDH2         Ins/Del ASXL1          17            9      7.605877e-07
## 10      Pattern XOR_TET2   Missense point EZH2          16            7      2.129905e-07
## 11     Pattern XOR_ASXL1    Missense point CBL          14            3      1.432328e-07
##    PROBABILITY.RAISING HYPERGEOMETRIC
```

```
## 1          7.766886e-05   0.0000000000
## 2          1.575005e-07   0.0049513989
## 3          9.491867e-06   0.1023425499
## 4          1.981523e-04   0.0781250000
## 5          4.930027e-06   0.0000000000
## 6          3.058910e-05   0.0075907809
## 7          5.965059e-05   0.0144204483
## 8          1.868612e-04   0.0009364534
## 9          3.653246e-07   0.0450857493
## 10         9.501132e-07   0.0302854437
## 11         5.704252e-08   0.0087365591
```

7.3 Confidence via non-parametric and statistical bootstrap

TRONCO provides three different strategies to perform bootstrap and assess confidence of each edge in terms of a score in the range [0, 100] (100 is the highest confidence). Non-parametric (default) and statistical bootstrap strategies are available, and can be executed by calling function `tronco.bootstrap` with `type` parameter set appropriately. This function is parallel, and parameter `cores.ratio` (default 1) can be used to percentage of available cores that shall be used to compute the scores. Parameter `nboot` controls the number of bootstrap iterations.

```
model.boot = tronco.bootstrap(model.capri, nboot = 3)
```

```
## *** Executing now the bootstrap procedure, this may take a long time...
##  Expected completion in approx. 00h:00m:01s
##  Using 7 cores via "parallel"
##  Reducing results
## Performed non-parametric bootstrap with 3 resampling and 0.05 as pvalue for the statistical tests.
```

```
model.boot = tronco.bootstrap(model.boot, nboot = 3, type = 'statistical')
```

```
## *** Executing now the bootstrap procedure, this may take a long time...
##  Expected completion in approx. 00h:00m:01s
##  Using 7 cores via "parallel"
##  Reducing results
## Performed statistical bootstrap with 3 resampling and 0.05 as pvalue for the statistical tests.
```

Bootstrap scores can be annotated to the `tronco.plot` output by setting them via the confidence parameter `confidence=c('npb', 'sb')`. In this case edge thickness will be proportional to the non-parametric (`npb`) scores – the last to appear in the `confidence` parameter.

```
tronco.plot(model.boot,
    fontsize = 12,
    scale.nodes = .6,
    confidence=c('sb', 'npb'),
    height.logic = 0.25,
    legend.cex = .35,
    pathways = list(priors= gene.hypotheses),
    label.edge.size=10)
```

```
## *** Expanding hypotheses syntax as graph nodes:
## *** Rendering graphics
## Nodes with no incoming/outgoing edges will not be displayed.
## Annotating nodes with pathway information.
## Annotating pathways with RColorBrewer color palette Set1 .
## Adding confidence information: sb, npb
## RGraphviz object prepared.
## Plotting graph and adding legends.
```

```
## Warning in strwidth(legend, units = "user", cex = cex, font = text.font):  font metrics unknown
for character 0xa
```

```
## Warning in strheight(legend, units = "user", cex = cex):  font metrics unknown for character
Oxa
## Warning in text.default(x, y, ...):  font metrics unknown for character Oxa
## Warning in text.default(x, y, ...):  font metrics unknown for character Oxa
```

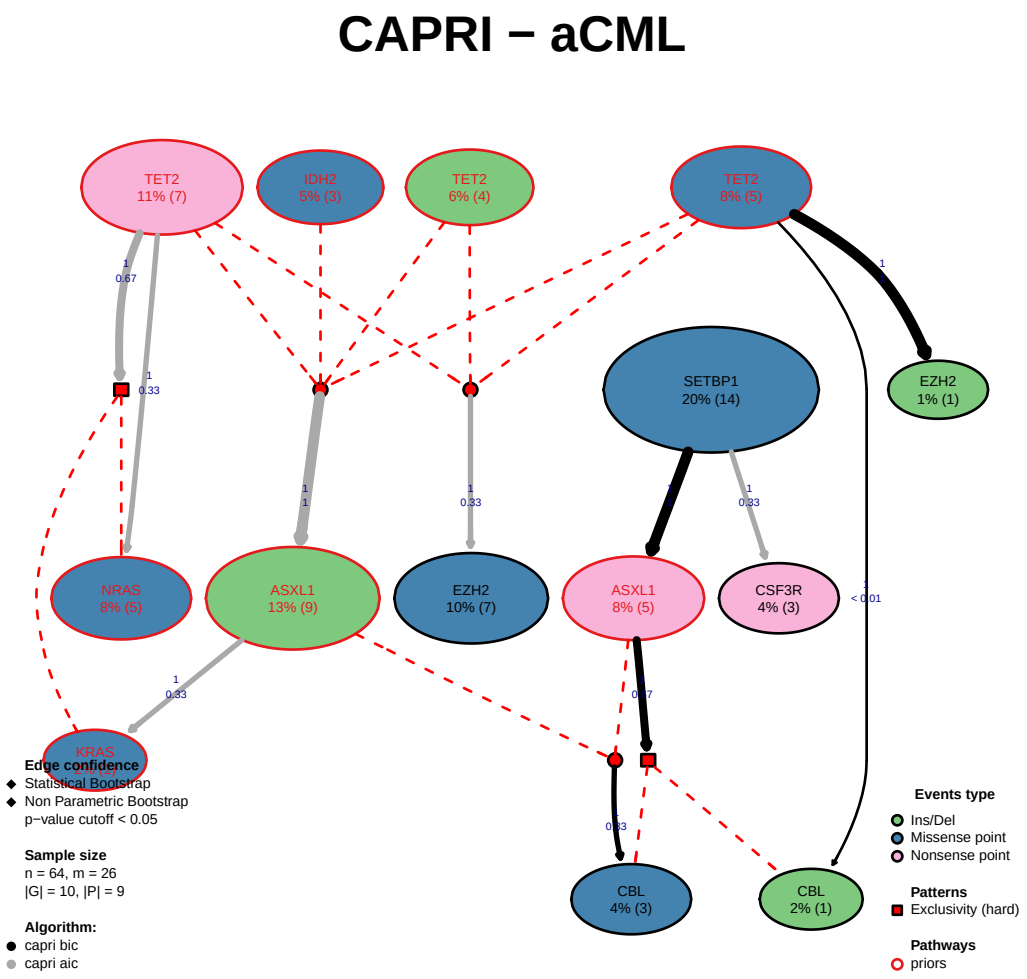

Figure 13: aCML model reconstructed by CAPRI with AIC/BIC as regolarizators and annotated with both non-parametric and statistical bootstrap scores. Edge thickness is proportional to the non-parametric scores.

Bootstrap scores can extracted or visualized even with other TRONCO functions. For instance, we can accessall scores via `as.bootstrap.scores`, which resembles function `as.selective.advantage.relations` and will display the scores per edge. Notice that even function `view` gives an update output by mentioning the available bootstrap scores.

```
as.bootstrap.scores(model.boot)
```

```
## $capri_bic
##                  SELECTS             SELECTED OBS.SELECTS OBS.SELECTED NONPAR.BOOT
## 1 Missense point SETBP1 Nonsense point ASXL1          14             5   100.00000
## 2   Missense point TET2         Ins/Del EZH2           5             1   100.00000
## 3   Missense point TET2          Ins/Del CBL           5             1     0.00000
## 4  Nonsense point ASXL1      Pattern XOR_CBL           5             4    66.66667
## 5     Pattern XOR_ASXL1   Missense point CBL          14             3    33.33333
##   STAT.BOOT
## 1       100
## 2       100
```

```
## 3       100
## 4       100
## 5       100
##
## $capri_aic
##                  SELECTS              SELECTED OBS.SELECTS OBS.SELECTED NONPAR.BOOT
## 1          Ins/Del ASXL1   Missense point KRAS           9            1    33.33333
## 2  Missense point SETBP1  Nonsense point ASXL1          14            5   100.00000
## 3  Missense point SETBP1  Nonsense point CSF3R          14            3    33.33333
## 4    Missense point TET2          Ins/Del EZH2           5            1   100.00000
## 5    Missense point TET2           Ins/Del CBL           5            1     0.00000
## 6    Nonsense point TET2   Missense point NRAS           7            5    33.33333
## 7    Nonsense point TET2 Pattern NRAS_xor_KRAS           7            6    66.66667
## 8   Nonsense point ASXL1       Pattern XOR_CBL           5            4   100.00000
## 9  Pattern TET2_xor_IDH2         Ins/Del ASXL1          17            9   100.00000
## 10      Pattern XOR_TET2   Missense point EZH2          16            7    33.33333
## 11     Pattern XOR_ASXL1    Missense point CBL          14            3    33.33333
##    STAT.BOOT
## 1        100
## 2        100
## 3        100
## 4        100
## 5        100
## 6        100
## 7        100
## 8        100
## 9        100
## 10       100
## 11       100
```

```
view(model.boot)
```

```
## Description: CAPRI - aCML.
## -- TRONCO Dataset: n=64, m=26, |G|=10, patterns=9.
## Events (types): Ins/Del, Missense point, Nonsense point, Pattern.
## Colors (plot): #7FC97F, #4483B0, #fab3d8, slateblue.
## Events (5 shown):
##   gene 4 : Ins/Del TET2
##   gene 5 : Ins/Del EZH2
##   gene 6 : Ins/Del CBL
##   gene 7 : Ins/Del ASXL1
##   gene 29 : Missense point SETBP1
## Genotypes (5 shown):
##
## -- TRONCO Model(s):  CAPRI
## Score optimization via Hill-Climbing.
## BIC, AIC regularizers.
## BIC: score -566.2114 | logLik -501.7487 | 5 selective advantage relations.
## AIC: score -528.5392 | logLik -491.5392 | 11 selective advantage relations.
## Available confidence measures:
##  Temporal priority | Probability raising | Hypergeometric
## Bootstrap estimation available.
```

If we want to access a matrix with the scores and visualize that in a heatmap we can use for instance the `pheatmap` function of TRONCO. In this case we need to use also function `keysToNames` to translate internal TRONCO keys to mnemonic names in the plot

```
pheatmap(keysToNames(model.boot, as.confidence(model.boot, conf = 'sb')$sb$capri_aic) * 100,
          main =  'Statistical bootstrap scores for AIC model',
```

```
        fontsize_row = 6,
        fontsize_col = 6,
        display_numbers = TRUE,
        number_format = "%d"
        )
```

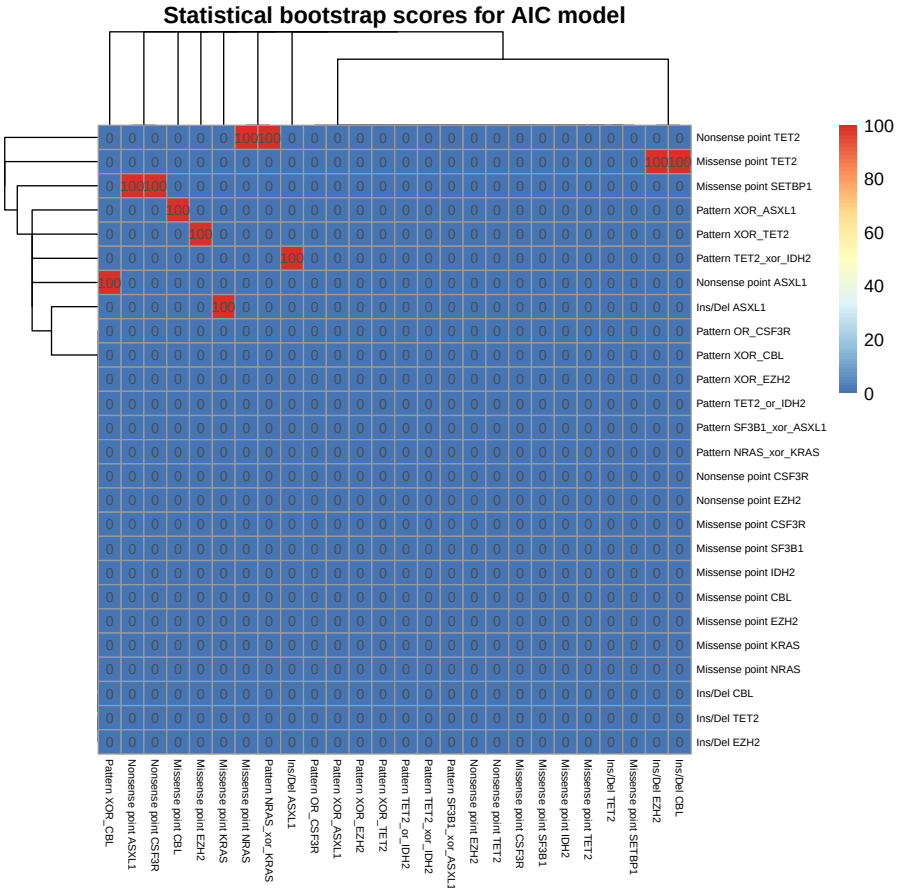

Figure 14: Heatmap of the bootstrap scores for the CAPRI aCML model (via AIC regularization).

7.4 Confidence via cross-validation (entropy loss, prediction and posterior classification errors)

TRONCO implements k-fold cross-validation routines (from the *bnlearn* package) to provide estimates of the following statistics:

- the *negative entropy* (via `tronco.kfold.eloss`) of a whole model ? i.e., the negated expected log-likelihood of the test set for the Bayesian network fitted from the training set.
- the *prediction error* (via `tronco.kfold.prederr`) for a single node x and its parents set X – i.e., how precisely we can predict the values of x by using only the information present in its local distribution, via X.
- the *posterior classification error* (via `tronco.kfold.posterr`) for a single node x and one of its parent node $y \in X$ – i.e., the values of x are predicted using only the information present in y by likelihood weighting and Bayesian posterior estimates.

By default, a 10 repetitions from 10-fold cross-validation experiment are perfomed, for all the models which are found inside a TRONCO object – in this case 2, one for CAPRI with BIC and one for CAPRI with AIC.

```
model.boot = tronco.kfold.eloss(model.boot)

## Calculating entropy loss with k-fold cross-validation
##   [ k = 10 | runs = 10 | regularizer = capri_bic ] ...  DONE
```

```
##   Model logLik = -501.7487
##   Mean   eloss = 8.363665  |  1.666903 %
##   Stdev  eloss =  0.03832627
## Calculating entropy loss with k-fold cross-validation
##  [ k = 10 | runs = 10 | regularizer = capri_aic ] ...  DONE
##   Model logLik = -491.5392
##   Mean   eloss = 8.302625  |  1.689108 %
##   Stdev  eloss =  0.04194586
```

```
model.boot = tronco.kfold.prederr(model.boot, runs = 2)
```

```
## *** Using 7 cores via "parallel"
##  Scanning 26 nodes for their prediction error (all parents).
##  Regularizer:  capri_bic
## *** Reducing results
##  Scanning 26 nodes for their prediction error (all parents).
##  Regularizer:  capri_aic
## *** Reducing results
```

```
model.boot = tronco.kfold.posterr(model.boot, runs = 2)
```

```
## *** Using 7 cores via "parallel"
##  Scanning 5 edges for posterior classification error.
##  Regularizer: capri_bic
## *** Reducing results
##  Scanning 11 edges for posterior classification error.
##  Regularizer: capri_aic
## *** Reducing results
```

These results can be visualized in terms of summary tables, as for the other confidence scores.

```
as.kfold.eloss(model.boot)
```

```
##               Mean %-of-logLik      Stdev
## capri_bic 8.363665    1.666903 0.03832627
## capri_aic 8.302625    1.689108 0.04194586
```

```
as.kfold.prederr(model.boot)
```

```
## $capri_bic
##                    SELECTED MEAN.PREDERR SD.PREDERR
## 5    Missense point SETBP1     0.218750          0
## 6      Missense point NRAS     0.078125          0
## 7      Missense point KRAS     0.015625          0
## 8      Missense point TET2     0.078125          0
## 9      Missense point EZH2     0.109375          0
## 10      Missense point CBL     0.046875          0
## 11     Missense point IDH2     0.046875          0
## 12    Missense point SF3B1     0.015625          0
## 13    Missense point CSF3R     0.078125          0
## 14     Nonsense point TET2     0.109375          0
## 15     Nonsense point EZH2     0.031250          0
## 16    Nonsense point ASXL1     0.078125          0
## 17    Nonsense point CSF3R     0.046875          0
## 18   Pattern NRAS_xor_KRAS     0.093750          0
## 19 Pattern SF3B1_xor_ASXL1     0.234375          0
## 20   Pattern TET2_xor_IDH2     0.265625          0
## 21    Pattern TET2_or_IDH2     0.281250          0
## 22        Pattern XOR_TET2     0.250000          0
## 23        Pattern XOR_EZH2     0.156250          0
## 24         Pattern XOR_CBL     0.078125          0
## 25       Pattern XOR_ASXL1     0.218750          0
```

```
## 26        Pattern OR_CSF3R     0.109375          0
##
## $capri_aic
##                   SELECTED MEAN.PREDERR SD.PREDERR
## 5    Missense point SETBP1    0.2187500 0.00000000
## 6      Missense point NRAS    0.0781250 0.00000000
## 7      Missense point KRAS    0.0156250 0.00000000
## 8      Missense point TET2    0.0781250 0.00000000
## 9      Missense point EZH2    0.1093750 0.00000000
## 10      Missense point CBL    0.0468750 0.00000000
## 11     Missense point IDH2    0.0468750 0.00000000
## 12    Missense point SF3B1    0.0156250 0.00000000
## 13    Missense point CSF3R    0.0781250 0.00000000
## 14     Nonsense point TET2    0.1093750 0.00000000
## 15     Nonsense point EZH2    0.0312500 0.00000000
## 16    Nonsense point ASXL1    0.0781250 0.00000000
## 17    Nonsense point CSF3R    0.0468750 0.00000000
## 18   Pattern NRAS_xor_KRAS    0.0937500 0.00000000
## 19 Pattern SF3B1_xor_ASXL1    0.2343750 0.00000000
## 20   Pattern TET2_xor_IDH2    0.2656250 0.00000000
## 21    Pattern TET2_or_IDH2    0.2812500 0.00000000
## 22        Pattern XOR_TET2    0.2500000 0.00000000
## 23        Pattern XOR_EZH2    0.1562500 0.00000000
## 24         Pattern XOR_CBL    0.0859375 0.01104854
## 25       Pattern XOR_ASXL1    0.2187500 0.00000000
## 26        Pattern OR_CSF3R    0.1093750 0.00000000
```

```
as.kfold.posterr(model.boot)
```

```
## $capri_bic
##                 SELECTS             SELECTED MEAN.POSTERR SD.POSTERR
## 1 Missense point SETBP1 Nonsense point ASXL1     0.078125 0.00000000
## 2   Missense point TET2         Ins/Del EZH2     0.031250 0.02209709
## 3   Missense point TET2          Ins/Del CBL     0.015625 0.00000000
## 4  Nonsense point ASXL1      Pattern XOR_CBL     0.093750 0.00000000
## 5     Pattern XOR_ASXL1   Missense point CBL     0.046875 0.00000000
##
## $capri_aic
##                  SELECTS              SELECTED MEAN.POSTERR SD.POSTERR
## 1          Ins/Del ASXL1   Missense point KRAS     0.015625 0.00000000
## 2  Missense point SETBP1  Nonsense point ASXL1     0.078125 0.00000000
## 3  Missense point SETBP1  Nonsense point CSF3R     0.046875 0.00000000
## 4    Missense point TET2          Ins/Del EZH2     0.015625 0.00000000
## 5    Missense point TET2           Ins/Del CBL     0.015625 0.00000000
## 6    Nonsense point TET2   Missense point NRAS     0.078125 0.00000000
## 7    Nonsense point TET2 Pattern NRAS_xor_KRAS     0.093750 0.00000000
## 8   Nonsense point ASXL1       Pattern XOR_CBL     0.078125 0.02209709
## 9  Pattern TET2_xor_IDH2         Ins/Del ASXL1     0.140625 0.00000000
## 10      Pattern XOR_TET2   Missense point EZH2     0.109375 0.00000000
## 11     Pattern XOR_ASXL1    Missense point CBL     0.046875 0.00000000
```

Notice that these can be combined to create a nice table with all these statistics – we make here the example of a table with all the BIC statistics. This format can be readily exported to external spreadsheets for further visualization.

```
tabular = function(obj, M){
    tab = Reduce(
        function(...) merge(..., all = TRUE),
            list(as.selective.advantage.relations(obj, models = M),
                as.bootstrap.scores(obj, models = M),
```

```
                as.kfold.prederr(obj, models = M),
                as.kfold.posterr(obj,models = M)))

    # merge reverses first with second column
    tab = tab[, c(2,1,3:ncol(tab))]
    tab = tab[order(tab[, paste(M, '.NONPAR.BOOT', sep='')], na.last = TRUE, decreasing = TRUE), ]
    return(tab)
}

head(tabular(model.boot, 'capri_bic'))
```

```
##        capri_bic.SELECTS   capri_bic.SELECTED capri_bic.OBS.SELECTS
## 2    Missense point TET2         Ins/Del EZH2                    5
## 12 Missense point SETBP1 Nonsense point ASXL1                   14
## 22  Nonsense point ASXL1      Pattern XOR_CBL                    5
## 3      Pattern XOR_ASXL1   Missense point CBL                   14
## 1    Missense point TET2          Ins/Del CBL                    5
## 4                    <NA> Missense point CSF3R                   NA
##    capri_bic.OBS.SELECTED capri_bic.TEMPORAL.PRIORITY capri_bic.PROBABILITY.RAISING
## 2                       1                3.042779e-07                  1.981523e-04
## 12                      5                3.586517e-07                  1.575005e-07
## 22                      4                4.969218e-03                  1.868612e-04
## 3                       3                1.432328e-07                  5.704252e-08
## 1                       1                5.138164e-07                  4.930027e-06
## 4                      NA                          NA                            NA
##    capri_bic.HYPERGEOMETRIC capri_bic.NONPAR.BOOT capri_bic.STAT.BOOT
## 2              0.0781250000             100.00000                  100
## 12             0.0049513989             100.00000                  100
## 22             0.0009364534              66.66667                  100
## 3              0.0087365591              33.33333                  100
## 1              0.0000000000               0.00000                  100
## 4                        NA                    NA                   NA
##    capri_bic.MEAN.PREDERR capri_bic.SD.PREDERR capri_bic.MEAN.POSTERR
## 2                      NA                   NA               0.031250
## 12               0.078125                    0               0.078125
## 22               0.078125                    0               0.093750
## 3                0.046875                    0               0.046875
## 1                      NA                   NA               0.015625
## 4                0.078125                    0                     NA
##    capri_bic.SD.POSTERR
## 2            0.02209709
## 12           0.00000000
## 22           0.00000000
## 3            0.00000000
## 1            0.00000000
## 4                    NA
```

We finally show the plot of the model with the confidences by cross-validation.

```
tronco.plot(model.boot,
    fontsize = 12,
    scale.nodes = .6,
    confidence=c('npb', 'eloss', 'prederr', 'posterr'),
    height.logic = 0.25,
    legend.cex = .35,
    pathways = list(priors= gene.hypotheses),
    label.edge.size=10)
```

```
## *** Expanding hypotheses syntax as graph nodes:
## *** Rendering graphics
```

```
## Nodes with no incoming/outgoing edges will not be displayed.
## Annotating nodes with pathway information.
## Annotating pathways with RColorBrewer color palette Set1 .
## Adding confidence information: npb, eloss, prederr, posterr
## RGraphviz object prepared.
## Plotting graph and adding legends.
## Warning in strwidth(legend, units = "user", cex = cex, font = text.font):  font metrics unknown
for character 0xa
## Warning in strheight(legend, units = "user", cex = cex):  font metrics unknown for character
0xa
## Warning in text.default(x, y, ...):  font metrics unknown for character 0xa
## Warning in text.default(x, y, ...):  font metrics unknown for character 0xa
```

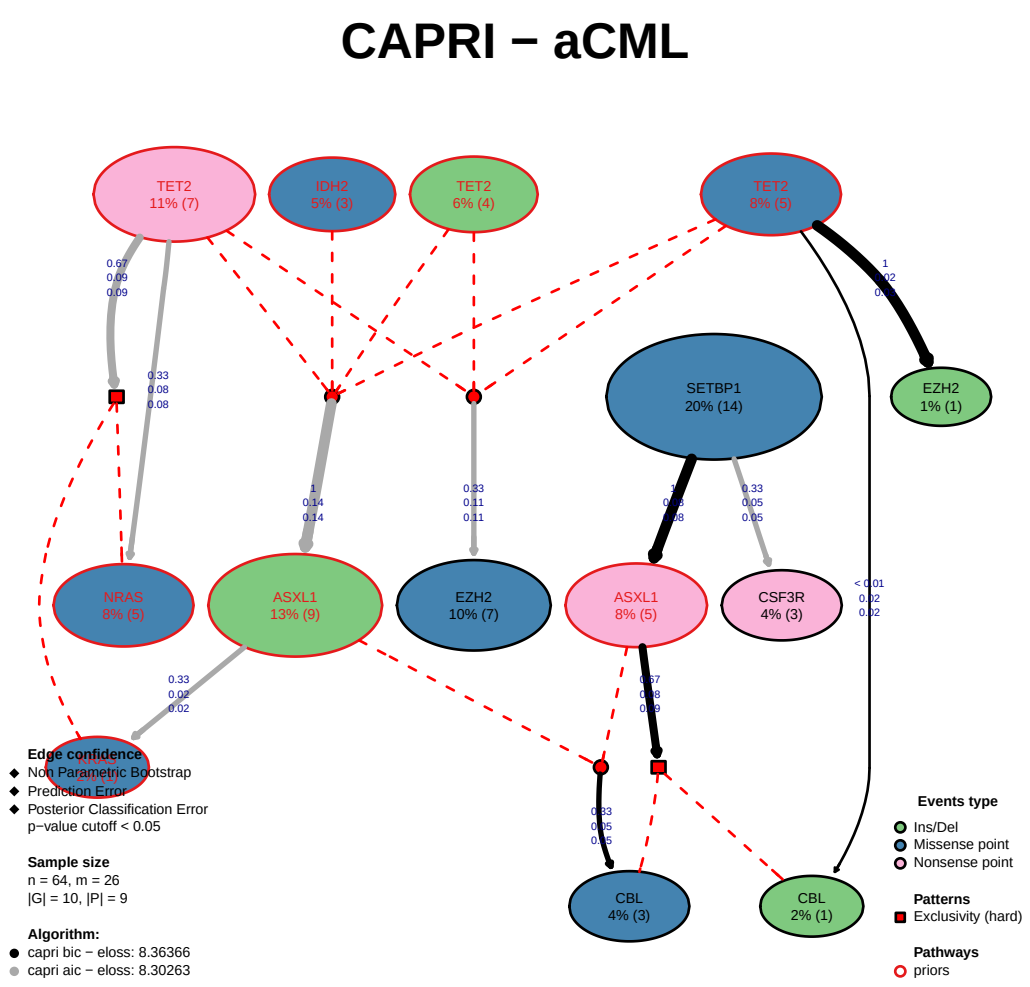

Figure 15: aCML model reconstructed by CAPRI with AIC/BIC as regolarizators and annotated with non-parametric, as well as with entropy loss, prediction and posterior classification errors computed via cross-validation. Edge thickness is proportional to the non-parametric scores.

8 Import/export to other tools

We implemented the interface of TRONCO with other tools to support the **Pipeline for Cancer Inference PiCnIc**, our attempt at devise an effective pipeline to extract ensemble-level cancer progression models from cross-sectional data (see **??**).

PiCnIc is versatile, modular and customizable and exploits state-of-the-art data processing and machine learning tools to:

1. identify tumor subtypes and then in each subtype;
2. select (epi)genomic events driving the progression;
3. identify groups of events that are likely to be observed as mutually exclusive;
4. infer progression models from groups and such data, and annotate them with associated statistical confidence.

The algorithms for cancer progression inference exploited by PicNiC are implemented within TRONCO, the other steps of the pipeline rely on dedicated tools (for clustering, drivers selection and exclusiviyt groups detection). The tools that PicNiC can exploit are of different nature, and we plan to interface them with TRONCO as far as our case studies are developed.

The current version of TRONCO supports input/output towards these tools:

1. Network Based Stratification (NBS), a method for stratification (clustering) of patients in a cancer cohort based on genome scale somatic mutations measurements and a gene interaction network. You can export a TRONCO object in the NBS input format with function `export.nbs.input`, clustering outputs can be handled with standard TRONCO functions.
2. MUTEX, a method for the identification of sets of mutually exclusive gene alterations in a given set of genomic profiles by scanning the groups of genes with a common downstream effect on the signaling network. You can export a TRONCO object in the MUTEX input format with function `export.mutex`, and output results can be imported with function `import.mutex.groups`.

Finally we also privede the possibility of exporting the inferred model to `graphML` format, which can be subsequently imported to *Cytoscape*. We now provide an example of such function.

```
export.graphml(model.boot,
    file = 'graph.gml',
    fontsize = 12,
    scale.nodes = .6,
    height.logic = 0.25)
```

```
## *** Expanding hypotheses syntax as graph nodes:
## *** Rendering graphics
## Nodes with no incoming/outgoing edges will not be displayed.
## Set automatic fontsize for edge labels:  6
## RGraphviz object prepared.
## Plotting graph and adding legends.
```

```
## Warning in strwidth(legend, units = "user", cex = cex, font = text.font):  font metrics unknown
for character 0xa
```

```
## Warning in strheight(legend, units = "user", cex = cex):  font metrics unknown for character
0xa
```

```
## Warning in text.default(x, y, ...):  font metrics unknown for character 0xa
```

```
## Warning in text.default(x, y, ...):  font metrics unknown for character 0xa
```

Follows an example of file generated by such function. Futhermore on the TRONCO website a *Cytoscape* style to visualize the progressions is available.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
         http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
```

```
  <!-- Created by igraph -->
  <key id="g_name" for="graph" attr.name="name" attr.type="string"/>
  <key id="g_models" for="graph" attr.name="models" attr.type="string"/>
  <key id="g_informations" for="graph" attr.name="informations" attr.type="string"/>
  <key id="v_name" for="node" attr.name="name" attr.type="string"/>
  <key id="v_label" for="node" attr.name="label" attr.type="string"/>
  <key id="v_type" for="node" attr.name="type" attr.type="string"/>
  <key id="v_fillcolor" for="node" attr.name="fillcolor" attr.type="string"/>
  <key id="v_fontcolor" for="node" attr.name="fontcolor" attr.type="string"/>
  <key id="v_bordercolor" for="node" attr.name="bordercolor" attr.type="string"/>
  <key id="v_shape" for="node" attr.name="shape" attr.type="string"/>
  <key id="v_width" for="node" attr.name="width" attr.type="double"/>
  <key id="v_height" for="node" attr.name="height" attr.type="double"/>
  <key id="v_fontsize" for="node" attr.name="fontsize" attr.type="double"/>
  <key id="v_borderwidth" for="node" attr.name="borderwidth" attr.type="double"/>
  <key id="e_weight" for="edge" attr.name="weight" attr.type="double"/>
  <key id="e_line" for="edge" attr.name="line" attr.type="string"/>
  <key id="e_arrow" for="edge" attr.name="arrow" attr.type="string"/>
  <key id="e_color" for="edge" attr.name="color" attr.type="string"/>
  <graph id="G" edgedefault="directed">
    <data key="g_name">CAPRI - aCML</data>
    <data key="g_models">CAPRI capri_bic - CAPRI capri_aic</data>
    <data key="g_informations">Generated with TRONCO v2.3.0</data>
    <node id="n0">
      <data key="v_name">CSF3R</data>
      <data key="v_label">CSF3R 4% (3)</data>
      <data key="v_type">Nonsense point</data>
      <data key="v_fillcolor">#FAB3D8</data>
      <data key="v_fontcolor">#000000</data>
      <data key="v_bordercolor">#000000</data>
      <data key="v_shape">ellipse</data>
      <data key="v_width">109.2</data>
      <data key="v_height">72.8</data>
      <data key="v_fontsize">18</data>
      <data key="v_borderwidth">1</data>
    </node>
    [...]
    <edge source="n1" target="n0">
      <data key="e_weight">1</data>
      <data key="e_line">Solid</data>
      <data key="e_arrow">True</data>
      <data key="e_color">#A9A9A9</data>
    </edge>
    [...]
  </graph>
</graphml>
```

9 sessionInfo()

```
toLatex(sessionInfo())
```

- R version 3.3.1 (2016-06-21), x86_64-apple-darwin13.4.0
- Locale: C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: TRONCO 2.4.3, gridExtra 2.2.1, knitr 1.14
- Loaded via a namespace (and not attached): BiocGenerics 0.18.0, BiocStyle 2.0.3, GlobalOptions 0.0.10, Matrix 1.2-7.1, R.matlab 3.6.0, R.methodsS3 1.7.1, R.oo 1.20.0, R.utils 2.4.0, RColorBrewer 1.1-2,

Rcpp 0.12.7, Rgraphviz 2.16.0, bnlearn 4.0, cgdsr 1.2.5, circlize 0.3.8, codetools 0.2-14, colorspace 1.2-6, compiler 3.3.1, doParallel 1.0.10, evaluate 0.9, foreach 1.4.3, formatR 1.4, gRapHD 0.2.4, graph 1.50.0, grid 3.3.1, gtable 0.2.0, gtools 3.5.0, highr 0.6, igraph 1.0.1, iterators 1.0.8, lattice 0.20-34, magrittr 1.5, munsell 0.4.3, parallel 3.3.1, plyr 1.8.4, scales 0.4.0, shape 1.4.2, stats4 3.3.1, stringi 1.1.1, stringr 1.1.0, tools 3.3.1, xtable 1.8-2