Extensions in the mistr World

Lukas Sablica[a] **and Kurt Hornik**[a]

[a] Institute for Statistics and Mathematics, WU Wirtschaftsuniversität Wien, Austria; https://www.wu.ac.at/en/statmath

This version was compiled on November 23, 2020

The main aim of this vignette is to introduce several available options for the package mistr. In the first place, we introduce the implementation of a distribution that is not directly supported by the framework, followed by a small example. Then we show how to add a new transformation and how this new transformation can be improved if some direct and invariant transformations are included. Note that this vignette serves as a guidebook for extensions in the mistr framework and does not cover examples for general purposes. These are more deeply described in the introduction vignette.

extensions | composite | mixture | R | tails | models | truncated | distributions

Adding new distribution

```
library(mistr)
```

While the framework provided by the mistr package currently supports all distributions that are included in the R **stats** package and many more as well, there are and always will be specific distributions that are not covered in the package code, and thus must be added by the user.

In such case, a new distribution can be added in a very simple way. Under the assumption that the [prefix][name] functions (d, p, q, and r) are loaded in the search path, the only additional function that is needed is the function that will create the object representing this random variable. This can be best explained on a concrete example. A function that creates an object for uniform distribution is designed as follows:

```
unifdist <- function(min = 0, max = 1){
  if (!is.numeric(min) || !is.numeric(max)){
    stop("parameters must be a numeric")
  }
  if (min >= max){
    stop("min must be smaller than max")
  }
  x <- list(parameters = list(min = min, max = max),
            type = "Uniform",
            support = list(from = min, to = max))
  class(x) <- c("unifdist", "contdist", "standist",
                "univdist", "dist")
  x
}
```

As the source code indicates, the only arguments of the function are the parameters. Another information that the user passes is the distribution family, which is set according to the function call. The rest of the information is fully determined by these two characterizations.

The class contains the mother class *dist* and a class *univdist*, which, unlike the class *trans_univdist*, expresses that the distribution is not transformed. Next class is the *standist* class that indicates that we are dealing with a standard distribution, and not a mixture or composite random variable. The class of *standist* is then split into continuous and discrete distributions, and this characterization is then stored as a next class. The last class is then the class of the distribution family. Thus, while the last three classes are present for internal purpose, the first two classes must be set by the user according to the new distribution.

To show also an example of a distribution that is not yet supported by **mistr**, we demonstrate the discrete uniform distribution. Discrete uniform distribution, also known as equally likely outcomes distribution, is a probability distribution where a finite number of outcomes are equally likely to be observed. Even though the distribution itself is non-parametric, it is broadly acceptable to represent its values by all integers in an interval $[min, max]$. This offers in a same way as the continuous uniform distribution a parametric representation using the two parameters, min and max. Using these two parameters, the cumulative distribution function and the probability density function are equal to

$$F(x) = \begin{cases} 0 & x < \min, \\ \frac{\lfloor x \rfloor - \min + 1}{\max - \min + 1} & \min \le x \le \max, \\ 1 & \max < x, \end{cases}$$

and

$$f(x) = \begin{cases} \frac{1}{\max - \min + 1} & x \in \{\min, \min+1, \ldots, \max\text{-1}, \max\}, \\ 0 & \text{else}, \end{cases}$$

respectively.

If these two functions together with the quantile function and a random sample function are available in the search path as a [prefix][name] function, e.g., ddunif(), pdunif(), qdunif(), and rdunif(), the function that will create an object can be defined as:

```
dunifdist <- function(min = 1, max = 6){
  if (!is.numeric(min) || !is.numeric(max)){
    stop("parameters must be a numeric")
  }
  if (min >= max){
    stop("min must be smaller than max")
  }
  if (min%%1 != 0 || max%%1 != 0){
    stop("min and max must be integers")
  }
  new_dist(name = "Discrete uniform",
           from = min, to = max, by = 1)
}
```

In this example the distribution is created using the help function new_dist(). If new_dist() is called from within the creator function, it takes only the name and support details of the distributions. Other specifications will be filled according to the parent

calls automatically. Note, that `new_dist()` can be called also directly from other functions and environments but in this case other arguments must be filled. For more details see the help file of `new_dist()`. The next important thing is that unlike the continuous distributions, the support information in the case of discrete distributions also contains the parameter by. This parameter describes the deterministic step between the support and for most known discrete distributions is equal to one, as they have support on the integers. It might of course differ for some distributions, which have support only on even numbers, or some scaled distributions. It is essential that this parametrization allows to perfectly define the support of a distribution, and hence allows to do more complicated operations and calculations. In the case the user would like a distribution with no equally distanced outcomes, one can perform a non-linear transformation. A final remark concerning the [prefix][name] functions for discrete distribution is that the d and p functions should have some rounding towards the support to avoid rounding errors. For distributions provided by the framework, this rounding is already implemented in the **stats** package calls.

Thus if the d p, q and r functions are reachable either from another package namespace or from the global environment

```
pdunif <- function(q, min = 0, max = 1,
                   lower.tail = TRUE,
                   log.p = FALSE){
  q <- round(q, 7)
  z <- ifelse(q < min, 0,
             ifelse(q >= max, 1,
             (floor(q) - min + 1)/(max - min + 1)))
  if(!lower.tail) z <- 1 - z
  if(log.p) log(z) else z
}
```

the distribution can be created and evaluated as:

```
D <- dunifdist(1, 6)
p(D, 4)
```

```
#  [1] 0.6666667
```

```
plim(D, 4)
```

```
#  [1] 0.5
```

Adding new transformations

The ability to perform transformations was already presented in the Introduction vignette. In this section we will cover how a new transformation can be added. The whole procedure will be described on an example, where an arcus tangent transformation will be implemented.

Arcus tangent, also known as atan, is a monotonic function on the whole support of the real numbers. Its inverse function is tangent (tan) and thanks to the nice one-to-one relationship from atan to tan and then back, atan can be easily implemented into current framework.

For each transformation, there is usually a need to create two new functions. One that performs transformations on yet untransformed distributions, and hence initializes the process (dispatches

on class *univdist*). Second that dispatches on class *trans_univdist* and is designed to add new transformation or upgrade or delete the previous one. Clearly, it is possible to add just one function that dispatches on class *dist* and hence works for both types, however, such approach does not contain enough flexibility for operations like inverse transformation of the previous one and is not recommended. The main function that is designed to make the life with the new transformation easier is the `trafo()` function. The function takes the object on which new manipulations will be performed, a specified type of manipulation and expressions that are used for this change. Thus, an `atan()` function that dispatches on untransformed distributions can be designed as follows.

```
atan.univdist <- function(x){
  trafo(x, type = "init",
        trans = quote(atan(X)),
        invtrans = quote(tan(X)),
        print = quote(atan(X)),
        deriv = quote(1+tan(X)^2),
        operation = "atan")
}
```

While the argument type in the `trafo()` call can be assigned with 4 different string values ("init", "new", "update", "go_back"), for the function that dispatches on *univdist* only type = "init" should be used. This initializes the history list that stores the information about the old transformations and assigns the first transformations according to the next four arguments. These (trans, invtrans, print and deriv) should be attached to the expressions that correspond to the transformation, inverse transformation, transformation that is used in print and the derivative of the inverse transformation, respectively. The last two arguments specify the name of the operation and (if any) additional value that was used in the transformation. In this example, function `atan()` is used for transformation, `tan()` as the inverse of `atan()`, again `atan()` for the print and since $\frac{d\tan(x)}{dx} = 1 + \tan(x)^2$, for the deriv argument such an expression is used. Additionally, name of the operation is added to be able to track and recognize it later. Furthermore, the argument value (not used here) can be assigned to a numeric if the function of a transformation contains two inputs such as multiplication or addition. This information can be later used for updating the transformation (i.e., $3+(2+X) = 5 + X$). Note that all expressions must use X as a placeholder to indicate the object in the transformation.

With this function we can now easily transform any distribution. An example of `arctan()` transformed standard normal distribution follows.

```
ataNorm <- atan(normdist())
library(ggplot2)
autoplot(ataNorm)
```

```
ataNorm
```
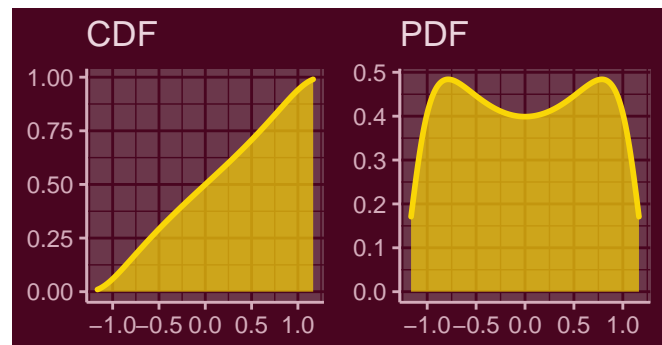
```
#     Trafo Distribution       Parameters
#   atan(X)        Normal mean = 0, sd = 1
```

This transformation is then fully able to cooperate with others.

```
log(2+ataNorm)
```

```
#                Trafo Distribution         Parameters
#   log(atan(X) + 2)        Normal mean = 0, sd = 1
```

Once the function for untransformed distributions is implemented, we can add also one for the transformed case. This function should offer more possibilities and it depends only on the creator how smart he wants the framework to be. For this particular case with `atan()`, we will assume that also `tan()` transformation is implemented. Even though `tan()` is not monotonic transformation, using the `sudo_support()` and modulus one can easily check if the transformation is performed only on the monotonic part of the support. Thus, if `tan()` is implemented also for the distribution, `atan.trans_univdist()` can be written as:

```
atan.trans_univdist <- function(x){
  if (last_history(x, "operation") == "tan") {
      return(trafo(x, type = "go_back"))
  } else {
      return(trafo(x, type = "new",
                     trans = quote(atan(X)),
                     invtrans = quote(tan(X)),
                     print = quote(atan(X)),
                     deriv = quote(1+tan(X)^2),
                     operation = "atan"))
  }
}
```

Here, the types "go_back" and "new" are used. The "go_back" is used when the code recognizes that the previous operation was the inverse and so rather cancels out both transformations. On the other hand, if there is no another way to eliminate or update a transformation, type = "new" adds a new transformation to the previous ones. This call again needs the expressions and operation name as in the example before. The last type, which was not used here is the "update" type, which can be used for updating previous transformations. For more details see the help file of `trafo()` or the source code of different transformations.

Finally, there is a possibility to add an invariant or direct transformation. This procedure is trivial and one only needs to create a call that dispatches on the distribution family rather than on the class *univdist*. For the discrete uniform distribution, which is invariant under linear transformation, this means that we can write the plus transformation as:

```
`+.dunifdist` <- function(e1, e2){
  if (is.dist(e1)) {
      O <- e1
      x <- e2
  } else {
      O <- e2
      x <- e1
  }
  dunifdist(min = parameters(O)["min"] + x,
```

```
      max = parameters(O)["max"] + x)
}
```

To summarize, we can perform two transformations on our new distribution, where one is invariant and the second one is the `atan()` transformation.

```
D2 <- atan(D+5)
D2
```

```
#     Trafo      Distribution            Parameters
#   atan(X) Discrete uniform min = 6, max = 11
```

```
p(D2, c(1.41, 1.43, 1.45, 1.47))
```

```
#   [1] 0.1666667 0.3333333 0.5000000 0.6666667
```

References

Bakar S, Nadarajah S, Kamarul Adzhar Z, Mohamed I (2016). "Gendist: An R Package for Generated Probability Distribution Models." *P L o S One*, **11**(6). ISSN 1932-6203. doi:10.1371/journal.pone.0156537.

Bolker B, Team RDC (2017). *bbmle: Tools for General Maximum Likelihood Estimation*. R package version 1.0.20, URL https://CRAN.R-project.org/package=bbmle.

Cooray K, Ananda MM (2005). "Modeling actuarial data with a composite lognormal-Pareto model." *Scandinavian Actuarial Journal*, **2005**(5), 321–334. doi:10.1080/03461230510009763. https://www.tandfonline.com/doi/pdf/10.1080/03461230510009763, URL https://www.tandfonline.com/doi/abs/10.1080/03461230510009763.

Kohl M, Ruckdeschel P (2010). "R Package distrMod: S4 Classes and Methods for Probability Models." *Journal of Statistical Software, Articles*, **35**(10), 1–27. ISSN 1548-7660. doi:10.18637/jss.v035.i10. URL https://www.jstatsoft.org/v035/i10.

Nadarajah S, Bakar S (2014). "New composite models for the Danish fire insurance data." *Scandinavian Actuarial Journal*, **2014**(2), 180–187. doi:10.1080/03461238.2012.695748. https://doi.org/10.1080/03461238.2012.695748, URL https://doi.org/10.1080/03461238.2012.695748.

Nadarajah S, Bakar SAA (2013). "CompLognormal: An R Package for Composite Lognormal Distributions." *The R Journal*, **5**(2), 97–103. URL https://journal.r-project.org/archive/2013/RJ-2013-030/index.html.

Ryan JA, Ulrich JM (2018). *quantmod: Quantitative Financial Modelling Framework*. R package version 0.4-13, URL https://CRAN.R-project.org/package=quantmod.

Scollnik DPM (2007). "On composite lognormal-Pareto models." *Scandinavian Actuarial Journal*, **2007**(1), 20–33. doi:10.1080/03461230601110447. https://doi.org/10.1080/03461230601110447, URL https://doi.org/10.1080/03461230601110447.

Wickham H (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN 978-3-319-24277-4. URL http://ggplot2.org.