R package for estimating parameters of some regression models with or without covariates using TensorFlow

Sara Garcés Céspedes

Universidad Nacional de Colombia
Facultad de Ciencias, Escuela de Estadística
Medellín, Colombia
2021

Propuesta de un paquete en **R** para la estimación de parámetros de algunos modelos de regresión con y sin covariables usando TensorFlow

Sara Garcés Céspedes

Tesis presentada como requisito parcial para optar al título de:
Magister en Ciencias Estadística

Director:
Freddy Hernández Barajas, Ph.D.

Universidad Nacional de Colombia
Facultad de Ciencias, Escuela de Estadística
Medellín, Colombia
2021

To my parents Gloria and Francisco

Acknowledgments

I want to thank the department of statistics of the Universidad Nacional de Colombia sede Medellín and specially its professors for contributing to my personal and professional growth throughout these past years. Also, I would like to express my sincere gratitude to professor Freddy Hernández Barajas, my dissertation advisor for his support, for motivating me to work hard every day and for contributing significantly with his ideas and advice to the development of this research work.

I also want to express my deep gratitude to my parents Gloria and Francisco, my brother Mateo and my sister Laura who have been always by my side and have supported me unconditionally. They are and will always be my greatest gift. Finally, I want to thank my boyfriend Tomas who has taught me to believe in myself and has given me the strength to finish this work.

Abstract

The task of estimating parameters is very important in both scientific and industrial applications. The R programming language provides a wide variety of functions created to find the maximum likelihood estimates of parameters from distributions and regression models. In this work the **estimtf** package with its main functions `mle_tf` and `mlereg_tf` are presented. This package was design with the aim of finding the maximum likelihood estimates of distributional and regression parameters using TensorFlow, an open-source library for numerical computation created by Google. To achieve this goal an iterative estimation process was design in which the TensorFlow optimizers are used to maximize the likelihood function. To illustrate the use of the **estimtf** package and evaluate the performance of the estimation process, a simulation study was performed as well as some applications using real datasets. From the simulation study, an impact of the sample size, the selected optimizer and the initial value of the learning rate on the estimates obtained with the `mle_tf` and the `mlereg_tf` functions was observed. Additionally, the estimates obtained with both functions were very close to the real value of the parameters and very similar to the estimates obtained with other R functions that are very popular and widely used for estimating parameters.

Keywords: TensorFlow, estimation of parameters, maximum likelihood, optimization algorithms.

Resumen

La tarea de estimar parámetros es muy importante tanto en aplicaciones científicas como de industria. El lenguaje de programación R provee una amplia variedad de funciones creadas para encontrar los estimadores de máxima verosimilitud de parámetros de distribuciones y de modelos de regresión. En este trabajo se presenta el paquete **estimtf** junto con sus principales funciones `mle_tf` y `mlereg_tf`. Este paquete fue diseñado con el objetivo de encontrar los estimadores de máxima verosimilitud de parámetros distribucionales y de regresión usando TensorFlow, una librería de código abierto para computación numérica creada por Google. Para alcanzar este objetivo se diseñó un proceso de estimación iterativo en el cual se utilizan los optimizadores incluidos en esta librería para maximizar la función de verosimilitud. Para ilustrar el uso del paquete **estimtf** y evaluar el desempeño del proceso de estimación, se llevó a cabo un estudio de simulación y se presentaron algunas aplicaciones usando bases de datos reales. A partir del estudio de simulación se observó que el tamaño de muestra, el optimizador seleccionado y el valor inicial de la tasa de aprendizaje afectan las estimaciones obtenidas con las funciones `mle_tf` y `mlereg_tf`. Adicionalmente, las estimaciones obtenidas con ambas funciones resultaron muy cercanas a los verdaderos valores de los parámetros y muy similares a las estimaciones obtenidas con otras funciones de R, las cuales son muy populares y comúnmente usadas para la estimación de parámetros.

Palabras clave: TensorFlow, estimación de parámetros, máxima verosimilitud, algoritmos de optimización.

Contents

List of figures

List of tables

1. Introduction

In 1922, Fisher stated that one of the fundamental problems of statistics is the problem of estimation, which involves the choice of the best method to estimate the parameters of a population (Fisher, 1922). Since then and even before, multiple methods have been developed with the objective of solving this problem. From the frequentist point of view, some of the most popular and used methods are the method of moments (Pearson, 1936), the maximum likelihood method (Fisher, 1922) and the least squares method (Legendre, 1805). In the literature we find authors interested in comparing the performance of these and other estimation methods such as Bakouch, Dey, Ramos, and Louzada (2017) that compared the maximum likelihood method, the method of moments, the percentile based estimation, among others, in the estimation of parameters from a binomial-exponential 2 distribution. On the other hand, Louzada, Ramos, and Perdoná (2016) compared the maximum likelihood method, the method of moments, the method of modified moments, the ordinary and weighted least squares methods and others, in the estimation of the parameters of the extended exponential geometric distribution and Ling (2018) used the maximum likelihood and the least-squares approaches to find the estimates of the parameters of the generalized gamma distribution.

In particular, the maximum likelihood method consists of estimating the parameters of a distribution by maximizing the likelihood function. This method is widely used because it can be easily implemented and the resulting estimates have many desirable statistical properties such as consistency and efficiency. The development of computational tools have facilitated the implementation of this and other estimation methods. One of these tools is R (R Core Team, 2021) which is an open source language and environment for statistical computing created by Ihaka and Gentleman (1996) that provides a wide variety of statistical and graphical techniques. Because R is an open source language, its users can contribute to its growth and extension by creating packages which are collections of functions and data sets. By June 2019, there were more than 14.000 packages available on the Comprehensive R Archive Network (Wickham, 2015). Some of these packages include functions that allow R users to implement estimation methods like the ones mentioned above and more. In the case of the maximum likelihood method, R provides some packages such as **bbmle**, **stats4** and **EstimationTools** that contain functions especially design to facilitate the implementation of optimization methods like the Nelder-Mead method, Newton-type methods, quasi-Newton methods, among others, in maximum likelihood estimation.

R also includes a package called **tensorflow**, an interface to TensorFlow which is a library for numerical computation created by Google. Since the launch of TensorFlow as an open source library in 2016, many researchers have used this tool for experimentation with machine learning methodologies, thus contributing to its improvement and making this library one of the most popular today. The majority of the projects that use TensorFlow are related to image recognition, object detection and prediction, such as the one developed by Sawant, Bhandari, Yadav, Yele, and Bendale (2018), who implemented a convolutional neural network with 5 layers in TensorFlow to detect cancer cells in the brain through MRI. On the other hand, Nesterov (2014) used TensorFlow for the recognition of handwritten with a convolutional neural network with 2 layers. Another very popular application is speech recognition which consists of the identification of words and phrases in a spoken language. Variani, Bagby, McDermott, and Bacchiani (2017) used 20.000 hours of spontaneous speaking by anonymous voices to train a TensorFlow model design for voice recognition. Other interesting application involving the use of TensorFlow is the one presented by Do, Son, and Chaudri (2017), who used TensorFlow and a database of hospitalized patients to predict the severity of asthma in those patients.

One of the main components of TensorFlow are its optimizers which are algorithms that use gradient-based numerical optimization to update the parameters of a function with the aim of minimizing it (Zeiler, 2012). Some of the optimizers included in the TensorFlow library are the Adam optimizer, the momentum optimizer, the gradient descent optimizer and the Adagrad optimizer which are mainly used to train neural networks models, however, they can also be implemented to find the maximum likelihood estimates of distributional and regression parameters.

The main goal of this work is to present and evaluate the performance of **estimtf**, an R package created by Garcés and Hernández (2021) that contains functions that allow R users to find the maximum likelihood estimates of distributional and regression parameters using the TensorFlow optimizers. In these functions we designed and implemented an iterative estimation process that uses a TensorFlow optimizer to maximize the log-likelihood function. This process requires to define the distribution of interest, the parameters to be estimated, an initial value for these parameters, a TensorFlow optimizer with its respective hyperparameters and a maximum number of iterations. Also, this package allows to estimate parameters from distributions which are not necessarily implemented in R and do it in a simple and intuitive way. Through this work we intent to demonstrate that by using the **estimtf** package we can obtain accurate estimates and to compare these estimates with the ones obtained with other optimization methods available in R.

The main motivation for creating the **estimtf** package is to take advantage of a powerful tool such as the TensorFlow library which is freely available for use and has multiple useful tools for numerical optimization to carry out a very important and frequently performed task in statistics, the estimation of parameters. Also, to evaluate the performance of TensorFlow optimization algorithms, which are very popular in the context of machine learning applications, in solving the maximum likelihood estimation problem.

This document includes 7 chapters. In chapter 2, the main theoretical aspects associated with the estimation of distributional and regression parameters are presented, including the most popular estimation methods and their implementation in R. In chapter 3, the TensorFlow library and its optimizers are introduced. In chapter 4, the **estimtf** package and its main functions with an example of their implementation in R are presented. Chapter 5 contains a simulation study designed to determine the effect that the selected TensorFlow optimizer, the learning rate and the sample size have in the performance of the estimation process implemented in the **estimtf** package. Chapter 6 contains 5 applications with real data. Finally, in chapter 7 our main conclusions and recommendations are presented.

2. Estimation of distributional and regression parameters

There are multiple methods for estimating unknown distributional and regression parameters. Particularly, from the frequentist point of view, some of the most used methods are presented below.

2.1. Method of moments

The method of moments was developed by the mathematician Karl Pearson in the late 1800s (Pearson, 1936). The main idea behind this method is to equate some certain sample characteristics, such as the mean or variance, to the corresponding expected population values and solve these equations for the unknown parameters (Devore, 2016). Let $X_1, X_2, \ldots, X_n$ be a random sample from a distribution with density/mass function $f(x|\theta_1, \theta_2, \ldots, \theta_s)$, where $\theta_1, \theta_2, \ldots, \theta_s$ are unknown parameters. The moment estimators are obtained by equating the first s sample moments to the corresponding first s population moments and solving for $\theta_1, \ldots, \theta_s$:

$$E\left(X^s\right) = m_s,$$

where $E\left(X^s\right)$ is the s-th population moment and m_s is the s-th sample moment. The method of moments requires the distribution to have finite moments and the first few moments to be known. This method often requires less computation than maximum likelihood method (Devore, 2016). Finally, the moment estimators are often used as starting values when searching for maximum likelihood estimates.

2.2. Least squares method

The least squares method was first published by Legendre (1805) followed by a statement in 1809 by the mathematician Carl Friedrich Gauss claiming to have used this method since

1795 (Stigler, 1981). It consists of estimating parameters by minimizing the squared distance between the observed data and the expected values and is most commonly used in linear regression (Little, 2014). In the context of a regression problem, this method allows obtaining the regression parameter estimates by minimizing the sum of the squared deviations between the data and a regression model (Devore, 2016). Let $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)^\top$ be n independent observations with $\mu_i = E(y_i)$ and let $\boldsymbol{X} = (x_{ij})$ denotes the $n \times p$ model matrix, where x_{ij} is the value of explanatory variable j for observation i, the ordinary linear model is:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon},$$

where $\boldsymbol{\beta}$ is a $p \times 1$ parameter vector with $p \leq n$ and $\boldsymbol{\epsilon}$ is an error term with $E(\boldsymbol{\epsilon}) = \mathbf{0}$ and covariance matrix $\boldsymbol{V} = var(\boldsymbol{\epsilon}) = \sigma^2 \boldsymbol{I}$ (Agresti, 2015). To obtain parameter estimates $\hat{\boldsymbol{\beta}}$ that best satisfy the linear model, the least squares method determines the values of $\beta_1, \beta_2, \ldots, \beta_p$ that minimizes the sum of the squares deviations:

$$\textstyle\sum_{i=1}^{n}(y_i - \hat{\mu}_i)^2 = \sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p}\hat{\beta}_j x_{ij}\right)^2.$$

Assuming that the model matrix $\boldsymbol{X}$ has full rank p, the least squares estimator of $\boldsymbol{\beta}$ is:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1}\boldsymbol{X}^\top \boldsymbol{y}.$$

2.3. Maximum Likelihood Estimation

The method of maximum likelihood was first introduced by Fisher (1922). This method consists of finding the values of the distribution parameters that maximize the likelihood function (Wilks, 2019). This method is frequently used, especially when working with large sample sizes because the resulting estimators known as Maximum Likelihood Estimates (MLE) have desirable efficiency properties such as (Devore, 2016):

- The MLE of any parameter θ is approximately unbiased, that is, $E[\hat{\theta}] \approx \theta$.
- The MLE of any parameter θ has a variance as small as or nearly as small as can be achieved by any estimator, which means that $\hat{\theta}$ is approximately the minimum-variance unbiased estimator (MVUE) of θ.
- The MLE $\hat{\theta}$ is asymptotically normally distributed (Sweeting, 1980).

Let $X_1, X_2, \ldots, X_n$ be a random sample with mass/density function $f(x|\theta_1, \ldots, \theta_s)$. The likelihood function $L(\boldsymbol{\theta}|\boldsymbol{x})$ of these random variables is defined as the joint density (Rizzo, 2007):

$$L(\boldsymbol{\theta}|\boldsymbol{x}) = \prod_{i=1}^{n} f(x_i|\boldsymbol{\theta}),$$

where, $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^\top$ and $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_s)^\top$. Therefore, a MLE of $\boldsymbol{\theta}$ is a value $\hat{\boldsymbol{\theta}}_{MLE}$ that maximizes $L(\boldsymbol{\theta}|\boldsymbol{x})$, that is:

$$\hat{\boldsymbol{\theta}}_{MLE} = \underset{\boldsymbol{\theta}\in\Theta}{\arg\max}\ L(\boldsymbol{\theta}|\boldsymbol{x}),$$

where Θ is the parameter space. In practice, it is easier to maximize the log-likelihood function $\ell(\boldsymbol{\theta}|\boldsymbol{x})$, which is defined as the logarithm of $L(\boldsymbol{\theta}|\boldsymbol{x})$ (Hernández & Usuga, 2019):

$$\ell(\boldsymbol{\theta}|\boldsymbol{x}) = \log L(\boldsymbol{\theta}|\boldsymbol{x}) = \sum_{i=1}^{n} \log f(x_i|\boldsymbol{\theta}).$$

The maximum likelihood method starts by defining the score function $S(\boldsymbol{\theta})$ as the first derivative vector of the log-likelihood, assuming $\ell(\boldsymbol{\theta}|\boldsymbol{x})$ is differentiable:

$$S(\boldsymbol{\theta}) \equiv \tfrac{\partial}{\partial\boldsymbol{\theta}}\ell(\boldsymbol{\theta}|\boldsymbol{x}).$$

Then, the MLE $\hat{\boldsymbol{\theta}}$ is obtained from the solution of the score equation (Pawitan, 2013):

$$S(\boldsymbol{\theta}) = \mathbf{0},$$

where $\hat{\boldsymbol{\theta}}$ may be a relative maximum, a relative minimum or an inflection point of the log-likelihood function $\ell(\boldsymbol{\theta}|\boldsymbol{x})$ (Rizzo, 2007).

2.3.1. Standard error and Wald statistic

The second derivative of the log-likelihood function contains information about the variance of $\hat{\boldsymbol{\theta}}$ (Rizzo, 2007). The *Fisher information matrix* $I(\boldsymbol{\theta})$ is the expectation of a $s \times s$ matrix of second derivates of the likelihood with respect to $\boldsymbol{\theta}$, whose elements are defined as:

$$I_{ij}(\boldsymbol{\theta}) \equiv -E\left[\tfrac{\partial^2}{\partial\theta_i\partial\theta_j}\ell(\boldsymbol{\theta})\right].$$

When evaluated at the MLE, $I(\hat{\boldsymbol{\theta}})$ is known as the *observed Fisher information* (Pawitan, 2013). Furthermore, the standard errors of the parameters in $\hat{\boldsymbol{\theta}}$ are the square roots of the diagonal terms in the variance-covariance matrix $var(\hat{\boldsymbol{\theta}})$ defined as:

$$var(\hat{\boldsymbol{\theta}}) = \left[I(\hat{\boldsymbol{\theta}})\right]^{-1},$$

therefore, the standard errors are computed as follows:

$$se(\hat{\boldsymbol{\theta}}) = diag\left(\left[I(\hat{\boldsymbol{\theta}})\right]^{-1/2}\right).$$

The main use of the standard error is to test the null hypothesis $H_0 : \theta = \theta_0$ using the Wald statistic (Pawitan, 2013):

$$z = \frac{\hat{\theta}-\theta_0}{se(\hat{\theta})}.$$

Also, it is used to compute the Wald confidence intervals (Pawitan, 2013). The Wald 95% confidence interval for θ is:

$$\hat{\theta} \pm 1.96 se(\hat{\theta}).$$

2.3.2. Optimization algorithms for Maximum Likelihood Estimation

The Maximum Likelihood Estimation problem can be expressed as a non-linear optimization problem (Mai Anh, Bastin, & Frejinger, 2014):

$$\min_{x \in \mathbb{R}^n} f(x),$$

where $f(x)$ is the objective function. For maximum likelihood estimation this function represents the negative log-likelihood $f(x) = -\ell(x)$. According to Rizzo (2007), in finding a solution some problems may arise such as:

- The derivatives of the likelihood function do not exist or do not exist on all of the parameter space Θ.
- The optimal value of $\boldsymbol{\theta}$ is not an interior point of Θ.
- The likelihood equation is difficult to solve.

Taking into account these problems, optimization algorithms are of great help in trying to find the optimal solution $\hat{\boldsymbol{\theta}}$. All optimization algorithms begin with an initial guess of the unknown variable or variables and generate estimates known as iterates until the process finishes. What differentiates one algorithm from another is the strategy and the information used to move from one iterate to the next. These algorithms are expected to perform well for reasonable and arbitrary values of the starting point and to do it as quickly as possible and without requiring excessive computational resources (Nocedal & Wright, 2006).

There are multiple optimization algorithms that can be used to solve the MLE problem of which, the best known and used are the Newton-Raphson algorithm, the Fisher-Scoring algorithm and the Expectation–Maximization algorithm.

The Newton-Raphson algorithm

The Newton-Raphson algorithm is an iterative procedure commonly used to find maximum likelihood estimates. As the log-likelihood functions are close to quadratic functions around their maximum points in many cases, a quadratic approximation of these functions is very convenient (Storvik, 2011). Therefore, in the case of maximum likelihood estimation, the main idea behind this algorithm is to approximate the log-likelihood function by a quadratic function using a Taylor series expansion (Millar, 2011).

Given the k-th estimate $\hat{\boldsymbol{\theta}}_k \in \boldsymbol{R}^s$ of the vector of true parameters $\boldsymbol{\theta}$, the Newton-Raphson algorithm is defined by the iteration:

$$\hat{\boldsymbol{\theta}}_{k+1} = \hat{\boldsymbol{\theta}}_k - \left[\boldsymbol{H}(\hat{\boldsymbol{\theta}}_k)\right]^{-1} S(\hat{\boldsymbol{\theta}}_k),$$

where $\boldsymbol{H}(\hat{\boldsymbol{\theta}}_k)$ is the $s \times s$ Hessian matrix of second-order partial derivatives of the log-likelihood function evaluated at $\hat{\boldsymbol{\theta}}_k$ (Millar, 2011). Even though this method is widely used and is very efficient in optimizing functions that are not to far from quadratic, it may fail when applying it in more complex problems mainly because the Hessian matrix may not be positive-definite. On the other hand, when the number of parameters is large or the likelihood function is very complex, this algorithm can be time consuming (Commenges, Jacqmin-Gadda, Proust-Lima, & Guedj, 2006).

The Fisher-Scoring algorithm

The Fisher-Scoring algorithm is a variation of the Newton-Raphson algorithm which instead of using the Hessian matrix itself, it uses the *Fisher information matrix* $\boldsymbol{I}(\boldsymbol{\theta}) = -E\left[\boldsymbol{H}(\boldsymbol{\theta})\right]$ (Agresti, 2015). The Fisher-Scoring algorithm is defined by the iteration:

$$\hat{\boldsymbol{\theta}}_{k+1} = \hat{\boldsymbol{\theta}}_k + \left[\boldsymbol{I}(\hat{\boldsymbol{\theta}}_k)\right]^{-1} S(\hat{\boldsymbol{\theta}}_k),$$

where $\boldsymbol{I}(\hat{\boldsymbol{\theta}})$ is the *Fisher information matrix* evaluated at $\hat{\boldsymbol{\theta}}_k$ (Storvik, 2011). Even though there are some cases as in generalized linear models in which $\boldsymbol{I}(\boldsymbol{\theta})$ is easily computed, this is not the case for all optimization problems and therefore the Fisher-scoring algorithm can not be easily implemented (Commenges et al., 2006).

The Expectation–Maximization algorithm

The Expectation–Maximization (EM) algorithm was proposed by Dempster, Laird, and Rubin (1977). This algorithm is an optimization method used to find maximum likelihood estimates particularly when some parts of the data are missing (Rizzo, 2007).

Given a statistical model that generates two sets of random variables, $\boldsymbol{Y}$ and $\boldsymbol{B}$, where only y is observed, and a vector of unknown parameters $\boldsymbol{\theta}$, the EM algorithm consists of the following two steps:

- **Expectation step:** Compute the conditional expected value:

$$Q(\boldsymbol{\theta}|\boldsymbol{\theta}_k) = E_{\theta_k}[\log L(\boldsymbol{\theta}; \boldsymbol{y}, \boldsymbol{B})|\boldsymbol{y}],$$

  where $\boldsymbol{\theta}_k$ is the value of $\boldsymbol{\theta}$ at iteration k.

- **Maximization step:** Maximize $Q(\boldsymbol{\theta})$ to obtain a new parameter estimate $\boldsymbol{\theta}_{k+1}$ and return to the Expectation step using the updated value. Repeat until convergence.

The EM algorithm is reliable at finding a global maximum and its implementation is relatively easy, however the convergence can be very slow (Rizzo, 2007).

2.3.3. Maximum likelihood estimation in R

In the R programming language, there are multiple general-purpose optimization tools designed to implement and simplify the maximum likelihood estimation method. The **base** package in R has the `optim` function which includes five optimization tools: (1) `Nelder-Mead` based on the Nelder-Mead method (J. Nelder & Mead, 1965), (2) `BFGS` which is a quasi-Newton method, (3) `L-BFGS-B` which is a quasi-Newton method that allows setting lower and/or upper bounds to each parameter (Byrd, Lu, Nocedal, & Zhu, 1995), (4) `CG` based on conjugate-gradient algorithm and (5) `SANN`, a variant of simulated annealing which is an stochastic global optimization method (Bélisle, 1992). This package also has the `nlm` function (Schnabel, Koonatz, & Weiss, 1985) which performs the minimization of functions using a Newton-type algorithm and the `nlminb` function that implements PORT (portable Fortran programs for numerical computation) routines for unconstrained and box-constrained optimization (Fox, Hall, & Schryer, 1978). Most of the methods mentioned above date from the 1970s to 1980s, however, these functions are still considered very good tools and they are still widely used even though there are more recent tools that include features that these base functions do not include (Nash, 2014).

Furthermore, there are multiple packages especially design to facilitate the implementation of some of the optimization tools mentioned above. The `mle` function included in the **stats4** package uses `optim` to estimate parameters via maximum likelihood. The **bbmle** package

Bolker and R Development Core Team (2020) includes the `mle2` function which is more robust and has additional warnings and arguments compared to the `mle` function. The **maxLik** package (Henningsen & Toomet, 2011) provides an unified interface for multiple optimization routines and unlike the two previous packages that are based on `optim`, the **maxLik** package includes an option to use the Newton-Raphson algorithm. On the other hand, the **EstimationTools** package (Mosquera & Hernandez, 2019) includes optimization procedures such as `optim`, `nlminb` and `DEoptim`. This last procedure implements the differential evolution algorithm for parameter estimation using the maximum likelihood method (Mullen, Ardia, Gil, Windover, & Cline, 2011).

Finally, it is important to mention that R also includes very useful functions for estimation of parameters from regression models. The `lm` function is one of the most popular functions for parameter estimation and is used to fit linear models through the method of least squares. Other frequently used function is `glm`, created to fit generalized linear models (J. A. Nelder & Wedderburn, 1972). The default method for fitting the models uses Iteratively Reweighted Least Squares (IWLS). R also includes the **gamlss** package for fitting GAMLSS, that is, Generalized Additive Models for Location Scale and Shape (Rigby & Stasinopoulos, 2005). GAMLSS are univariate distributional regression models where all the parameters of the distribution of the response variable can be modelled as functions of the explanatory variables (Stasinopoulos, Rigby, Heller, Voudouris, & De Bastiani, 2017).

3. TensorFlow

TensorFlow is an open source software library for numerical computation that uses directed graphs (Galeone, 2019). This library was originally developed by the Google Brain Team and is mainly used for machine learning and deep neural networks research. However, TensorFlow can be applied in a wide variety of other domains like numerical optimization, providing tools that simplify the optimization process especially when working with gradient descent methods (RStudio, 2020). Although the core TensorFlow library is implemented in `C++`, there are a wide variety of APIs available in several languages like Python, Java, JavaScript, MATLAB and `R`. Through these APIs, users of these programming languages can construct and execute TensorFlow graphs (TensorFlow, 2020).

3.1. TensorFlow graphs

As mentioned before, TensorFlow uses directed graphs also known as data flow graphs to represent computations. A directed graph is a set of nodes connected with edges that have a direction associated with them. Nodes in these data flow graphs represent mathematical operations while the edges represent multidimensional data arrays called tensors on which the operations are performed (Abadi et al., 2016). Representing computations through data flow graphs allows for parallelism, that is, allows to execute operations that do not depend on each other simultaneously. Depending on the structure of the graph, TensorFlow schedules the operations and execute them in the most efficient manner. Also, TensorFlow is capable of executing these operations in various hardware platforms like GPUs or CPUs on a single machine or distribute the execution across multiple machines (Abadi et al., 2016).

When working with TensorFlow there are two main steps. First, it is required to build a data flow graph and then create a TensorFlow Session to execute operations and evaluate tensors. The data flow graph contains all or some of the following elements:

- **Variables:** TensorFlow variables represent changeable parameters, that is, variables are defined when values need to be updated at any point in time. Also, they have to be initialized before running the data flow graph. When estimating parameters from probability distributions or regression models, the distributional parameters and the regression parameters must be defined as variables.

- **Placeholders:** A placeholder is a especial tensor used to feed the data into the graph. They do not need to be initialized as the TensorFlow variables, however, it is required to specify the type of data to be feed, that is, integer, float, among others. Placeholders are typically used for feeding response or explanatory variables.
- **Constants:** TensorFlow constants represent parameters that cannot be change. When defining a constant it is required to specify its value.
- **Operations:** TensorFlow operations are graph nodes through which the tensors flow while performing mathematical operations on them.

Below is a simple program written in R programming language in which TensorFlow constant addition and multiplication are performed:

```
# Build data flow graph
x <- tf$constant(5.0, dtype =  tf$float32)
y <- tf$constant(12.0, dtype =  tf$float32)
z <- tf$constant(8.0, dtype =  tf$float32)
product <- tf$multiply(x, y)
sum <- tf$add(x, y)
result <- tf$add(product, sum)

# Create session and run the graph
sess <- tf$compat$v1$Session()
sess$run(result)
```

Code 3.1: Addition and multiplication of constants with TensorFlow

In Figure **3-1**, it is possible to observe the data flow graph that represents the program presented above. As seen in the graph, the output of one node becomes the input of another node and the data flows from one node to the other through the edges. To estimate distributional or regression parameters the graph must also include a loss measure and an optimization method which will be discussed later in this chapter.

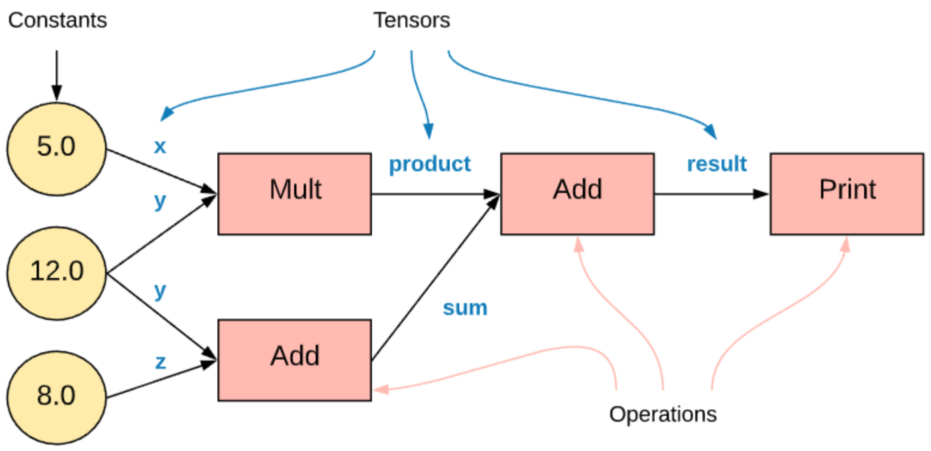

Figure 3-1.: Data flow graph representing a program in which some operations are performed on TensorFlow constants.

3.2. Eager execution

Even though TensorFlow was initially created to used directed graphs to represent computations, it also includes a programming environment known as Eager execution in which operations are evaluated immediately without the need of building a graph. In this case, TensorFlow calculates the values of tensors as they appear in the code. When working in eager mode, elements like graphs, sessions, placeholders, variable initialization, among others, are not longer required. The main advantages of eager execution are easier debugging processes and the possibility to implement the functionalities of the host language. Also, almost all of the TensorFlow operations are available in eager execution mode (TensorFlow, 2020). The latest versions of TensorFlow come with eager execution by default, however `tf$compat$v1$disable_eager_execution()` function can be used to disable the eager execution mode and work with TensorFlow graphs.

3.3. Estimation process with TensorFlow

As mentioned previously, our main goal is to estimate parameters from probability distributions and linear regression models via maximum likelihood using the TensorFlow library. To do so, we designed an estimation process in which the optimizers included in this library are used to minimize a loss or cost function defined in this case as the negative log-likelihood function:

$$C(\boldsymbol{\theta}) = -\log L(\boldsymbol{\theta}),$$

where $L(\boldsymbol{\theta})$ corresponds to the likelihood function that depends on the distribution of interest when estimating parameters from probability distributions or on the distribution of the response variable when estimating the regression parameters from regression models. A lot of the optimization algorithms included in the TensorFlow library use gradient-based numerical optimization to deal with these optimization problems.

3.3.1. Gradient-based numerical optimization

The aim of numerical optimization is to optimize an objective function $C(\boldsymbol{\theta})$ by updating a set of parameters $\boldsymbol{\theta}$. This optimization process involves an iterative procedure that applies changes to the parameters at each iteration of the algorithm (Zeiler, 2012):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \triangle\boldsymbol{\theta}_t,$$

where $\triangle\boldsymbol{\theta}_t$ represents the change applied to the parameters in iteration t and is called *learning step*. Some of these methods use the gradient as the core element for the update of the parameter vector $\boldsymbol{\theta}$ including the majority of the TensorFlow optimizers (Bengio, 2012). In these cases, TensorFlow becomes a very powerful tool because it includes automatic differentiation techniques that efficiently computes the gradients (TensorFlow, 2020).

The gradient descent algorithms seek to minimize an objective or cost function $C(\boldsymbol{\theta})$ by updating the parameters in the opposite direction of the gradient of the objective function with respect to the parameters as follows (Zeiler, 2012):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha\tfrac{\partial C(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}_t},$$

where α is a hyperparameter called the *learning rate* that controls how large is the step taken in the opposite direction of the gradient and takes a small positive value, usually between 0 and 1. Choosing a value for this hyperparameter is crucial because a small learning rate may result in a long optimization process as it requires more iterations given the small changes made to the parameters in each update. On the contrary, a large learning rate requires fewer iterations but can cause the algorithm to converge quickly to a sub-optimal solution. Some authors have proposed various strategies to determine the optimal value for the learning rate. One of them is to start with a large learning rate and if the algorithm diverges, try again with a learning rate that is three times smaller and repeat this process until no divergence is observed. Other strategies consists of using a decreasing learning rate, that is, a learning rate that decreases after every step or a learning rate that remains constant during the first steps and at some point starts to decrease after every step following some schedule (Bengio, 2012).

In Figure **3-2**, it is possible to observe how the gradient descent algorithm minimizes a loss function $C(\theta)$. Starting in an initial value of θ, the algorithm applies changes to the parameter in each iteration until it reaches the value of θ that minimizes the loss function.

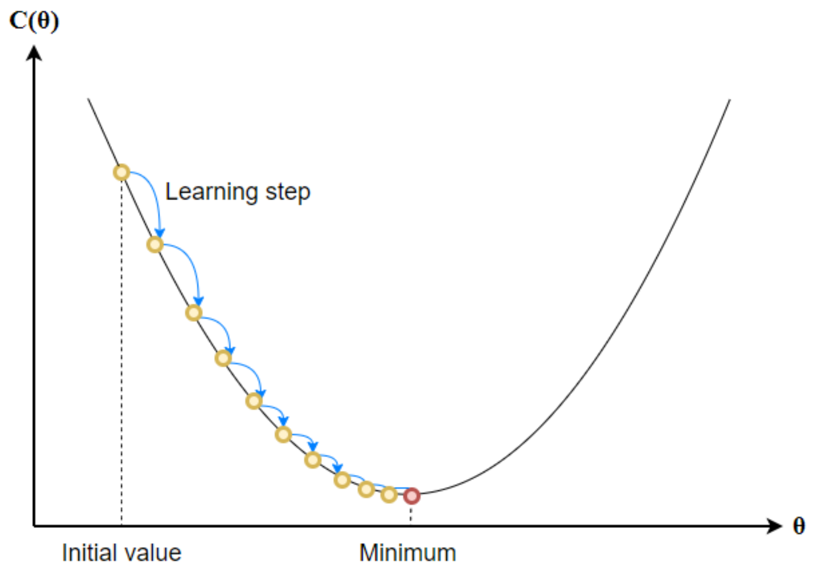

Figure 3-2.: Illustration of the gradient descent algorithm used to minimize a loss function $C(\theta)$ with only one parameter θ.

The amount of data used to compute the gradient of the objective function may vary in order to find a balance between the accuracy and the time required for the parameter update (Ruder, 2016). Batch gradient descent, stochastic gradient descent and mini-batch gradient descent are variants of the gradient descent algorithm which differ by the amount of data used. Although in practice Mini-batch gradient descent is the most popular, the application of these three variants depends on the context and main features of the optimization problem (Ruder, 2016).

Batch gradient descent

Batch gradient descent is one of the variants of the gradient descent algorithm in which the gradient of the loss function $C(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}$ is computed for the whole data set, that is, the parameters are updated using the average of the gradients of all observations as follows (Bottou, 2010):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \tfrac{1}{n} \textstyle\sum_{i=1}^{n} \frac{\partial C(z_i, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}},$$

where z_i corresponds to the i-th observation of the data set. In this case, for just one parameter update it is required to calculate the gradients for the entire data set.

Stochastic gradient descent

As mentioned above, when applying batch gradient descent, to move a single step towards the minimum of the loss function $C(\boldsymbol{\theta})$ it is required to compute the gradients for all the observations in the data set and when working with large data sets this method is not very efficient. To deal with this problem stochastic gradient descent (SGD) was created. SGD is a variant of the gradient descent method in which the gradient of the cost function $C(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}$ is computed for each observation in the data set, that is, the parameters are updated based on the gradient of a single randomly picked observation z_t as follows (Bottou, 2010):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \frac{\partial C(z_t, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}.$$

Contrary to the batch gradient descent, in this case, one observation at a time is considered to take a single step towards the minimum of the loss function. Is guaranteed that stochastic gradient descent as well as batch gradient descent converge to the global minimum for convex surfaces and to a local minimum for non-convex surfaces.

Mini-batch gradient descent

Both, batch gradient descent and stochastic gradient descent have advantages and disadvantages in their application. Batch gradient descent converges to the global minimum, however it can be very slow when working with large data sets. On the other hand, stochastic gradient descent converges faster for large data sets but the loss function fluctuates significantly because the parameters are updated frequently. To take advantage of the benefits of both variants and find solution for some of their disadvantages, mini-batch gradient descent was created. In this case, not the entire data set is used, nor a single observation at the same time. The parameters are updated based on the average of the gradients of a batch with a fixed number of observations less than the entire data set called mini-batch (Bengio, 2012):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \frac{1}{B} \sum_{i=Bt+1}^{B(t+1)} \frac{\partial C(z_i, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}},$$

where B is the size of the mini-batch. In this way, the algorithm performs an update for every mini-batch of B observations (Ruder, 2016).

3.3.2. Optimization algorithms in TensorFlow

There are a wide variety of gradient descent optimizers included in the TensorFlow library. In this section, some of these optimizers are presented.

Momentum optimizer

The Momentum method is an extension to SGD created to increase the rate of convergence by including a momentum term (Rumelhart, Hinton, & Williams, 1986). The update rule in this case is as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \tfrac{\partial C(\boldsymbol{\theta}_t)}{\partial \boldsymbol{\theta}_t} + \gamma \triangle \boldsymbol{\theta}_t \ ,$$

where $\triangle\boldsymbol{\theta}_t$ corresponds to the changed applied to the parameters in the previous iteration and γ is the momentum parameter. With this method, the modification of the parameter vector at the current iteration depends on the current gradient as well as on the parameter change of the previous iteration (Qian, 1999).

AdaGrad optimizer

The adaptative gradient algorithm known as AdaGrad is an algorithm for gradient-based optimization that uses a different learning rate for every parameter θ_i at every time step t (Duchi, Hazan, & Singer, 2011). The learning rate is modified based on the past gradients that have been computed for each parameter (Ruder, 2016). The update rule for AdaGrad is as follows (Zeiler, 2012):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \tfrac{\alpha}{\sqrt{\sum_{\tau=1}^{t} g_\tau^2 + \epsilon}} g_t,$$

where g_t^2 is the square of the gradient of the parameters at the t-iteration, g_t contains the gradients of the objective function with respect to the parameters at step t and ϵ is an small positive number added to avoid division by zero. An initial value α for the learning rate must be defined before starting the optimization process. The main advantage of this algorithm is that is no longer required to tune manually the learning rate before starting the optimization process. However, because the sum of squares of all previous gradients of the parameters keeps growing after every iteration, the learning rate decreases until it becomes infinitesimally small, stopping the progress of the algorithm (Ruder, 2016).

AdaDelta optimizer

In order to overcome the main disadvantage of AdaGrad algorithm regarding the continual decay of the learning rates, AdaDelta was created (Zeiler, 2012). Instead of accumulating all squared gradients until step t, this algorithm restricts the number of accumulated past gradients to some fixed number w and accumulates the previous squared gradients as an exponentially decaying average of the squared gradients (Zeiler, 2012). The update rule for AdaDelta is as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\sqrt{E[\triangle\theta^2]_t + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} g_t,$$

where $E[g^2]_t$ and $E[\triangle\theta^2]_t$ are defined as:

$$\begin{aligned} E[g^2]_t &= \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \\ E[\triangle\theta^2]_t &= \gamma E[\triangle\theta^2]_{t-1} + (1-\gamma) \triangle \theta_t^2, \end{aligned}$$

where γ is a decay constant. It is required to initialize $E[g^2]_t$ and $E[\triangle\theta^2]_t$ before starting the optimization process. The main advantages of this method is that it ensures that even after many iterations the learning process continues. Although, with the original version of this method is not required to set an initial learning rate, the implementation of this method in TensorFlow allows to set this initial value.

RMSprop optimizer

RMSProp is an adaptive learning rate method proposed by Geoff Hinton that was developed around the same time that the AdaDelta optimizer to overcome the disadvantages of the AdaGrad algorithm (Ruder, 2016). The update rule for RMSprop is as follows:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} g_t,$$

where $E[g^2]_t$ is defined as:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2,$$

where γ is a decay constant that Hilton suggests to be set to 0.9. As shown above, the RMSprop is very similar to the AdaDelta algorithm.

Adam optimizer

The Adaptive Moment Estimation method (Adam) is an adaptive leaning rate method that combines the advantages of the AdaGrad and the RMSprop methods (Kingma & Ba, 2014). Adam, as well as RMSprop, updates the exponentially decaying average of the squared gradients v_t and also the exponentially decaying average of the gradients m_t as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \end{aligned}$$

where $\beta_1, \beta_2 \in [0,1)$. Also, m_t and v_t are estimates of the 1st moment and the 2nd raw moment of the gradient respectively. As m_t and v_t vectors are always initialize as 0's, they are biased towards zero. Because of this, instead of using these estimates, the bias-corrected estimates are computed and used to update the parameters as follows:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t. \end{aligned}$$

AdaDelta, RMSprop and Adam are very similar algorithms that work well in similar scenarios, however, the bias-correction helps Adam slightly outperform RMSprop (Ruder, 2016).

3.3.3. Hyperparameters of the TensorFlow optimizers

As seen in the previous section, the TensorFlow optimizers differ by the update rule, that is, by the strategy used to change the value of the parameters from one iteration to another. Each update rule involves what are known as hyperparameters. The hyperparameters are values that are manually specified and that help estimate parameters. In Table **3-1**, the hyperparameters of each TensorFlow optimizer are presented as well as their default values. These default values are set in the **tensorflow** package. From Table **3-1**, we observe that all optimizers have hyperparameter α which corresponds to the learning rate. Also, there are some hyperparameters that do not have default values and therefore, the user must provide these values when using the respective optimizer.

It is important to mention that it is not possible to know the best value for an hyperparameter on a given problem. We can search this value by trial and error or by designing a tuning process in which we use multiple values for the hyperparameter in the optimization process and determine with which of these values we obtain the best results.

Optimizer	Hyperparameters	Hyperparameter names	Default values
GradientDescent	α	learning_rate	-
Momentum	α	learning_rate	-
	γ	momentum	-
Adagrad	α	learning_rate	-
	ϵ	epsilon	1×10^{-7}
Adadelta	α	learning_rate	0.001
	γ	rho	0.95
	ϵ	epsilon	1×10^{-8}
RMSProp	α	learning_rate	-
	γ	decay	0.9
	ϵ	epsilon	1×10^{-10}
Adam	α	learning_rate	0.001
	β_1	beta1	0.9
	β_2	beta2	0.999
	ϵ	epsilon	1×10^{-8}

Table 3-1.: Default values and names of hyperparameters of TensorFlow optimizers.

For more information about the hyperparameters of each TensorFlow optimizer go to the **TensorFlow website**.

3.3.4. Gradient descent method versus the Newton-Raphson method

The gradient descent method and the Newton-Raphson method are some of the most important approaches for solving optimization problems. As shown previously, the Newton-Raphson is a second-order gradient method, which means that it uses the second derivatives to update the parameters in each iteration as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \left[\boldsymbol{H}(\hat{\boldsymbol{\theta}}_k)\right]^{-1} S(\boldsymbol{\theta}_k), \tag{3-1}$$

On the other hand, the gradient descent method is a first-order gradient method that updates the parameters as follows:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha \frac{\partial C(\boldsymbol{\theta}_k)}{\partial \boldsymbol{\theta}_k}, \tag{3-2}$$

The Newton-Raphson method is very efficient especially when close to the minimum and it usually converges after just a few iterations. However, its main disadvantage is the cost of forming and storing the Hessian and the cost of computing the Newton step (3-1) (Boyd & Vandenberghe, 2004). Also, in some cases when the starting point is to far away from the true value, the iteration process may fail (Yang, 2021). Due to the greater complexity of this method, it is not frequently used in machine learning problems. This is why, the majority of the TensorFlow optimizers are based on the gradient descent method.

In the case of the gradient descent method, its main advantage is its simplicity as it only requires first-order information to update the parameters values. On the other hand, despite being an easily applicable method its convergence can be very slow. Additionally, is a method that depends heavily on the choice of the step size or as it is commonly known in machine learning, the learning rate. This is why the efforts of a lot of authors have been concentrated in the creation of optimization algorithms based on this method such as the Adam optimizer, the Adagrad optimizer, the RMSProp optimizer, among others, which use adaptive learning rates.

Due to the differences in the way each of these approaches addresses the optimization problem, their implementation depends on the problem to solve. However, some authors recommend to combine both methods by using the gradient descent method at the initial stage of the optimization process and for the last iterations use the Newton-Raphson method (Nesterov, 2014).

3.4. TensorFlow for R

As mention previously, there are a wide variety of APIs available in several languages that allow users to interact with the TensorFlow library. In the particular case of the R programming language, the R interface to TensorFlow consists of a collection of R packages that provide a variety of interfaces to TensorFlow for different tasks and levels of abstraction. One of these packages is **tensorflow** (Allaire & Tang, 2021), a low-level interface to the TensorFlow computational graphs. With this package it is possible to create TensorFlow graphs, initiate TensorFlow sessions, define variables and placeholders and implement the TensorFlow optimizers to minimize a loss function. For more details of the **tensorflow** package go to the **TensorFlow website**.

Another available package in R is **tfprobability** (Keydana, 2020), an interface to the Python library TensorFlow Probability for statistical computation and probabilistic modeling. These package includes multiple statistical distributions and methods such as the mean, the mode, the standard deviation, the variance, among others. For more details of the **tfprobability** package go to the **tfprobability website**. Some functions included in both of these packages are used in the **estimtf** package.

4. estimtf package

The **estimtf** package (Garcés & Hernández, 2021) allows the implementation of the maximum likelihood method to estimate parameters of multiple probability distributions and linear regression models using TensorFlow.

The main functions in this package are `mle_tf` which allows the user to estimate parameters of some probability distributions and `mlereg_tf` to estimate parameters of some linear regression models. The **estimtf** package also includes the `summary` function to compute and return some summary statistics related with the estimates, the `print` function to display the parameters estimates and the `plot_loss` function to display a graph that contains the loss value, that is, the negative log-likelihood value in each iteration of the estimation process (see Appendix A for more details).

4.1. Estimation process with the estimtf package

In general terms, the estimation process designed and implemented in the **estimtf** package consist of the following steps:

1. Identify the parameters to be estimated depending on the provided distribution and the list of fixed parameters. For each parameter to estimate, a TensorFlow variable is created with its respective initial value using the `tf$Variable()` function.
2. Depending on the user selection of the TensorFlow optimizer and the provided hyperparameters such as the learning rate, define the optimizer using the `tf$compat$v1$train()` function.
3. Define the loss function which corresponds to the negative log-likelihood function for the provided distribution.
4. Start the iterative process to find the parameters values that minimize the loss function. The parameters values change from one iteration to another starting from the provided initial values until convergence or until the maximum number of iterations is reached.
5. Compute the Hessian matrix and the standard error for each estimated parameter.

6. Compute the Z-score and the p-value of the significance test for each estimated parameter.

4.2. Available distributions

When working with the **estimtf** package, it is possible to estimate parameters for eight well-known distributions presented in Table **4-1**, for which the user must only provide the name of the distribution.

Distribution	pdf / pmf	Domain	Parameters
Normal	$f(x\|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$	$-\infty \leq x < \infty$	$-\infty < \mu < \infty,\ \sigma > 0$
Poisson	$p\,(x\|\lambda) = \frac{e^{-\lambda}\lambda^x}{x!}$	$x = 0, 1, \ldots$	$0 \leq \lambda < \infty$
Weibull	$f(x\|\gamma,\beta) = \frac{\gamma}{\beta} x^{\gamma-1} e^{-\frac{x^\gamma}{\beta}}$	$0 \leq x < \infty$	$\gamma > 0,\ \beta > 0$
Exponential	$f(x\|\beta) = \frac{1}{\beta} e^{-\frac{x}{\beta}}$	$0 \leq x < \infty$	$\beta > 0$
Lognormal	$f(x\|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} \frac{\exp\left(-(\log x-\mu)^2/(2\sigma^2)\right)}{x}$	$0 \leq x < \infty$	$-\infty < \mu < \infty,\ \sigma > 0$
Beta	$f(x\|\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}$	$0 \leq x \leq 1$	$\alpha > 0,\ \beta > 0$
Gamma	$f(x\|\alpha,\beta) = \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1} e^{-\frac{x}{\beta}}$	$0 \leq x < \infty$	$\alpha > 0,\ \beta > 0$
Binomial	$p\,(x\|n,p) = \binom{n}{x} p^x (1-p)^{n-x}$	$x = 0, 1, \ldots, n$	$0 \leq p \leq 1$

Table 4-1.: Available distributions in the **estimtf** package for which the user must only provide the name of the distribution.

In addition to the eight distributions included in Table **4-1**, the **estimtf** package allows users to estimate parameters for other discrete or continuous distributions that are not necessarily implemented in R by providing its probability mass/density functions. These functions must be defined as R functions whose arguments are the distributional parameters. As new distributions are being created constantly, this is a very important and useful feature of the **estimtf** package because it facilitates the process of parameter estimation carried out by the authors and those interested in these distributions. In further sections, we show how to use the **estimtf** package for estimating parameters of distributions different from those presented above.

4.3. Installing the estimtf package in R

To be able to use the **estimtf** package we recommend to follow the next steps:

1. Download Anaconda (Optional). TensorFlow depends on Python to work so it is recommended to install Anaconda, the most popular Python distribution platform. For download visit the **Anaconda website**.

2. Install the **reticulate** package. This R package provides an R interface to Python modules, classes and functions.

3. Install the **tensorflow**package. After installation load the package.

4. Use the `install_tensorflow()` function to install TensorFlow python module.

5. Load the **tensorflow** package and confirm that the TensorFlow installation succeeded making sure there are no errors when using a TensorFlow function.

6. Install the **devtools** package. This R package is very useful for package development.

7. Install the **estimtf** package. This R package can be installed from GitHub. After installation load the package and use it.

Steps 2 to 7 can be carried out by running the following code in R:

```r
# Step 2
install.packages("reticulate")
# Step 3
install.packages("tensorflow")
library(tensorflow)
# Step 4
install_tensorflow()
# Step 5
library(tensorflow)
tf$constant("Hello Tensorflow")
# Step 6
install.packages("devtools")
# Step 7
devtools::install_github("SaraGarcesCespedes/estimtf", force=TRUE)
library(estimtf)
```

Code 4.1: Code of recommended steps for installing the **estimtf** package in R.

In the case of having problems with the installation of the **estimtf** package, we recommend to use Google Colab, an interactive notebook provided by Google that allows to write and execute R and Python code. To use the notebook with R, use this URL: **https://colab.research.google.com/create=truelanguage=r**. To use the **estimtf** package in Google Colab, only step 7 is required.

4.4. Maximum likelihood estimation of distributional parameters

As mentioned above, with the `mle_tf` function it is possible to estimate distributional parameters given a random sample from a distribution. The following is the structure of the `mle_tf` function:

```
mle_tf(x,
       xdist = "Normal",
       fixparam = NULL,
       initparam,
       bounds = NULL,
       optimizer = "AdamOptimizer",
       hyperparameters = NULL,
       maxiter = 10000,
       tolerance = .Machine$double.eps)
```

Code 4.2: Structure of the `mle_tf` function.

To estimate parameters with this function the user must provide the following arguments:

- `x`: a vector with data.
- `xdist`: the name of the distribution of interest (Table 4-1) or the name of the R function that contains the probability density/mass function of the distribution.
- `fixparam`: a list with the names of the fixed parameters and their respective values.
- `initparam`: a list with the initial values of the parameters to be estimated.
- `bounds`: a list with lower and upper bounds for each parameter to be estimated. The list must contain the parameters names and vectors with the bounds.
- `optimizer`: the name of the TensorFlow optimizer to use in the estimation process.

- `hyperparameters:` a list with the names and values of the hyperparameters of the selected optimizer. The hyperparameters names and their default values of each optimizer are presented in Table **3-1**.

- `maxiter:` the maximum number of iterations for the estimation process.

- `tolerance:` a small positive number. The estimation process stops when the difference between the loss value or the parameters values from one iteration to another is lower than this value.

Some of these arguments have default values as can be seen in Code 4.2. If information for the arguments `x` and `initparam` is not provided, the estimation process will not be performed. The `mle_tf` function returns a list containing the parameters estimates, their standard deviations, the estimated variance-covariance matrix and the number of iterations to convergence. For more details about the `mle_tf` function see Appendix A.

It is important to mention that if the user wants to estimate parameters from a distribution not included in Table **4-1** using the `mle_tf` function, it is required to provide the name of an R object of class function that contains the probability mass/density function of the distribution of interest. This function must have as arguments x and the parameters of the distribution. Also, the probability mass/density function must not contain curly brackets. The only curly brackets that the function can contain are those that enclose the function, that is, those that define the beginning and end of the R function. Within the function, the user should not add curly brackets as this can generate problems in the computation of the log-likelihood function. The following code shows how to define the probability mass/density function to estimate the parameters of the extended exponential geometric distribution (Adamidis, Dimitrakopoulou, & Loukas, 2005) with a probability density function given by:

$$f(x;\gamma,\beta) = \frac{\beta\gamma e^{-\beta x}}{(1-(1-\gamma)e^{-\beta x})^2}.$$

```r
# Define the pdf of the EEG distribution
deeg <- function(x, beta, gamma) {
(beta * gamma * exp(-beta * x))/(1 - (1 - gamma) * exp(-beta * x))^2
}
```

Code 4.3: How to define the probability density function of the EEG distribution in R when using the `mle_tf` function.

The `mle_tf` function in practice

To illustrate the use of the `mle_tf` function we simulate data from a normal distribution with $\mu = 10$ and $\sigma = 3$ and try to estimate these parameters using the `mle_tf` function. The following, is the code written in R to estimate the parameters of a normal distribution using the `mle_tf` function of the **estimtf** package. First, we have to load the package and create a vector with the data to be fitted. Then we have to provide values for the arguments of the `mle_tf` function. It is required to provide the initial values of the parameters.

```r
# load the estimtf package
library(estimtf)

# simulate data from normal distribution
x <- rnorm(n = 1000, mean = 10, sd = 3)

# use the mle_tf function
estimation_1 <- mle_tf(x = x,
                       xdist = "Normal",
                       initparam = list(mean = 1.0, sd = 1.0),
                       bounds = NULL,
                       optimizer = "AdamOptimizer",
                       hyperparameters = list(learning_rate = 0.1))
```

Code 4.4: Process to estimate parameters of a normal distribution using the `mle_tf` function.

In Code 4.5, the R output obtained by estimating the parameters μ and σ from the normal distribution using the `mle_tf` function is presented. This output provides the parameter estimates, their standard error and the Z-score and p-value for the significant test of each parameter. We observe that the MLE $\hat{\mu} = 10.009$ and $\hat{\sigma} = 2.954$ obtained using the `mle_tf` function, are very close to the true value of the parameters $\mu = 10$ and $\sigma = 3$ respectively.

```
# print the results
summary(estimation_1)


## Distribution: Normal
## Number of observations: 1000
## TensorFlow optimizer: AdamOptimizer
## ------------------------------------------------------
##      Estimate  Std. Error Z value Pr(>|z|)
## mean  10.00952    0.09343  107.13   <2e-16 ***
## sd     2.95464    0.06659   44.37   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Code 4.5: Summary of the estimates of parameters μ and σ using the `mle_tf` function.

4.5. Maximum likelihood estimation of regression parameters

To estimate parameters of linear regression models, the `mlereg_tf` function should be used. The following is the structure of this function:

```
mlereg_tf(ydist = y ~ Normal,
          formulas,
          data,
          available_distribution = TRUE,
          fixparam = NULL,
          initparam = NULL,
          link_function = NULL,
          optimizer = "AdamOptimizer",
          hyperparameters = NULL,
          maxiter = 10000,
          tolerance = .Machine$double.eps)
```

Code 4.6: Structure of the `mlereg_tf` function.

To estimate parameters with this function the user must provide the following arguments:

- `ydist`: a formula object that specifies the response variable name and distribution. The distribution is either the name of the distribution of interest (Table 4-1) or the name of the R function that contains the probability density/mass function of the distribution.

- `formulas:` a list with objects of type formula, one for each parameter to be estimated, that specifies its linear predictor.
- `data:` a data frame containing the response variable and the covariates.
- `available_distribution:` if `TRUE` the distribution of the response variable is one of the available distributions in the package (Table 4-1).
- `fixparam:` a list with the names of the fixed parameters and their respective values.
- `initparam:` a list with the initial values of the parameters to be estimated.
- `link_function:` a list with names of the parameters to be linked and the corresponding name of the link function. The available link functions are: `log`, `logit`, `squared` and `identity`.
- `optimizer:` the name of the TensorFlow optimizer to use in the estimation process.
- `hyperparameters:` a list with the names and values of the hyperparameters of the selected optimizer. The hyperparameters names and their default values of each optimizer are presentes in Table **3-1**.
- `maxiter:` the maximum number of iterations for the estimation process.
- `tolerance:` a small positive number. The estimation process stops when the difference between the loss value or the parameters values from one iteration to another is lower than this value.

Some of these arguments have default values as can be seen in Code 4.6. If information for the arguments `formulas` and `data` is not provided, the estimation process will not be performed. The `mlereg_tf` function returns a list containing the parameters estimates, their standard deviations, the estimated variance-covariance matrix and the number of iterations to convergence. For more details about the `mlereg_tf` function, see Appendix A.

It is important to mention that if the user wants to estimate parameters from a distribution not included in table **4-1** using the `mlereg_tf` function, it is required to provide the name of an R object of class function that contains the probability mass/density function of the distribution of interest. This function must have as arguments the response variable and the parameters of the response variable distribution. Also, the probability mass/density function must not contain curly brackets. The only curly brackets that the function can contain are those that enclose the function, that is, those that define the beginning and end of the R function. Within the function, the user should not add curly brackets as this can generate problems in the computation of the log-likelihood function. The following code shows how to define the probability mass/density function when the distribution of the response variable Y is the extended exponential geometric distribution (Adamidis et al., 2005):

```r
# Define the pdf of the response variable 'Y' distribution
deeg <- function(y, beta, gamma) {
(beta * gamma * exp(-beta * y))/(1 - (1 - gamma) * exp(-beta * y))^2
}
```

Code 4.7: How to define the probability density function of the EEG distribution in R when using the `mlereg_tf` function.

Notice that when using the `mlereg_tf` function, the names of the R function arguments must be the name of the response variable and the name of the parameters of the response variable distribution. On the contrary, when using the `mle_tf` function the name of the arguments must be x and the name of the parameters of the distribution of interes.

The `mlereg_tf` function in practice

To illustrate the use of the `mlereg_tf` function we simulate a random sample with $Y \sim N(\mu, \sigma^2)$ where $\mu = \beta_0 - \beta_1 x$, $\sigma = 3$, $x \sim U(-3, 3)$ and try to estimate $\beta_0 = 5$ and $\beta_1 = -2$ using the `mlereg_tf` function. The following, is the code written in the R to estimate the parameters of a simple linear regression model using the `mlereg_tf` function of the **estimtf** package. First, we have to load the package and create a data frame with the data for the response variable and the covariates. Then we have to provide values for the arguments of the `mlereg_tf` function. With this function we have to specify the distribution of the response variable using a formula object and as in this case the distribution of interest is available in the **estimtf** package, the argument `available_distribution` must be equal to `TRUE`. As the σ parameter is constant, we must provide its value through the `fixparam` argument. Finally, when using the `mlereg_tf` function, if we do not provide the list with initial values for the parameters, default values of zero are used in the estimation process.

```r
# load the estimtf package
library(estimtf)

# simulate data from a simple linear regression model
x <- runif(n = 1000, min = -3, max = 3)
y <- rnorm(n = 1000, mean = 5 - 2 * x, sd = 3)
data <- data.frame(y = y, x = x)
```

```
# use the mlereg_tf function
estimation_2 <- mlereg_tf(ydist = y ~ Normal,
                          formulas = list(mean = ~ x),
                          data = data,
                          available_distribution = TRUE,
                          fixparam = list(sd = 3),
                          initparam = list(mean=list(Intercept=1.0,x=0)),
                          link_function = NULL,
                          optimizer = "AdamOptimizer",
                          hyperparameters = list(learning_rate = 0.1))
```

Code 4.8: Process to estimate parameters of a simple linear regression model using the `mlereg_tf` function.

In Code 4.9 the R output obtained by estimating the parameter μ from the normal distribution using the `mlereg_tf` function is presented. As the parameter μ is a function of covariate x, this output provides the estimates of coefficients β_0 and β_1, their standard error and the Z-score and p-value for the significant test of each coefficient. We observe that the MLE $\hat{\beta}_0 = 4.810$ and $\hat{\beta}_1 = -2.014$ obtained using the `mlereg_tf` function, are very close to the true value of the coefficients $\beta_0 = 5$ and $\beta_1 = -2$ respectively.

```
# print the results
summary(estimation_2)


## Distribution: Normal
## Number of observations: 1000
## TensorFlow optimizer: AdamOptimizer
## ---------------------------------------------------------------
## Distributional parameter: mean
## ---------------------------------------------------------------
##             Estimate. Std..Error t.value Pr...t..
## (Intercept)   4.81055    0.09492   50.68   <2e-16 ***
## x            -2.01495    0.05412  -37.23   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## ---------------------------------------------------------------
```

Code 4.9: Summary of the estimates of parameter μ using the `mlereg_tf` function.

To see the code of the `mle_tf` and the `mlereg_tf` functions and more information about the **estimtf** package, visit this **GitHub repository**. Also, to learn more about how to use the package, visit this **Colab notebook** in which we included some examples with their respective R code.

Finally, it is important to mention that when using the `mle_tf` function or the `mlereg_tf` function, the estimation process may fail because when evaluating the negative log-likelihood function during an iteration R returns `NaN` which stands for *Not A Number* and suggests that an invalid computation was conducted. This can be caused for multiple reasons including problems with the input data, a high learning rate or a poor choice of the initial values of the parameters. In this case, the user should follow some of these recommendations and start the process again:

- Reduce the learning rate.
- Check the input data as it is possible that some of the values are neither integer nor float.
- Change the initial values provided for the parameters.
- Try different optimizers.

It is also important that when estimating parameters from regression models, the user determines first if it is necessary to apply a link function to the parameters that are linear functions of the explanatory variables.

4.6. Starting values for the parameters

A lot of optimization methods including all the TensorFlow optimizers are iterative and therefore, require the user to specify some initial point from which to begin the iterations. One of the main challenges facing these methods is how the choice of this initial point affects their performance. In some cases, the initial point can cause the algorithm to fail. On the other hand, when the algorithm does converge it influence the speed of convergence and its ability to find the global minimum (Goodfellow, Bengio, & Courville, 2016).

Despite being extremely important for numerical optimization, the problem of selecting the initial point is complex and we do not intend to address it in this work. However, as the functions of the **estimtf** package require to set an initial point, in this section we mention some of the methods frequently used for this selection:

- Begin with the estimates obtained by other estimation methods, like the method of moments (Karlis & Xekalaki, 2003).

- Start from different initial values and stop after a small number of iterations. Keep the initial value that led to the largest value of the log-likelihood after those initial iterations (Karlis & Xekalaki, 2003).
- If there are bounds on the parameters, use the midpoints of the intervals as initial values. For positive parameters, use the square root of the upper bound. Also, these initial values should not be on a bound (Nash, 2014).
- Use "good starting values" suggested in the literature or based on expert knowledge (Nash, 2014).
- If it is known that the problem can be solved starting from any value, use 1 as the initial value for all the parameters (Nash, 2014).

These are some of the methods recommended to set the initial values of parameters in optimization problems and can be used to initiate the estimation process implemented in the `mle_tf` function and the `mlereg_tf` function.

4.7. Details of the estimation process

As mentioned before, the functions of the **estimtf** package implement an iterative optimization process designed to use a TensorFlow optimizer to minimize the negative log-likelihood function. Before starting this iterative process, it is required to define all tensors that represent the random variables and the parameters to estimate, as well as the optimizer responsible for the minimization of the objective function and the values for its hyperparameters. Finally, it is important to set a value for the tolerance which is used as a criterion to end the iterative process.

In Algorithm 1, we present the iterative optimization process implemented in the functions `mle_tf` and `mlereg_tf` of the **estimtf** package. This process starts by initializing the parameters using the initial values provided by the user and initializing the number of iterations t. Then, we enter a while loop in which the gradients of the objective function $C(\boldsymbol{\theta})$ are computed. These gradients are the partial derivatives of the objective function with respect to each of the parameters. With these gradients, we compute $\phi(\cdot)$ which represents the rule used by the selected optimizer to update the parameters in each iteration. This rule varies by optimizer but in general it requires the gradients and the values for the optimizer hyperparameters represented by $\boldsymbol{h}$. After updating the parameters' values, we evaluate them in the objective function. If the objective function value has not changed much with respect to the value in the previous iteration, it means that the values of the parameters did not change significantly and therefore, the computed gradients are very small. This indicates that we are close to the global minimum of the function in the case that the objective function is convex or at least close to a local minimum when this function is not convex. To determine if the change from one iteration to another in the objective function is small enough to stop the optimization process, a tolerance value ϵ is used. If the absolute difference between the objective function value in the actual iteration and in the previous iteration is lower than ϵ, the optimization process stops and the maximum likelihood estimates are set as the current values of the parameters. Otherwise, the process continues and all the steps mentioned above are repeated until the condition mentioned above is met or until the maximum number of iterations is reached.

As a result of this optimization process, the maximum likelihood estimates are provided, as well as the total number of iterations required, the value of the objective function in the last iteration and the values of the parameters, the gradients and the objective function obtained in each iteration of the optimization process. All this information is provided to be analyzed by the user if required.

Algorithm 1 Optimization algorithm implemented in the functions `mle_tf` and `mlereg_tf`

```
Require: Objective function $f$; Maximum number of iterations $T$; Tolerance $\epsilon$; Vector with
  values for the hyperparameters of the TensorFlow optimizer $\boldsymbol{h}$;
  $\theta^{(0)} \leftarrow$ initial point in the domain of $f$
  $t \leftarrow 0$
  while $t \leq T$ do
      $g^{(t)} \leftarrow \frac{\partial C(\boldsymbol{\theta}^{(t)})}{\partial \boldsymbol{\theta}^{(t)}}$          ▷ $C(\boldsymbol{\theta})$ represents the negative log-likelihood function
      $\triangle\boldsymbol{\theta}^{(t)} \leftarrow \phi(\boldsymbol{\theta}^{(t)}, g^{(t)}, \boldsymbol{h})$          ▷ $\phi(\cdot)$ depends on the selected optimizer
      $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \triangle\boldsymbol{\theta}^{(t)}$
      if $|C(\boldsymbol{\theta}^{(t+1)}) - C(\boldsymbol{\theta}^{(t)})| < \epsilon$ then
          Stop
      end if
      $t \leftarrow t + 1$
  end while
```

4.8. Bounds on the parameters

The optimization problems can be constrained or unconstrained. When solving the maximum likelihood problem depending on the distribution or regression model, it mat be required to set lower and upper bounds on the parameters that we want to estimate. The conditions imposed on these parameters are in the form of single bounds as follows (Nash, 2014):

$$lower_i \geq \theta_i \text{ or } \theta_i \geq upper_i,$$

or in the form of interval bounds:

$$lower_i < \theta_i < upper_i.$$

Depending on the type of bounds that the parameters require, a transformation is applied on each parameter to prevent them from taking values outside the imposed limits in any iteration of the estimation process.

On of the main features of the estimtf package is that allows R users to define bounds on the parameters when using the `mle_tf` function or the `mlereg_tf` function to find their maximum likelihood estimates.

4.8.1. Set bounds with the mle_tf function

The `mle_tf` function has an argument called `bounds` through which users must provide a list with lower and upper bounds of each parameter of interest. If the user does not provide this information, it is assumed in the estimation process that no limits on the parameters are required. On the contrary, when the user does provide the parameters bounds, these are classified as single bound or interval bounds. For the parameters that have single bounds, we apply a transformation from the original parameter θ to an internal parameter θ^* as follows (Nash, 2014):

$$\theta^* = log(\theta).$$

On the other hand, for the parameters that have interval bounds, we apply a transformation from the original parameter θ to an internal parameter θ^* as follows (Nash, 2014):

$$\theta^* = \text{arctanh}\left(2 \times \tfrac{(log(\theta)-lower)}{(upper-lower)} - 1\right).$$

Although a transformation is applied to the original parameters for the optimization process, the results are reported in the original scales of the parameters. Also, for the parameters that have bounds, the standard errors, the Z-values and the p-values are reported as NA. Finally, it is important to mention that prior to the optimization process, the initial values provided for the parameters through the argument `initparam` of the `mle_tf` function are checked to ensure that they are not outside the limits defined for each parameter in the argument `bounds`.

4.8.2. Set bounds with the mlereg_tf function

The `mlereg_tf` function has an argument called `link_function` through which users must provide a list with a link function $g(\theta)$ for each parameter of interest. The link functions help us avoid problems of estimation in the limits of parametric space by transforming the linear predictors of the parameters of interest. If the user does not provide this information, no link function is applied to the parameters.

The following link functions are available when using the `mlereg_tf` function:

Parameter range	**Link function**	**Formula link function**
$-\infty$ to ∞	`identity`	θ
0 to ∞	`log`	$\log(\theta)$
	`squared`	θ^2
0 to 1	`logit`	$\log\left(\frac{\theta}{1-\theta}\right)$

Table 4-2.: Link functions available in the `mlereg_function`.

5. Simulation Study

In this section, we performed a simulation study to analyze the effect that the sample size, the learning rate and the optimizer have in the performance of the **estimtf** package while estimating parameters from a distribution or a linear regression model. To do so, we created different scenarios to determine under which conditions good estimates are obtained. The simulations were programmed and performed using the statistical programming language R and the **estimtf** package. We used version 2.0 of TensorFlow and version 4.0.5 of R.

5.1. Methodology

For the simulation study four scenarios were considered in which we generated samples from the extended exponential geometric distribution and the Poisson distribution. In each scenario we changed the parameter vector $\boldsymbol{\theta}$, the sample size n, the learning rate α and the TensorFlow optimizer.

In all scenarios we set the initial values of the parameters at 0.5. We decided to use this value to avoid that the initial point of the estimation process was in the limit of any of the parameters and therefore cause it to fail. Also, we decided to use the same initial value for all the parameters in all scenarios to avoid helping the estimation process in any way.

5.2. The Poisson distribution

The Poisson distribution is a discrete distribution that measures the probability of a given number of occurrences of an event happening in a specified time interval (Kissell & Poserina, 2017). A discrete random variable X has a Poisson distribution with parameter λ if the probability mass function of X is given by (Devore, 2016):

$$P(X = x|\lambda) = \frac{\lambda^x e^{-\lambda}}{x!}, \tag{5-1}$$

where $x = 0, 1, 2, 3, \ldots$; $\lambda > 0$ and $E(Y) = V(Y) = \lambda$.

5.3. The extended exponential geometric distribution

The extended exponential geometric distribution (EEG) is a two parameter distribution proposed by Adamidis et al. (2005). This distribution is often used in modelling lifetime data and according to Louzada et al. (2016), it can also be used in applications in medicine. Let X be a random variable representing a lifetime data. If X has an extended exponential geometric distribution, its probability density function (pdf) is given by:

$$f(x|\gamma, \beta) = \frac{\beta\gamma e^{-\beta x}}{(1-(1-\gamma)e^{-\beta x})^2}, \tag{5-2}$$

where $x > 0$, $\gamma > 0$ and $\beta > 0$.

5.4. Scenarios of the simulation study

The following are the four scenarios considered in the simulation study.

5.4.1. Scenario 1: Poisson distribution without covariates

In this scenario we considered the following model:

$$\begin{aligned} &Y_i \sim P(\lambda), \\ &\lambda = 2.91, \end{aligned} \tag{5-3}$$

with $i = 1, 2, \ldots, n$ where n is the sample size. The parameter was set as $\theta = \lambda = 2.91$ with $\theta_0 = 0.5$ as the initial value of the parameter for the estimation process. The value of λ was chosen trying to replicate the data from a study of female horseshoe crabs on the Gulf of Mexico analyzed by Agresti (2015). The data set contains 173 observations of the number of male crabs attached to the posterior spine of a female crab. The male crabs are called *satellites* and they attach to female crabs as they migrate to shore to reproduce. Also, it contains information about the female crab's color, weight and width (Table **5-1**). The complete data set can be found in the following website: **http://users.stat.ufl.edu/ aa/glm/-data/Crabs.dat**.

crab	y	weight	color
1	8	3.05	2
2	0	1.55	3
3	9	2.3	1
4	0	2.1	3
5	4	2.6	3
6	0	2.1	2
7	0	2.35	1
8	0	1.9	3
9	0	1.95	2
10	0	2.15	3
11	0	2.15	3
12	0	2.65	2
13	11	3.05	2
14	0	1.85	4
15	14	2.3	2

Table 5-1.: Data from a study of female horseshoe crabs on the Gulf of Mexico. color (1, medium light; 2, medium; 3, medium dark; 4, dark), weight (kg).

5.4.2. Scenario 2: Poisson distribution with covariates

In this scenario we considered the following model:

$$\begin{aligned} Y_i &\sim P(\lambda_i), \\ \log(\lambda_i) &= \beta_0 + \beta_1 x_{1i}, \\ X_1 &\sim N(2.5, 0.6), \end{aligned} \tag{5-4}$$

with $i = 1, 2, \ldots, n$ where n is the sample size. The parameter vector was set as $\boldsymbol{\theta} = (\beta_0 = -0.42, \beta_1 = 0.58)$ with $\boldsymbol{\theta_0} = (0.5, 0.5)$ as the vector of initial values of the regression parameters for the estimation process. As in the previous scenario, the values of β_0 and β_1 were chosen trying to replicate the data presented in Table **5-1**.

5.4.3. Scenario 3: EEG distribution without covariates

In this scenario we considered the following model:

$$\begin{aligned} Y_i &\sim EEG(\beta, \gamma), \\ \beta &= 4, \\ \gamma &= 2, \end{aligned} \tag{5-5}$$

with $i = 1, 2, \ldots, n$ where n is the sample size. The parameter vector was set as $\boldsymbol{\theta} = (\beta = 4, \gamma = 2)^\top$ with $\boldsymbol{\theta_0} = (0.5, 0.5)^\top$ as the vector of initial values of the parameters for the estimation process. The values of β and γ were chosen based on a simulation study performed by Louzada et al. (2016).

5.4.4. Scenario 4: EEG distribution with covariates

In this scenario we considered the following model:

$$\begin{aligned} Y_i &\sim EEG(\beta_i, \gamma_i), \\ \log(\beta_i) &= \beta_{10} + \beta_{11} x_{1i}, \\ \log(\gamma_i) &= \beta_{20} + \beta_{21} x_{2i}, \\ X_1 &\sim U(0, 1), \\ X_2 &\sim U(0, 1), \end{aligned} \tag{5-6}$$

with $i = 1, 2, \ldots, n$ where n is the sample size. The parameter vector was set as $\boldsymbol{\theta} = (\beta_{10} = 0.5, \beta_{11} = 1.5, \beta_{20} = 2, \beta_{21} = -3)$ with $\boldsymbol{\theta_0} = (0.5, 0.5, 0.5, 0.5)$ as the vector of initial values of the regression parameters for the estimation process. The values of β_{10}, β_{11}, β_{20} and β_{21} were chosen to emulate a distribution similar to the one presented in the previous scenario.

For all scenarios we considered 3 TensorFlow optimizers: Adam, RMSProp and Adagrad, 3 different values for the learning rate $\alpha = 0.1, 0.01, 0.001$ and 6 values for the sample size $n = 20, 50, 100, 200, 500, 1000$. By combining these values we created 54 different cases in each scenario. For each case, N samples were generated using the models presented above and with each sample we obtained the maximum likelihood estimate $\hat{\boldsymbol{\theta}}$ for the parameter vector $\boldsymbol{\theta}$ using the **estimtf** package.

5.5. Criteria for performance evaluation

To evaluate the performance of the **estimtf** package in estimating the k parameter θ_k of the parameter vector $\boldsymbol{\theta}$ we used the following metrics:

$$Mean\left(\hat{\theta}_k\right) = \frac{1}{N}\sum_{i=1}^{i=N}\hat{\theta}_{ki}, \quad MSE\left(\hat{\theta}_k\right) = \frac{1}{N}\sum_{i=1}^{i=N}(\theta_k - \hat{\theta}_{ki})^2, \quad Bias\left(\hat{\theta}_k\right) = \frac{1}{N}\sum_{i=1}^{i=N}(\theta_k - \hat{\theta}_{ki}),$$

where $N = 1000$ and $\hat{\theta}_{ki}$ corresponds to the estimate of the k parameter θ_k included in the parameter vector $\boldsymbol{\theta}$.

5.6. Simulation process

In each scenario, the simulation process used to evaluate the performance of the **estimtf** package consists of the following steps:

1. Generate n observations from model 5-3, model 5-4, model 5-5 or model 5-6 depending on the scenario. In the case of the EEG distribution, the observations were generated using the inverse transform method (Ross, 2006). For the Poisson distribution, we used the `rpois()` function.
2. Compute the maximum likelihood estimate $\hat{\boldsymbol{\theta}}$ for the parameter vector $\boldsymbol{\theta}$ using the `mle_tf` function or the `mlereg_tf` function (depending on the scenario) of the **estimtf** package.
3. Repeat $N = 1000$ times steps 1 and 2.
4. Compute the Mean, the MSE and the Bias for $\hat{\boldsymbol{\theta}}$.

For each simulation we set the maximum number of iterations of the optimization process, that is, argument `maxiter` of the `mle_tf` function and the `mlereg_tf` function at 10000.

5.7. Results

In this section, we present the results of the four scenarios of the simulation study. For each estimate we analyze the graphs for the Mean, the MSE and the Bias to determine under which conditions we obtain the best results. The dotted lines included in the graphs correspond to the expected values of these metrics.

5.7.1. Results scenario 1

Figure **5-1** shows the mean of the maximum likelihood estimate $\hat{\lambda}$ obtained with different TensorFlow optimizers, sample sizes and learning rates. We observe that with the RMSProp optimizer and with the three different values of the learning rate, the mean of the estimates are close to the real value of the parameter λ, especially when $n > 50$. On the contrary, with the Adagrad optimizer and the Adam optimizer we obtained good estimates only with a learning rate of 0.1. In the case of the the Adagrad optimizer, we observe this behaviour because it is designed in such a way that the learning rate decreases significantly in each iteration and therefore, when starting the optimization process with small learning rates as 0.01 and 0.001, the change in the parameter values from one iteration to another is very small. From this result we can conclude that with this distribution in particular, when using the Adagrad optimizer it is recommended to use an initial learning rate greater than 0.01.

Figure **5-2** shows the MSE of the maximum likelihood estimate $\hat{\lambda}$ obtained with different TensorFlow optimizers, sample sizes and learning rates. From Figure **5-2** we observe that the MSE of $\hat{\lambda}$ decreases to a value very close to 0 as the sample size increases with the Adam optimizer and the RMSProp optimizer regardless of the value of the learning rate. In the case of the Adagrad optimizer, only with a learning rate of 0.1 we obtained a MSE close to 0.

Finally, figure **5-3** shows the bias of the maximum likelihood estimate $\hat{\lambda}$, obtained with different TensorFlow optimizers, sample sizes and learning rates. From Figure **5-3** we observe that with the RMSProp optimizer, the Bias of $\hat{\lambda}$ is close to 0 especially when $n > 50$ with the different values of the learning rate. On the other hand, when using the Adagrad optimizer and the Adam optimizer, only with a learning rate of 0.1 the Bias gets close to 0.

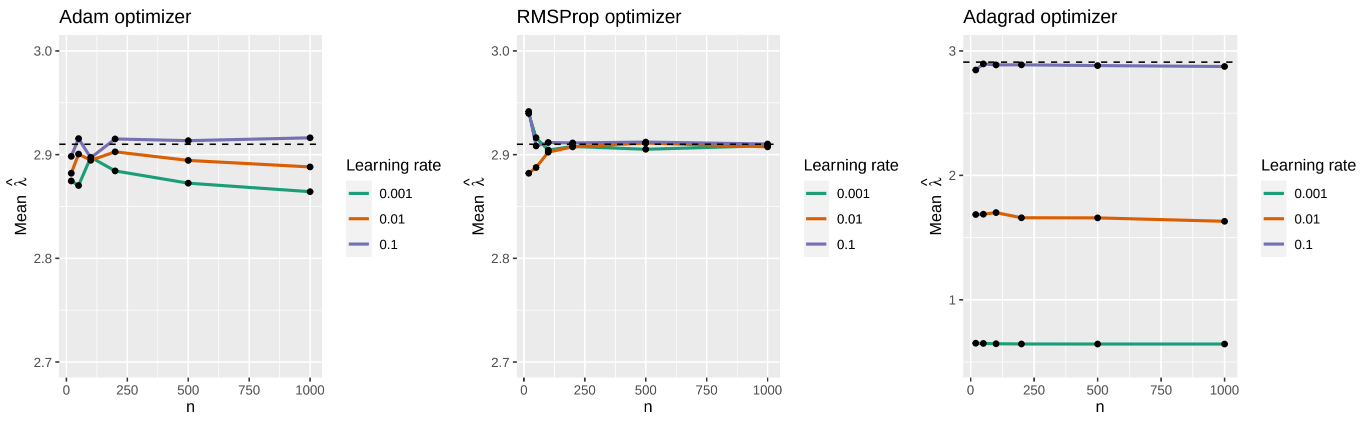

Figure 5-1.: Mean of estimates of $\lambda = 2.91$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

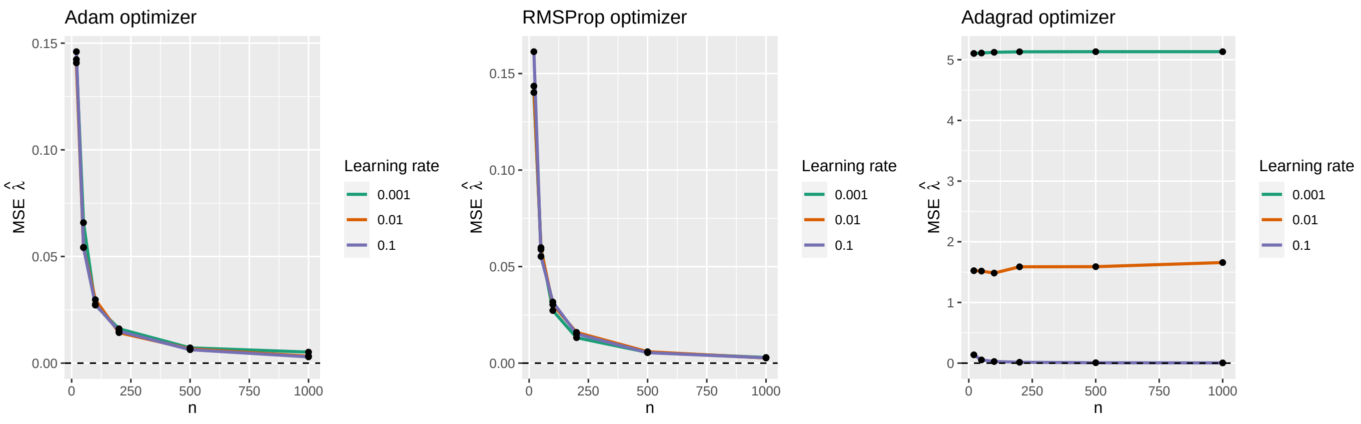

Figure 5-2.: MSE of estimates of $\lambda = 2.91$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

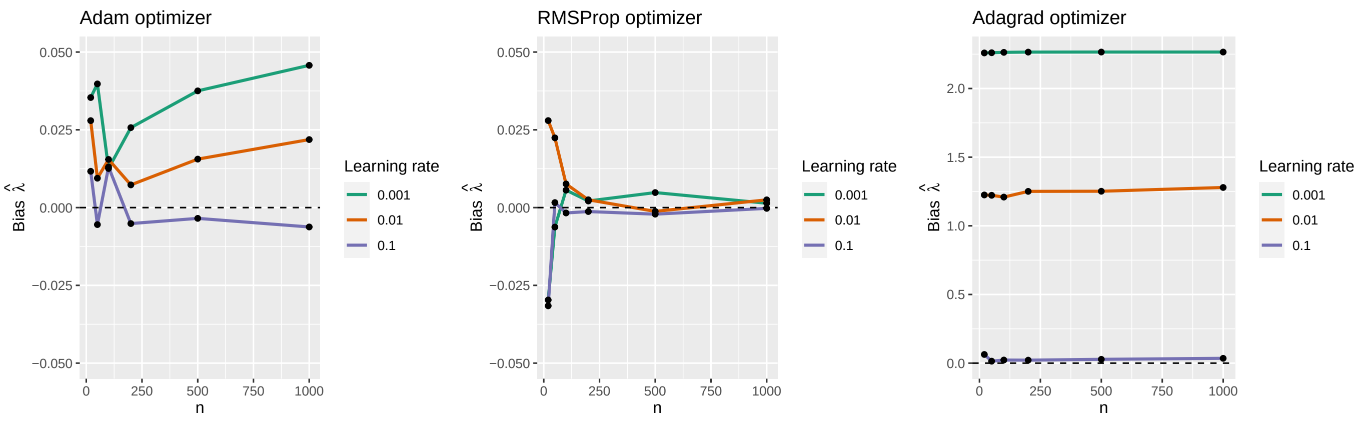

Figure 5-3.: Bias of estimates of $\lambda = 2.91$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

In general, in this scenario we obtained estimates close to the true value of the parameter λ of the Poisson distribution when using the RMSProp optimizer. In the case of the Adagrad optimizer and the Adam optimizer it is very important to use an initial learning rate greater than 0.01 to get more accurate estimates of λ.

5.7.2. Results scenario 2

Figure **5-4** shows the mean of the maximum likelihood estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ obtained with different TensorFlow optimizers, sample sizes and learning rates. We observe that with the Adam optimizer the estimates are close to the real values of the parameters especially when working with a learning rate of 0.1 and when $n > 50$. In the case of the RMSProp optimizer, we obtained estimates close to the real value of the parameters when using learning rates of 0.01 and 0.001. With the Adagrad optimizer, the best performance is achieved when working with a learning rate of 0.1 just like in the previous scenario.

Figure **5-5** shows the MSE of the maximum likelihood estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ obtained with different TensorFlow optimizers, sample sizes and learning rates. From Figure **5-5** we observe that for all optimizers, the MSE of $\hat{\beta}_0$ and $\hat{\beta}_1$ gets close to 0 only when working with a sample size grater than 50. In the particular case of the Adam optimizer, the lowest MSE are obtained when using a learning rate of 0.1. With the RMSProp optimizer errors are closer to 0 when when using learning rates of 0.01 and 0.001. In the case of the Adagrad optimizer, only with a learning rate of 0.1 we obtained a MSE close to 0.

Finally, figure **5-6** shows the bias of the maximum likelihood estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ obtained with different TensorFlow optimizers, sample sizes and learning rates. From Figure **5-6** we observe that with the Adam optimizer the bias is very close to 0 when working with a learning rate of 0.1 and $n > 50$ for both parameters. In the case of the RMSProp optimizer, we obtained a bias closer to 0 as n increased when using learning rates of 0.01 and 0.001 for both parameters. On the other hand, when using the Adagrad optimizer, only with a learning rate of 0.1 the bias is close to 0.

In general, in this scenario we obtained estimates close to the true values of the parameters $\hat{\beta}_0$ and $\hat{\beta}_1$ of the Poisson regression model with all three optimizers but for specific values of the learning rate.

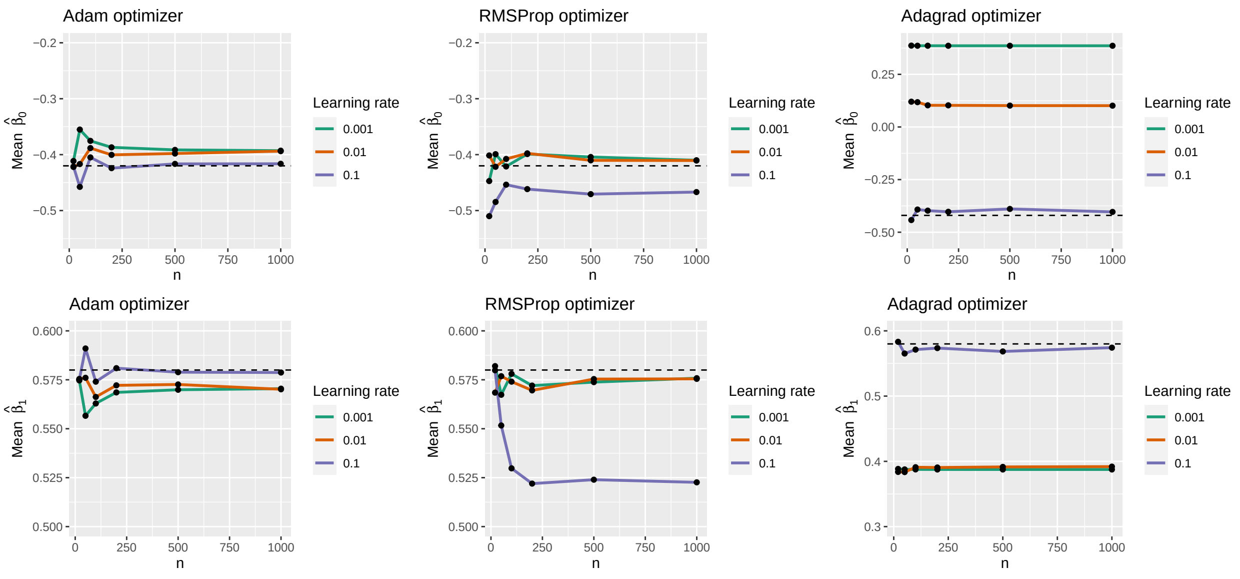

Figure 5-4.: Mean of estimates of $\beta_0 = -0.42$ and $\beta_1 = 0.58$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

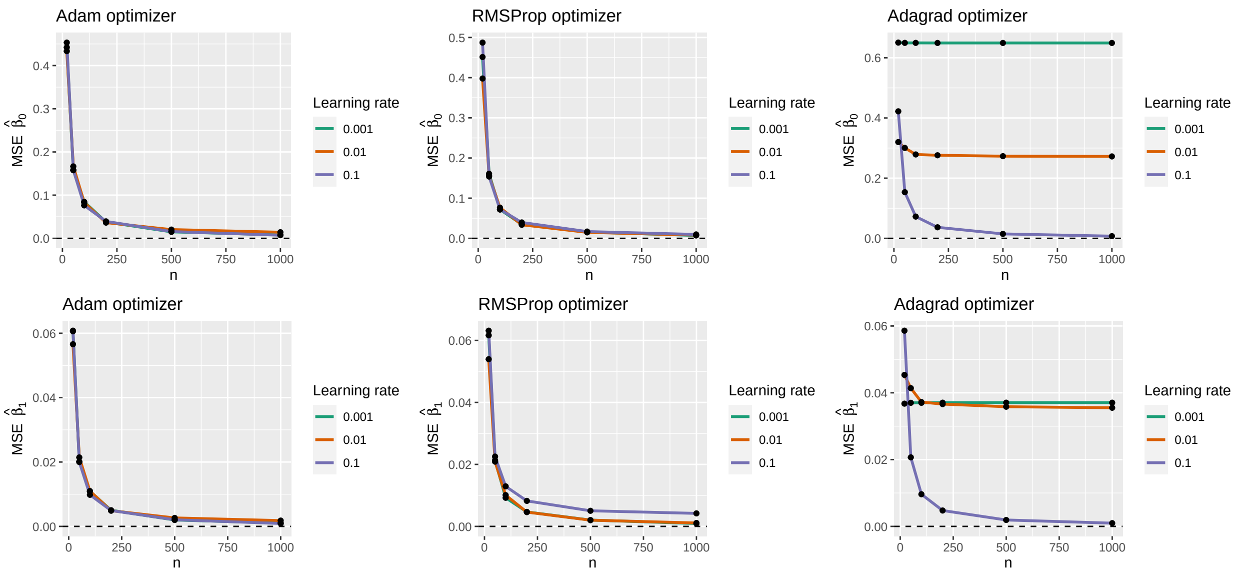

Figure 5-5.: MSE of estimates of $\beta_0 = -0.42$ and $\beta_1 = 0.58$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

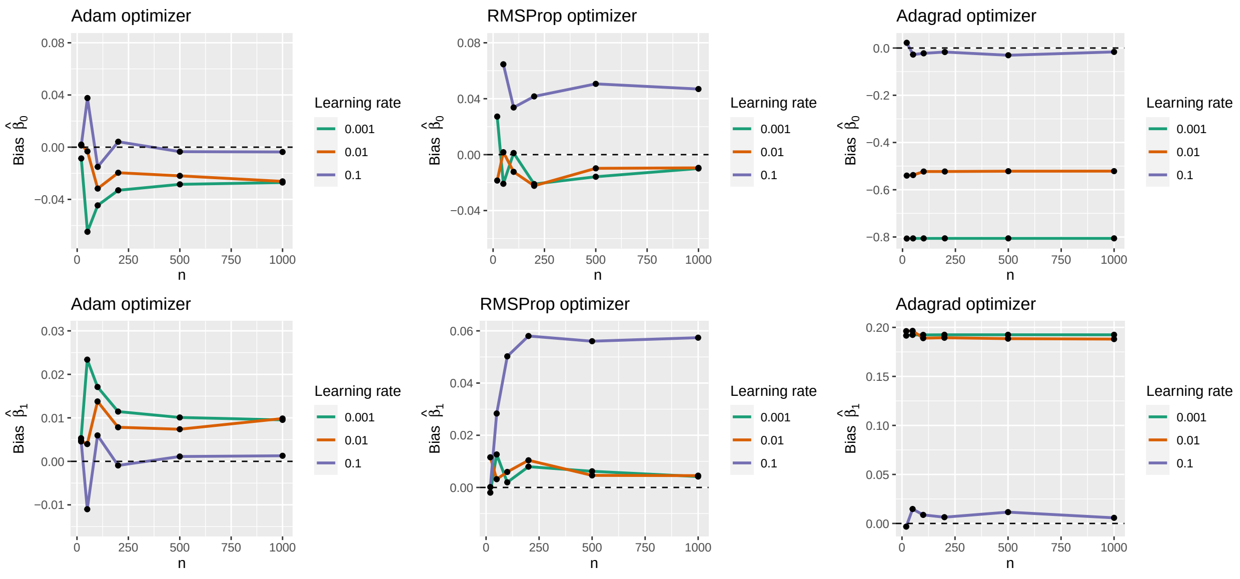

Figure 5-6.: Bias of estimates of $\beta_0 = -0.42$ and $\beta_1 = 0.58$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

5.7.3. Results scenario 3

Figure **5-7** shows the mean of the maximum likelihood estimates $\hat{\beta}$ and $\hat{\gamma}$, obtained with different TensorFlow optimizers, sample sizes and learning rates. We observe that for $\hat{\beta}$ and $\hat{\gamma}$, with the Adam optimizer and the RMSProp optimizer and with the three different values of the learning rate, as the sample size increases, the estimates are closer to the real value of the parameters β and γ. On the contrary, with the Adagrad optimizer we obtained good estimates only with a learning rate of 0.1. From this result we can conclude that with this distribution in particular, when using the Adagrad optimizer it is recommended to use an initial learning rate greater than 0.01.

Figure **5-8** shows the MSE of the maximum likelihood estimates $\hat{\beta}$ and $\hat{\gamma}$, obtained with different TensorFlow optimizers, sample sizes and learning rates. From Figure **5-8** we observe that the MSE of $\hat{\beta}$ and $\hat{\gamma}$ decreases to a value very close to 0 as the sample size increases with the Adam Optimizer and the RMSProp optimizer regardless of the value of the learning rate. However, it is important to notice that with these optimizers, the MSE of the estimates when $n = 20$ is very high especially in the case of the γ parameter. Therefore, we can conclude that when estimating the parameters of the EEG distribution it is recommended to work with $n > 20$ especially when using the Adam or the RMSProp optimizer. In the case of the Adagrad optimizer, only with a learning rate of 0.1 we obtained a MSE close to 0.

Finally, figure **5-9** shows the bias of the maximum likelihood estimates $\hat{\beta}$ and $\hat{\gamma}$, obtained with different TensorFlow optimizers, sample sizes and learning rates. From Figure **5-9** we observe that with the Adam optimizer and the RMSProp optimizer, the Bias of $\hat{\beta}$ and $\hat{\gamma}$ approaches 0 as n increases with the three different values of the learning rate. On the other hand, when using the Adagrad optimizer and a learning rate of 0.1 the Bias gets closer to 0 as the sample size increases, however, with a lower learning rate, the Bias is always greater than 0 regardless of the sample size.

In general, in this scenario we obtained estimates close to the true values of the parameters β and γ of the EEG distribution with the Adam optimizer and the RMSProp optimizer especially with $n > 20$. In the case of the Adagrad optimizer, it is very important to use an initial learning rate greater than 0.01 to get more accurate estimates.

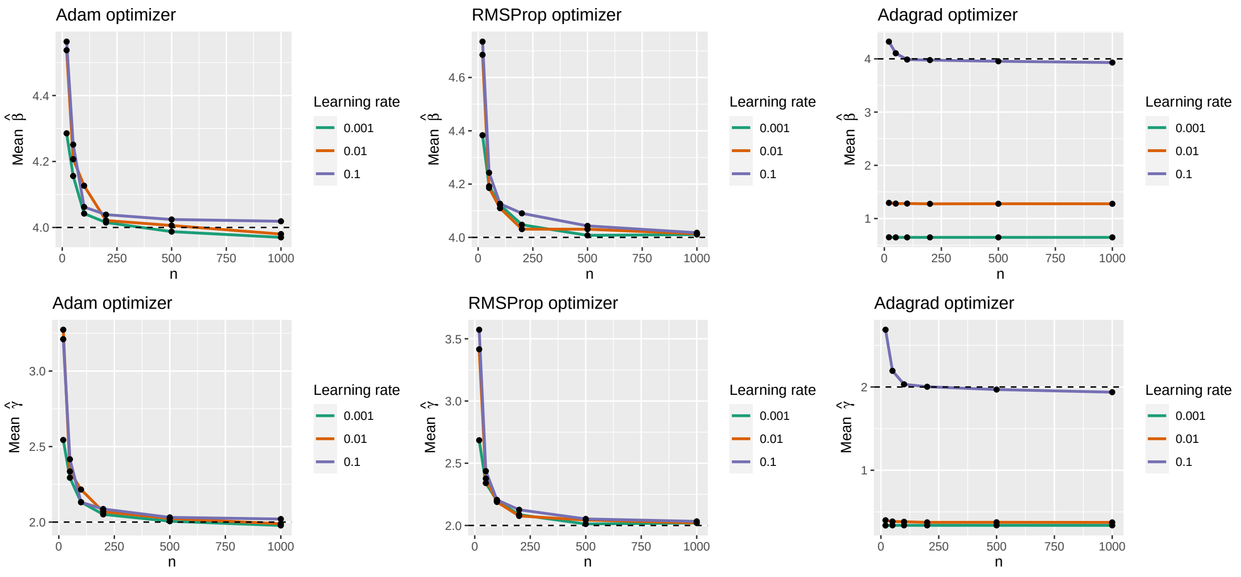

Figure 5-7.: Mean of estimates of $\beta = 4$ and $\gamma = 2$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

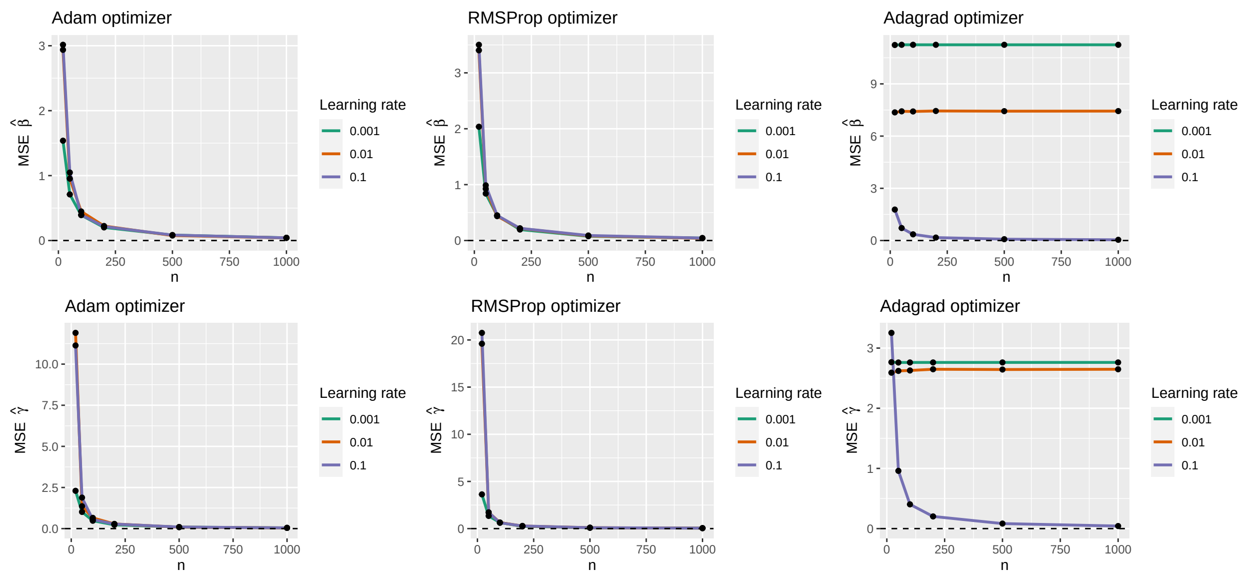

Figure 5-8.: MSE of estimates of $\beta = 4$ and $\gamma = 2$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

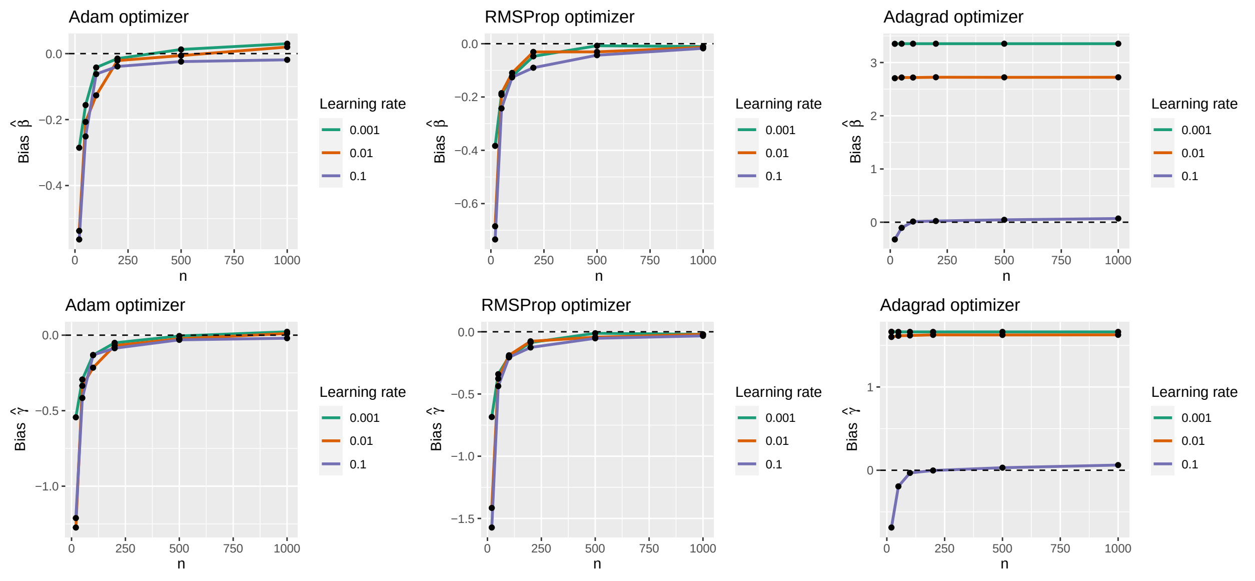

Figure 5-9.: Bias of estimates of $\beta = 4$ and $\gamma = 2$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

5.7.4. Results scenario 4

Figure **5-10** shows the mean of the maximum likelihood estimates $\hat{\beta}_{10}$ and $\hat{\beta}_{11}$, obtained with different TensorFlow optimizers, sample sizes and learning rates. We observe that for β_{10} with the Adam optimizer and with the three different values of the learning rate, as the sample size increases, the estimates are closer to the real values of the parameter. For β_{11}, the estimates are close to its true value with the three different values of the learning rate and regardless of the sample size. With the RMSProp optimizer, we observe the same behavior as with the Adam optimizer when using a learning rate of 0.01 or 0.001. With a learning rate of 0.1, the mean of the estimates of β_{10} and β_{11} deviates from the expected values. On the contrary, with the Adagrad optimizer we obtained good estimates only with a learning rate of 0.1. Therefore, as in the previous scenarios, with this distribution in particular when using the Adagrad optimizer it is recommended to start with a higher learning rate as it decreases significantly throughout the iterations.

Figure **5-11** shows the mean of the maximum likelihood estimates $\hat{\beta}_{20}$ and $\hat{\beta}_{21}$, obtained with different TensorFlow optimizers, sample sizes and learning rates. We observe that for β_{20} and β_{21}, with the Adam optimizer and the RMSProp optimizer and specially with a learning rate of 0.01 or 0.001, as the sample size increases, the estimates are closer to the real value of the parameters. On the contrary, with the Adagrad optimizer we obtained estimates close to the real values of the parameters only when using a learning rate of 0.1.

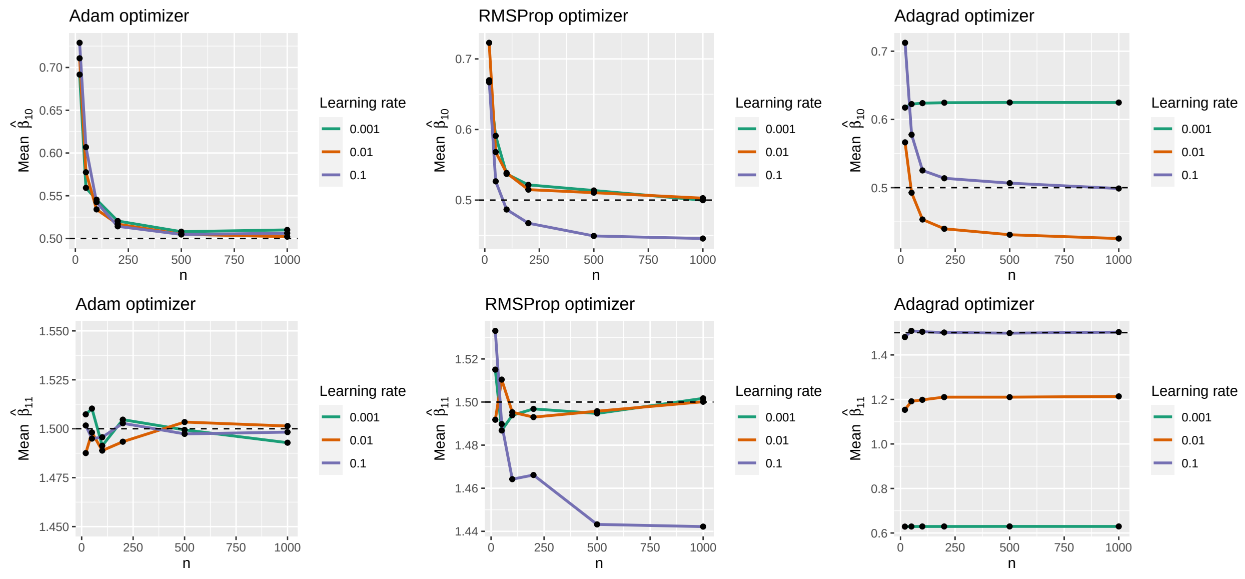

Figure 5-10.: Mean of estimates of $\beta_{10} = 0.5$ and $\beta_{11} = 1.5$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

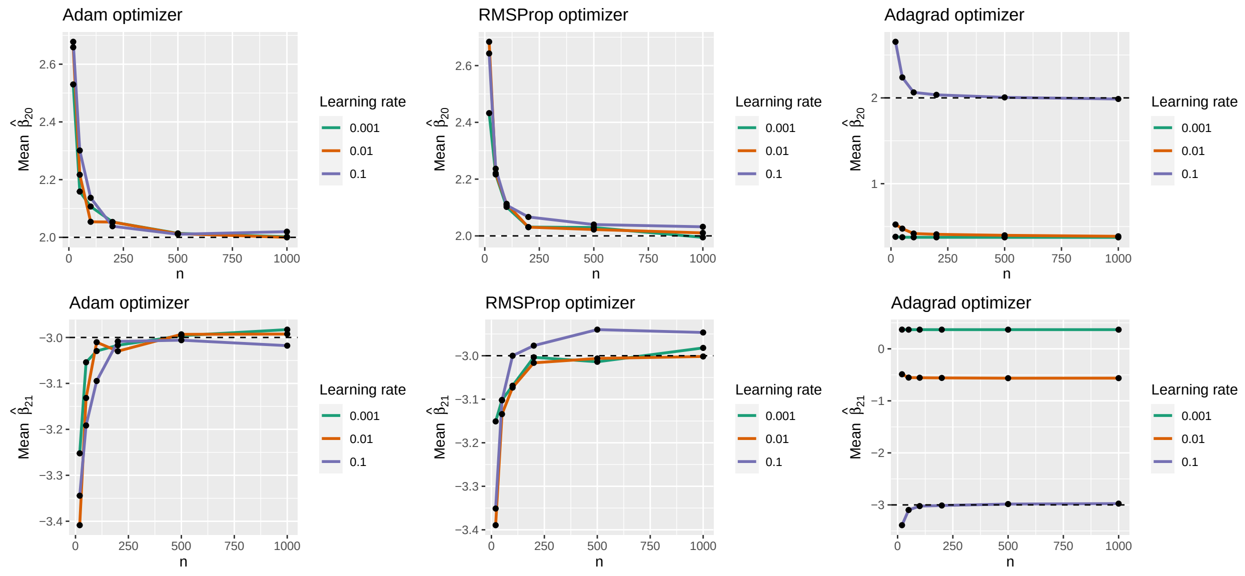

Figure 5-11.: Mean of estimates of $\beta_{20} = 2$ and $\beta_{21} = -3$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

Figure **5-12** shows the MSE of the maximum likelihood estimates $\hat{\beta}_{10}$ and $\hat{\beta}_{11}$, obtained with different TensorFlow optimizers, sample sizes and learning rates. From Figure **5-12** we observe that the MSE of $\hat{\beta}_{10}$ and $\hat{\beta}_{11}$ decreases to a value very close to 0 as the sample size increases with the Adam optimizer and the RMSProp optimizer regardless of the value of the learning rate. In the case of the Adagrad optimizer, we observe that the MSE of $\hat{\beta}_{10}$ and the MSE of $\hat{\beta}_{11}$ get very close to 0 as n increases only when using a learning rate of 0.1. It is important to notice that the MSE of $\hat{\beta}_{10}$ when using a learning rate of 0.01 or 0.001 and the Adagrad optimizer is close to 0 for different values of the sample size, which contradicts what is observed in the Figure **5-10**. This is because the initial value for all the regression parameters were set at 0.5 and for this parameter in particular this value is equal to the real value of the parameter. As these learning rates are small, the value of the parameter does not vary much with respect to its initial value during the estimation process and therefore, the MSE is small regardless of the conditions.

Figure **5-13** shows the MSE of the maximum likelihood estimates $\hat{\beta}_{20}$ and $\hat{\beta}_{21}$, obtained with different TensorFlow optimizers, sample sizes and learning rates. From Figure **5-13** we observe that the MSE of $\hat{\beta}_{20}$ and $\hat{\beta}_{21}$ decreases to a value very close to 0 as the sample size increases with the Adam optimizer and the RMSProp optimizer regardless of the value of the learning rate. In the case of the Adagrad optimizer, only with a learning rate of 0.1 we obtained a MSE close to 0 when $n > 50$. Also, it is important to notice that the MSE of the estimates of parameters β_{20} and β_{21} when the sample size is small and regardless of the learning rate and the optimizer, is higher than the MSE of the estimates of parameters β_{10} and β_{11} in the same conditions. This allows us to conclude that in general the performance of the `mlereg_tf` function in estimating the parameter β_{10} and β_{11} is much better than in estimating parameters β_{20} and β_{21} when the sample sizes are small.

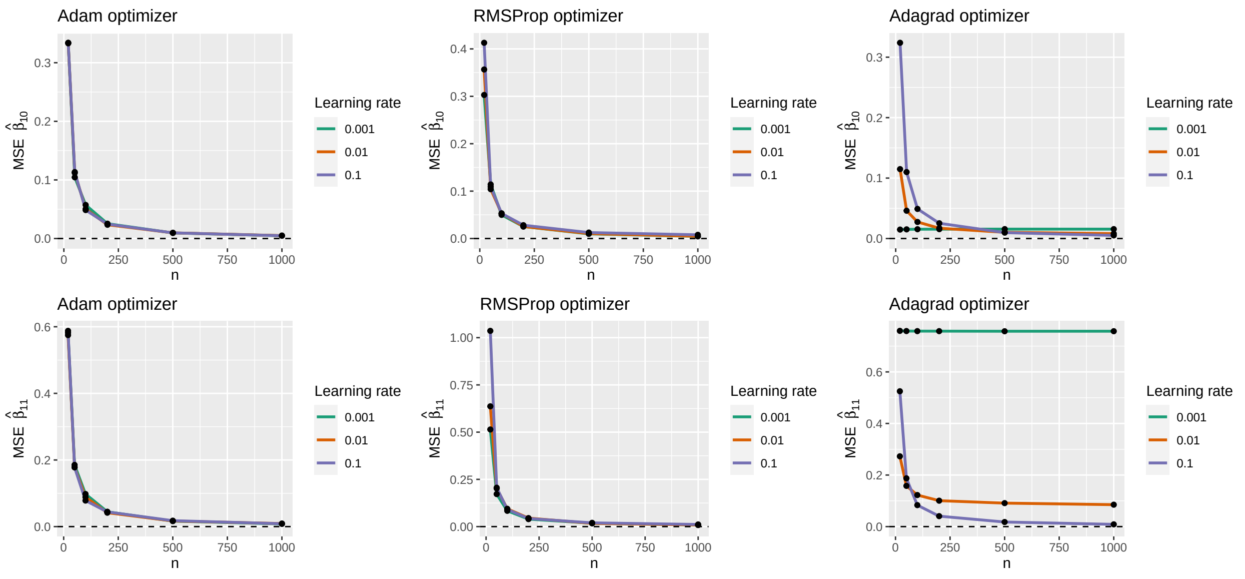

Figure 5-12.: MSE of estimates of $\beta_{10} = 0.5$ and $\beta_{11} = 1.5$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

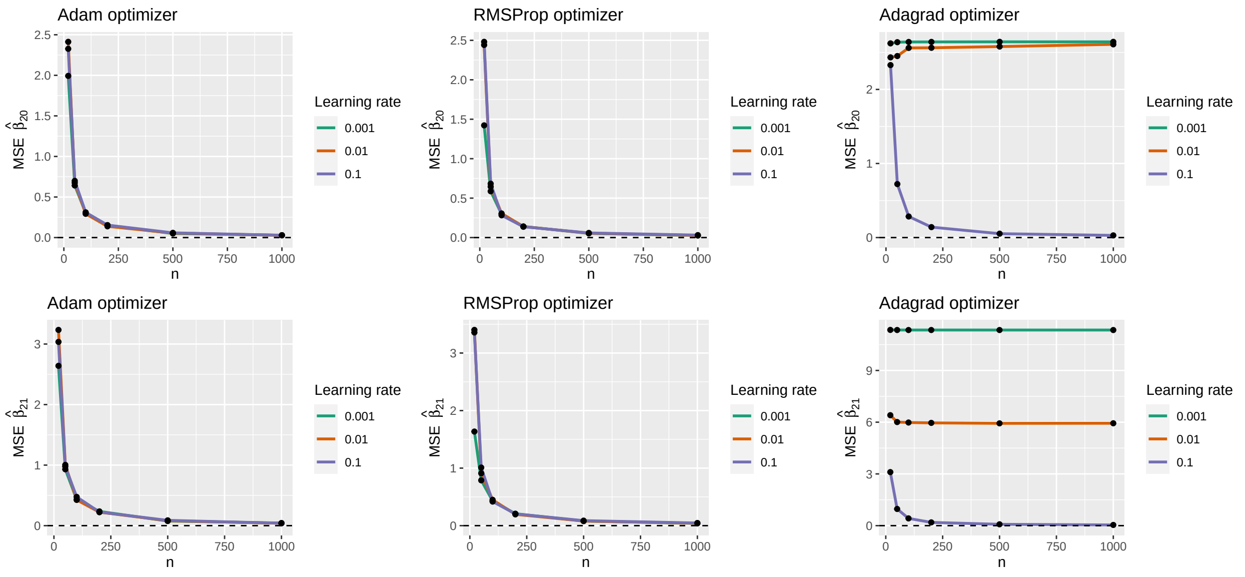

Figure 5-13.: MSE of estimates of $\beta_{20} = 2$ and $\beta_{21} = -3$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

Figure **5-14** shows the bias of the maximum likelihood estimates $\hat{\beta}_{10}$ and $\hat{\beta}_{11}$, obtained with different TensorFlow optimizers, sample sizes and learning rates. From Figure **5-14** we observe that with the Adam optimizer, the Bias of $\hat{\beta}_{10}$ approaches 0 as n increases with the three different values of the learning rate. On the other hand, the Bias of $\hat{\beta}_{11}$ fails to approach 0 in most cases. When using the RMSProp optimizer, we observed that with a learning rate of 0.1 the Bias differs always from 0 especially with large sample sizes. Finally, in the case of the Adagrad optimizer, only with a learning rate of 0.1 we observe the expected results, that is, the Bias gets closer to 0 as the sample size increases.

Figure **5-15** shows the bias of the maximum likelihood estimates $\hat{\beta}_{20}$ and $\hat{\beta}_{21}$, obtained with different TensorFlow optimizers, sample sizes and learning rates. From Figure **5-15** we observe that with the Adam optimizer, the Bias of $\hat{\beta}_{20}$ and $\hat{\beta}_{21}$ approaches 0 as n increases with the three different values of the learning rate. With the RMSProp optimizer we observed this same behavior especially with learning rates of 0.01 and 0.001. On the other hand, when using the Adagrad optimizer and a learning rate of 0.1 the Bias gets closer to 0 as the sample size increases, however, with a lower learning rate the Bias is far from 0 regardless of the sample size.

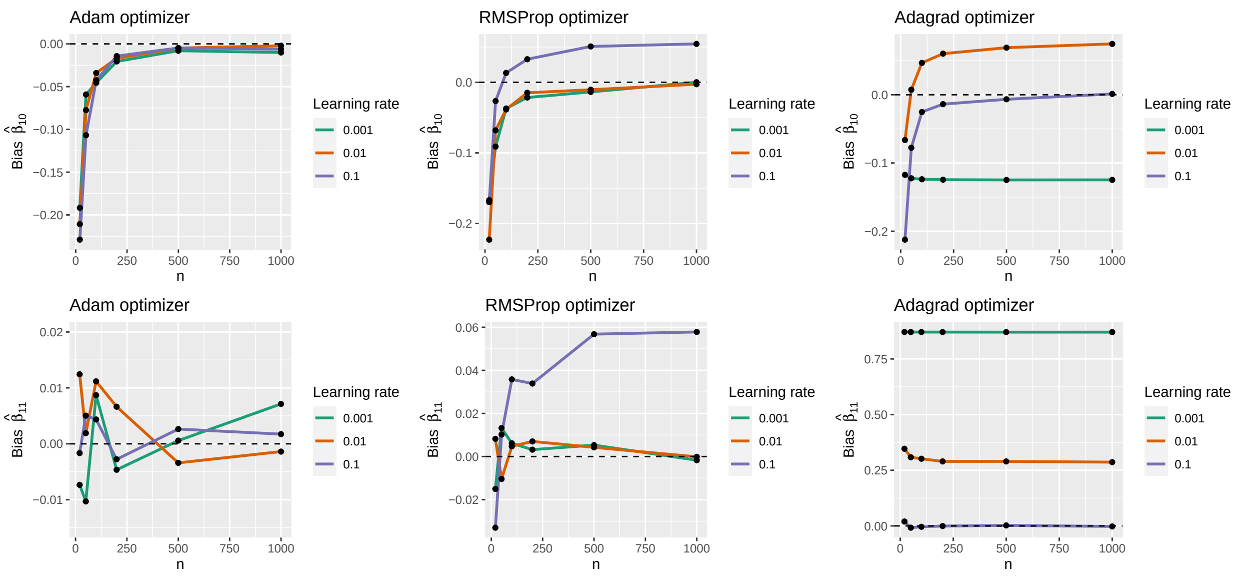

Figure 5-14.: Bias of estimates of $\beta_{10} = 0.5$ and $\beta_{11} = 1.5$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

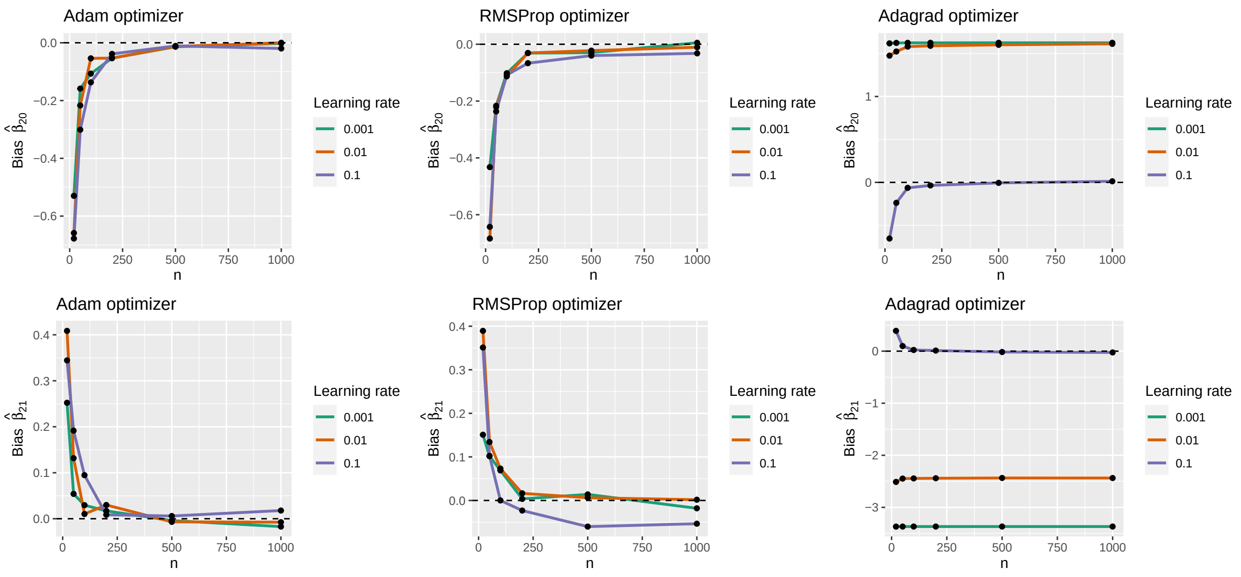

Figure 5-15.: Bias of estimates of $\beta_{20} = 2$ and $\beta_{21} = -3$ for N simulated samples considering different values of the sample size n, the learning rate and the optimizer.

In this scenario we obtained estimates close to the true values of the regression parameters β_{10} and β_{11} with the Adam optimizer and the RMSProp optimizer for specific values of the learning rate. In the case of the Adagrad optimizer it is very important to use an initial learning rate greater than 0.01 to obtain more accurate estimates of these parameters. For parameters β_{20} and β_{21}, we observed that the three optimizers did not performed as expected when the sample sizes were small. Otherwise, with $n > 50$, a behavior similar to that obtained in the estimation of the parameters β_{10} and β_{11} was observed with the Adam optimizer, the RMSProp optimizer and the Adagrad optimizer.

6. Applications

As mentioned in the **estimtf** package section, one of the main features of the **estimtf** package is the possibility to estimate parameters from distributions that are not yet implemented in R. In this chapter the `mle_tf` function is used to estimate parameters from three different probability distributions not implemented in R using real data sets. Also, the `mlereg_tf` function is used to estimate the coefficients of 2 linear regression models. On the other hand, we considered important to compare the estimates obtained with the `mle_tf` function with estimates provided by other R functions that use different optimization methods to maximize the likelihood function. Therefore, we estimated the parameters using the `nlm,` `nlminb` and `optim` functions. In the case of the `optim` function, the L-BFGS-B method and the Nelder-Mead method are used. In the case of the linear regression models, we compared the results with the `glm` function and the `gamlss` functions.

To estimate the parameters we took into account the results obtained in the simulation study. Therefore, we set the initial learning rate as 0.01 and we selected the Adam optimizer for the estimation process. Also, we set the maximum number of iterations at 10000. Finally, as in the simulation study, we set the initial values of the parameters at 0.5 for all the applications.

6.0.1. Extended exponential geometric distribution

As mentioned in the previous chapter, the extended exponential geometric distribution is a distribution with parameters γ and β proposed by Adamidis et al. (2005). To illustrate the use of the **estimtf** package we consider a real data set analyzed by Louzada et al. (2016). This data set refers to the ages in months of 18 patients who died of causes other than cancer.

0.3	4	7.4	15.5	23.4	46	46	51	65
68	83	88	96	110	111	112	132	162

Table 6-1.: Data set of ages (in months) of 18 patients who died of causes other than cancer.

The following is the code written in the R programming language for estimating the parameters of the EEG distribution using the `mle_tf` function of the **estimtf** package. First, we have to load the package and create a vector with the data to be fitted (Table **6-1**). Then we have to create an R function that contains the probability density function of the EEG distribution and whose arguments are the random variable x and the parameters of this distribution. Finally, we have to provide values for the arguments of the `mle_tf` function. In this case, we set the initial values for both parameters as 0.5.

```
# Call required libraries
library(estimtf)

# Create vector with data
x <- c(0.3, 4, 7.4, 15.5, 23.4, 46, 46, 51, 65, 68, 83, 88, 96, 110,
       111, 112, 132, 162)

# Define the probability density function of the EEG distribution
deeg <- function(x, beta, gamma) {
(beta * gamma * exp(-beta * x))/(1 - (1 - gamma) * exp(-beta * x))^2
}

# Use mle_tf function to estimate the parameters
estimation <- mle_tf(x = x, xdist = deeg,
                     initparam = list(beta = 0.5, gamma = 0.5),
                     optimizer = "AdamOptimizer",
                     hyperparameters = list(learning_rate=0.01),
                     maxiter = 10000)
```

Code 6.1: R code for the estimation of parameters λ and γ from the EEG distribution using the `mle_tf` function.

In Code 6.2 the R output obtained by estimating the parameters β and γ of the EEG distribution using the `mle_tf` function is presented. From the summary we observe that the maximum likelihood estimates of these parameters are $\hat{\beta} = 0.0252$ and $\hat{\gamma} = 3.5521$.

```
# print the results
summary(estimation)

## Number of observations: 18
## TensorFlow optimizer: AdamOptimizer
## Negative log-likelihood: 92.7191
## ---------------------------------------------------
##       Estimate  Std. Error Z value Pr(>|z|)
## beta   0.025283   0.008111   3.117  0.00183 **
## gamma  3.545941   2.873021   1.234  0.21712
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Code 6.2: Summary of the estimates of parameters β and γ using the `mle_tf` function.

In Code 6.3 we present the R code for estimating parameters β and γ using the `nlm`, the `nlminb` and the `optim` function. To estimate parameters with these functions it is required to provide the negative log-likelihood function and an initial values for each parameter. In the case of the `nlminb` function and the `optim` function, lower and upper limits for the parameters are also required.

```
# Define the negative log-likelihood function
loglike_fun <- function(param) {
                beta <- param[1]
                gamma <- param[2]
                -sum(log((beta * gamma * exp(-beta * x)) /
                (1 - (1 - gamma) * exp(-beta * x))^2))}

# Estimation with nlm function
result_nlm <- nlm(f = loglike_fun, p = c(0.5, 0.5))

# Estimation with nlminb function
result_nlminb <- nlminb(objective = loglike_fun, start = c(0.5, 0.5),
                        lower = c(0.001, 0.001), upper = c(Inf, Inf))

# Estimation with optim function
result_optim1 <- optim(par = c(0.5, 0.5), fn = loglike_fun,
                       method = "Nelder-Mead")

result_optim2 <- optim(par = c(0.5, 0.5), fn = loglike_fun,
                       method = "L-BFGS-B", lower = c(0.001, 0.001),
                       upper = c(Inf, Inf))
```

Code 6.3: R code for the estimation of parameters λ and γ from the EEG distribution using the `nlm, nlminb, optim` functions.

It is important to notice that unlike the `nlm`, the `nlminb` and the `optim` function, when using the `mle_tf` or the `mlereg_tf` function, the user does not have to compute the log-likelihood function since this task is part of the estimation process implemented in these functions. This is of one of the main advantages of the `mle_tf` and the `mlereg_tf` function because it is possible that some users do not know how to compute the log-likelihood function in R and this could make the estimation of parameters more difficult when using the `nlm`, the `nlminb` or the `optim` function.

In Table **6-2** we compared the estimates obtained from the different R functions and their respective execution times. From Table **6-2**, we observed that the estimates of both parameters obtained from the `nlm, optim` and `nlminb` functions are very similar to each other while the estimates obtained with the `mle_tf` function differ from these estimates although the difference is not very significant. On the other hand, we observed a significant difference in the execution time of the `mle_tf` function with respect to the other functions. This high execution time is due to the fact that the chosen learning rate is relatively small and as mentioned above, a small learning rate may result in a long optimization process. Also, the maximum number of iterations is high and this may also increase the execution time. Therefore, if we want to reduce this time we must select a higher learning rate or a lower number of iterations.

R function	**Execution time (seconds)**	$\hat{\beta}$	$\hat{\gamma}$
`mle_tf`	30.4079	0.0252	3.5459
`nlm`	0.1069	0.0261	3.9192
`optim (Nelder-Mead)`	0.0293	0.0261	3.9289
`optim (L-BFGS-B)`	0.0037	0.0261	3.9299
`nlminb`	0.0297	0.0261	3.9199

Table 6-2.: Processing time and estimates of the parameters of the EEG distribution obtained with R functions: `mle_tf, nlm, optim, nlminb`.

Figure **6-1** shows the histogram of the data presented in Table **6-1** and the estimated density curves obtained with the R functions `mle_tf, nlm, optim` and `nlminb`. From Figure **6-1**, we observed that the 4 density curves are very similar and show a good fit. In this particular case, it can be concluded that to estimate the parameters β and γ of the EEG distribution, it is possible to use any of the functions presented above since they have a similar performance in terms of the quality of the estimates. However, in terms of execution time, the `nlm`, `optim` and `nlminb` functions would be preferred.

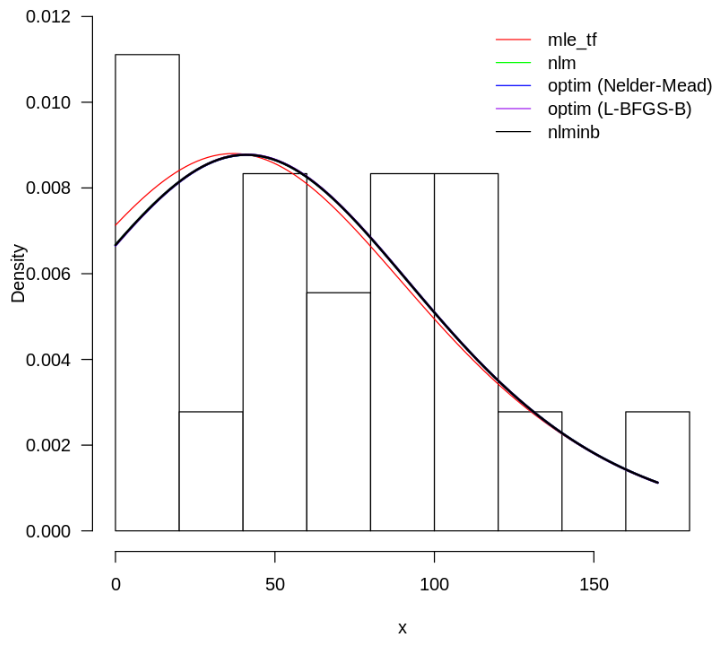

Figure 6-1.: Histogram and estimated density curves super-imposed over the data presented in Table **6-1**.

6.0.2. Distribution for instantaneous failures

Ramos and Louzada (2019) proposed a new distribution with parameter λ that allows for the occurrence of instantaneous failures that are natural in many areas. This distribution is a combination of a reparametrized version of the Zakerzadeh and Dolati distribution with a particular case of the gamma model (Zakerzadeh & Dolati, 2009). A non-negative random variable X with a distribution for instantaneous failures has the following probability density function:

$$f(x|\lambda) = \frac{1}{\lambda^2(\lambda - 1)}(\lambda^2 + x - 2\lambda)e^{-\frac{x}{\lambda}}, \tag{6-1}$$

where $x \geq 0$ and $\lambda \geq 2$ is the shape parameter. To illustrate the use of the **estimtf** package, an example will be carried out using a data set analysed by Muralidharan and Khabia (2014) which represent monthly rainfall (in mm) during year 2004 in Andhra Pradesh. This data set was also analyzed by Ramos and Louzada (2019).

3.4	0.0	0.0	15.8	232.8	8.8
123.2	47	154	103.2	89.8	12.2

Table 6-3.: Data set of monthly rainfall (in mm) during year 2004 in Andhra Pradesh.

The following is the code written in the R programming language for estimating the parameter of the distribution for instantaneous failures using the `mle_tf` function of the **estimtf** package. As $\lambda \geq 2$, for its initial value we draw a sample of size 1 from a normal distribution with $\mu = 5$ and $\sigma = 1$. Thus, the initial value for λ was set as 6.1661.

```
# Call required libraries
library(estimtf)

# Create vector with data
x <-  c(3.4, 0.0, 0.0, 15.8, 232.8, 8.8, 123.2, 47, 154,
        103.2, 89.8,  12.2)

# Define the probability density function of the distribution for
   instantaneous failures
dinstantaneousfailures <- function(x, lambda) {
(1 / ((lambda ^ 2) * (lambda - 1))) * (lambda ^ 2 + x - 2 * lambda) *
exp(-x / lambda)
}
# Use mle_tf function to estimate the parameters
estimation <- mle_tf(x = x,
                     xdist = dinstantaneousfailures,
                     initparam = list(lambda = 6.1661),
                     bounds = list(lambda = c(2, Inf)),
                     optimizer = "AdamOptimizer",
                     hyperparameters = list(learning_rate = 0.01),
                     maxiter = 10000)
```

Code 6.4: R code for the estimation of parameter λ of the distribution for instantaneous failures using the `mle_tf` function.

In Code 6.5 the R output obtained by estimating the parameter λ of the distribution for instantaneous failures using the `mle_tf` function is presented. From the summary we observe that the maximum likelihood estimate of this parameter is $\hat{\lambda} = 65.08$. It is important to notice that the standard error, Z-value and p-value are `NA`. As mentioned in chapter 4, when bounds are defined for the parameter to be estimated, the standard error and consequent statistics values are not presented in the summary because as an internal transformation of the parameter is made, these values do not correspond to those of the parameter in its original scale.

```
# print the results
summary(estimation)

## Number of observations: 12
## TensorFlow optimizer: AdamOptimizer
## Negative log-likelihood: 62.2489
## --------------------------------------------------
##         Estimate  Std. Error Z value Pr(>|z|)
## lambda     65.08          NA      NA       NA
```

Code 6.5: Summary of the estimates of parameter λ using the `mle_tf` function.

In Code 6.6 we present the R code for estimating parameter λ using the `nlm`, the `nlminb` and the `optim` function.

```
# define the negative log-likelihood function
loglike_fun <- function(param) {
                lambda <- param
                -sum(log((1 / ((lambda ^ 2) * (lambda - 1))) *
                (lambda^2 + x - 2*lambda) * exp(-x/lambda)))}

# estimation with nlm function
result_nlm <- nlm(f = loglike_fun, p = 6.166147)

# estimation with nlminb function
result_nlminb <- nlminb(objective = loglike_fun, start = 6.166147,
                        lower = 2, upper = Inf)

# estimation with optim function
result_optim1 <- optim(par = 6.166147, fn = loglike_fun,
                       method = "Nelder-Mead")

result_optim2 <- optim(par = 6.166147, fn = loglike_fun,
                       method = "L-BFGS-B", lower = 2, upper = Inf)
```

Code 6.6: R code for the estimation of parameter λ from the distribution for instantaneous failures using the `nlm, nlminb, optim` functions.

In Table **6-4** we compared the estimates obtained with the different R functions and their respective execution times. From Table **6-4**, we observed that the estimates of the parameter λ obtained with the `mle_tf, nlm, optim` and `nlminb` functions are very similar. On the other hand, we observed a difference in the execution time of the `mle_tf` function with respect to the other functions. Although this difference is not very large, we could try to decrease the execution time of this function by increasing the learning rate or by decreasing the number of iterations.

R function	**Execution time (seconds)**	$\hat{\lambda}$
`mle_tf`	4.06	65.08
`nlm`	0.0211	64.8414
`optim (Nelder-Mead)`	0.0201	64.8601
`optim (L-BFGS-B)`	0.0026	64.8414
`nlminb`	0.0081	64.8414

Table 6-4.: Processing time and estimates of the parameters of the distribution for instantaneous failures obtained with R functions: `mle_tf, nlm, optim, nlminb`.

Figure **6-2** shows the histogram of the data presented in Table **6-3** and the estimated density curves obtained with the R functions `mle_tf, nlm, optim` and `nlminb`. From Figure **6-2**, we observed that the density curves built from the estimates provided by the `mle_tf, nlm, optim` and `nlminb` functions are very similar as expected and they all show a good fit. In this particular case, it can be concluded that to estimate the parameter λ of the distribution for instantaneous failures, it is possible to use any of the functions presented above since they have a similar performance in terms of the quality of the estimates. However, in terms of execution time, the `nlm`, `optim` and `nlminb` functions would be preferred.

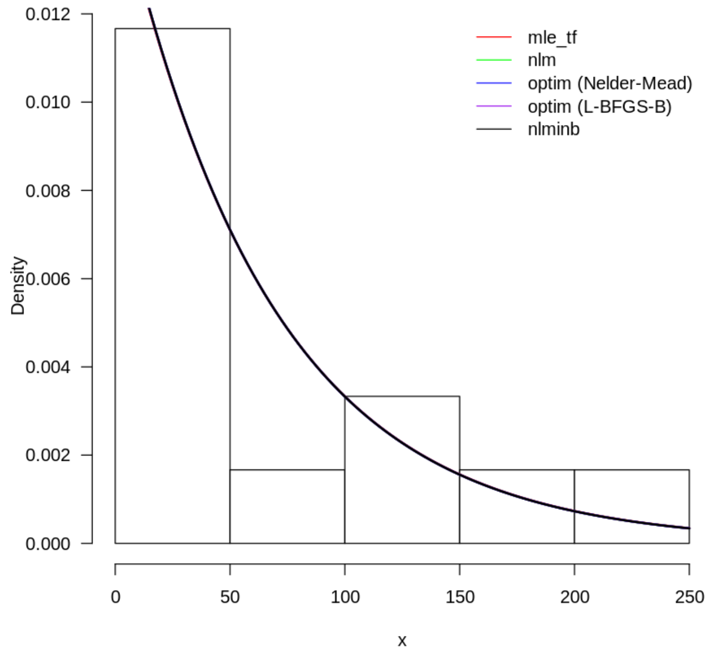

Figure 6-2.: Histogram and estimated density curves super-imposed over the data presented in Table **6-3**.

6.0.3. Transmuted Rayleigh distribution

The transmuted Rayleigh distribution was proposed by Merovci (2013) as an extension of the Rayleigh distribution to provide more flexibility in modeling real data. This distribution is mainly used to model lifetime data which is very important in many applied sciences such as medicine and engineering. A random variable X has a transmuted Rayleigh distribution if its pdf is given by:

$$f(x|\sigma,\lambda) = \frac{x}{\sigma^2}\exp\left(-\frac{x^2}{2\sigma^2}\right)\left(1-\lambda+2\lambda\exp\left(-\frac{x^2}{2\sigma^2}\right)\right), \tag{6-2}$$

where $x > 0$, $\sigma > 0$, and $|\lambda| \leq 1$.

To illustrate the use of the **estimtf** package we consider a real data set analyzed by Dey, Raheem, and Mukherjee (2017). This data set consists of 100 measurements on breaking stress of carbon fibres (in GPa).

0.39	0.81	0.85	0.98	1.08	1.12	1.17	1.18	1.22	1.25
1.36	1.41	1.47	1.57	1.59	1.61	1.69	1.71	1.73	1.80
1.84	1.87	1.89	1.92	2.00	2.03	2.05	2.12	2.17	2.35
2.38	2.41	2.43	2.48	2.50	2.53	2.55	2.56	2.59	2.67
2.73	2.74	2.76	2.77	2.79	2.81	2.82	2.83	2.85	2.87
2.88	2.93	2.95	2.96	2.97	3.09	3.11	3.15	3.19	3.22
3.27	3.28	3.31	3.33	3.39	3.51	3.56	3.60	3.65	3.68
3.70	3.75	4.20	4.38	4.42	4.70	4.90	4.91	5.08	5.56
1.58	1.60	1.61	1.70	1.85	2.04	2.17	2.17	2.48	2.55
2.82	2.98	3.11	3.16	3.19	3.23	3.31	3.39	3.68	3.69

Table 6-5.: Data set of breaking stress of carbon fibres (in GPa).

The following is the code written in the R programming language for estimating the parameters of the transmuted Rayleigh distribution using the `mle_tf` function of the **estimtf** package. In this case, we set the initial values for both parameters as 0.5.

```r
# Call required libraries
library(estimtf)
# Create vector with data
x <- c(0.39,0.81,0.85,0.98,1.08,1.12,1.17,1.18,1.22,1.25,
       1.36,1.41,1.47,1.57,1.59,1.61,1.69,1.71,1.73,1.80,
       1.84,1.87,1.89,1.92,2.00,2.03,2.05,2.12,2.17,2.35,
       2.38,2.41,2.43,2.48,2.50,2.53,2.55,2.56,2.59,2.67,
       2.73,2.74,2.76,2.77,2.79,2.81,2.82,2.83,2.85,2.87,
       2.88,2.93,2.95,2.96,2.97,3.09,3.11,3.15,3.19,3.22,
       3.27,3.28,3.31,3.33,3.39,3.51,3.56,3.60,3.65,3.68,
       3.70,3.75,4.20,4.38,4.42,4.70,4.90,4.91,5.08,5.56,
       1.58,1.60,1.61,1.70,1.85,2.04,2.17,2.17,2.48,2.55,
       2.82,2.98,3.11,3.16,3.19,3.23,3.31,3.39,3.68,3.69)
# Define the probability density function of the transmuted Rayleigh
dtr <- function(x, sigma, lambda) {
(x / sigma^2)*exp(-(x^2) / (2*sigma^2)) * (1 - lambda + 2 * lambda * exp
   (-(x^2) / (2*sigma^2)))
}
# Use mle_tf function to estimate the parameters
estimation <- mle_tf(x = x, xdist = dtr,
                     initparam = list(lambda = 0.5, sigma = 0.5),
                     bounds =  list(lambda = c(-1, 1), sigma = c(0, Inf)),
                     optimizer = "AdamOptimizer",
                     hyperparameters = list(learning_rate = 0.01),
                     maxiter = 10000)
```

Code 6.7: R code for the estimation of parameters λ and σ from the transmuted Rayleigh distribution using the `mle_tf` function.

In Code 6.8 the R output obtained by estimating the parameters σ and λ of the transmuted Rayleigh distribution using the `mle_tf` function is presented. From the summary we observe that the maximum likelihood estimates of these parameters are $\hat{\sigma} = 1.649$ and $\hat{\lambda} = -0.908$.

```
# print the results
summary(estimation)

## Number of observations: 100
## TensorFlow optimizer: AdamOptimizer
## Negative log-likelihood: 141.4435
## ---------------------------------------------------
##        Estimate  Std. Error Z value Pr(>|z|)
## sigma      1.649          NA      NA       NA
## lambda    -0.908          NA      NA       NA
```

Code 6.8: Summary of the estimates of parameters σ and λ using the `mle_tf` function.

In Code 6.9 we present the R code for estimating parameters λ and σ using the `nlm`, the `nlminb` and the `optim` function.

```
# define the negative log-likelihood function
loglike_fun <- function(param) {
                lambda <- param[1]
                sigma <- param[2]
                -sum(log((x / sigma^2)*exp(-(x^2) / (2*sigma^2)) *
                (1 - lambda + 2 * lambda * exp(-(x^2) / (2*sigma^2)))))}

# estimation with nlm function
result_nlm <- nlm(f = loglike_fun, p = c(0.5, 0.5))

# estimation with nlminb function
result_nlminb <- nlminb(objective = loglike_fun, start = c(0.5, 0.5),
                        lower = c(-1, 0.001), upper = c(1, Inf))

# estimation with optim function
result_optim1 <- optim(par = c(0.5, 0.5), fn = loglike_fun,
                       method = "Nelder-Mead")

result_optim2 <- optim(par = c(0.5, 0.5), fn = loglike_fun,
                       method = "L-BFGS-B", lower = c(-1, 0.001),
                       upper = c(1, Inf))
```

Code 6.9: R code for the estimation of parameters λ and σ from the transmuted Rayleigh distribution using the `nlm, nlminb, optim` functions.

In Table **6-6** we compared the estimates obtained with the different R functions and their respective execution times. From Table **6-6**, we observed that the estimates of both parameters obtained with the `mle_tf`, `optim` and `nlminb` functions are very similar to each other while the estimates obtained with the `nlm` function differ significantly from these estimates. This is possibly due to the initial values passed to the function. Finally, as in the two previous applications, the execution time of the `mle_tf` function is higher than the execution time of the other three functions.

R function	**Execution time (seconds)**	$\hat{\lambda}$	$\hat{\sigma}$
`mle_tf`	11.6136	-0.908	1.649
`nlm`	0.1577	13717.2360	4.7465
`optim (Nelder-Mead)`	0.0484	-0.9192	1.6459
`optim (L-BFGS-B)`	0.0034	-0.9189	1.6461
`nlminb`	0.0088	-0.9189	1.6461

Table 6-6.: Processing time and estimates of the parameters of the transmuted Rayleigh distribution obtained with R functions: `mle_tf, nlm, optim, nlminb`.

Figure **6-3** shows the histogram of the data presented in Table **6-5** and the estimated density curves obtained with the R functions `mle_tf, nlm, optim` and `nlminb`. In this case, the density curve for the `nlm` function is not included in this graph because it is very different from the other curves and therefore it is not possible to visualize it in the graph. From Figure **6-3**, we observed that all three density curves corresponding to the `mle_tf, optim` and `nlminb` functions are very similar and show a good fit. In this particular case, it can be concluded that to estimate the parameters σ and λ of the transmuted Rayleigh distribution, it is possible to use any of the functions presented above since they have a similar performance in terms of the quality of the estimates. However, in terms of execution time, the `nlm`, `optim` and `nlminb` functions would be preferred.

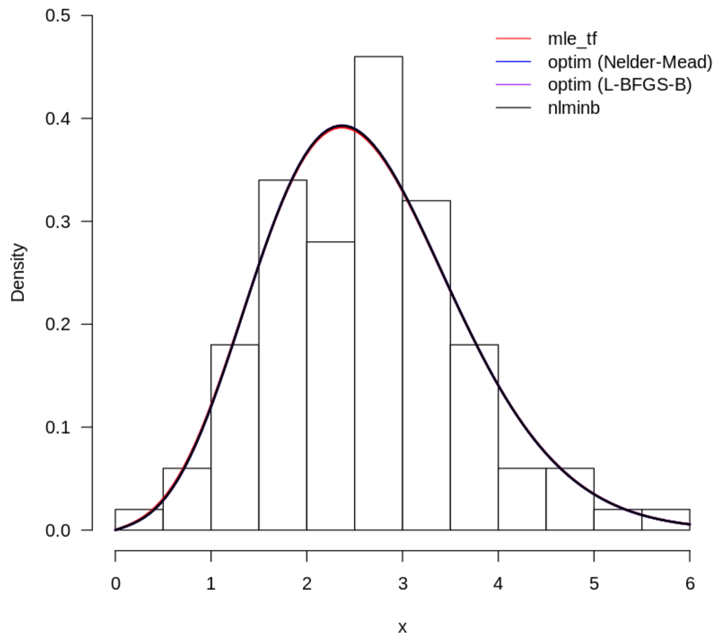

Figure 6-3.: Histogram and estimated density curves super-imposed over the data presented in Table **6-5**.

6.0.4. Poisson regression model

The Poisson regression model assumes that the response variable Y, which is a count and only takes discrete and non-negative values, has a Poisson distribution with probability mass function given by:

$$P(Y = y|\lambda) = \frac{\lambda^y e^{-\lambda}}{y!}, \tag{6-3}$$

where $y = 0, 1, 2, \ldots$; $\lambda \geq 0$ and $E(Y) = V(Y) = \lambda$. In order to incorporate the covariates, the mean is included in the model using a log-link function:

$$log(\lambda_i) = \boldsymbol{X}_i^\top \boldsymbol{\beta} = \sum_{j=1}^{p} \beta_j x_{ij}, \tag{6-4}$$

where $\boldsymbol{X}_i$ denotes the vector of explanatory variables, and $\boldsymbol{\beta}$ denotes the vector of regression parameters.

To illustrate the use of the **estimtf** package we consider the data set from a study of female horseshoe crabs on the Gulf of Mexico presented in Table **5-1**. In this case, we are going to fit a Poisson regression model in which the response variable is $Y = \textit{number of satellites}$ and the explanatory variables are the female crab's color and weight.

$$\begin{aligned} Y_i &\sim Poisson(\lambda_i), \\ \log(\lambda_i) &= \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i3} + \beta_4 x_{i4}, \end{aligned} \tag{6-5}$$

where $i = 1, 2, \ldots 173$, x_{i1} denotes weight and $x_{ij} = 1$ when the female crab has color j and $x_{ij} = 0$ otherwise, for $j = 2, 3, 4$.

The following is the code written in the R programming language for estimating the coefficients $\beta_0, \beta_1, \beta_2, \beta_3$ and β_4 using the `mlereg_tf` function of the **estimtf** package. First, we have to load the package and define the data frame that contains the data for the response variable **y** and the covariates **weight** and **color**. Then, we have to provide values for the arguments of the `mlereg_tf` function. As the Poisson distribution is included in Table **4-1**, we only have to provide the name of the distribution in the `ydist` argument. In this case, we set the initial values for the coefficients as 0.5.

```r
# Call required libraries
library(estimtf)

# Create data frame with data for response explanatory variables
data <- read.table("Crabs.dat", header=T)
data <- data %>% select(y, color, weight)
data$color <- as.factor(data$color)
data <- as.data.frame(data)

# Use mlereg_tf function to estimate the coefficients
estimation <- mlereg_tf(ydist = y ~ Poisson, data = data,
                        formulas = list(lambda = ~color + weight),
                        available_distribution = TRUE,
                        initparam = list(lambda = 0.5),
                        link_function = list(lambda = "log"),
                        optimizer = "AdamOptimizer",
                        hyperparameters = list(learning_rate = 0.01),
                        maxiter = 10000)
```

Code 6.10: R code for the estimation of coefficients of the Poisson regression model using the `mlereg_tf` function.

In Code 6.11 the R output obtained by estimating the coefficients $\beta_0, \beta_1, \beta_2, \beta_3$ and β_4 of the Poisson regression model using the `mlereg_tf` function is presented. From the summary we observe that the maximum likelihood estimates of these parameters are $\hat{\beta}_0 = -0.0070$, $\hat{\beta}_1 = 0.5354$, $\hat{\beta}_2 = -0.2181$, $\hat{\beta}_3 = -0.4663$ and $\hat{\beta}_4 = -0.4706$.

```
# print the results
summary(estimation)

## Distribution: Poisson
## Number of observations: 173
## TensorFlow optimizer: AdamOptimizer
## ---------------------------------------------------------------
## Distributional parameter: lambda
## ---------------------------------------------------------------
##             Estimate. Std..Error Z.value Pr...z..
## (Intercept) -0.007007   0.233010  -0.030  0.97601
## color2      -0.218131   0.152793  -1.428  0.15340
## color3      -0.466363   0.174963  -2.665  0.00769 **
## color4      -0.470660   0.207808  -2.265  0.02352 *
## weight       0.535451   0.068366   7.832  4.8e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## ---------------------------------------------------------------
```

Code 6.11: Summary of the estimates of coefficients of the Poisson regression model using the `mlereg_tf` function.

In Table **6-7** we compared the estimates obtained from the different R functions and their respective execution times. From Table **6-7**, we observed that the estimates of the coefficients $\beta_1, \beta_2, \beta_3$ and β_4 obtained with the `mlereg_tf` function and with the `glm` function are very similar. However, in the case of the coefficient β_0, the difference between the estimates obtained with both functions is greater.

In this particular case, based on the results presented in Table **6-7** it can be concluded that to estimate the coefficients $\beta_0, \beta_1, \beta_2, \beta_3$ and β_4 of this Poisson regression model (6-5), it is possible to use the `glm` function or the `mlereg_tf` function since they have a similar performance in terms of the quality of the estimates. However, in terms of execution time, the `glm` function would be preferred.

R function	**Execution time (seconds)**	$\hat{\beta}_0$	$\hat{\beta}_1$	$\hat{\beta}_2$	$\hat{\beta}_3$	$\hat{\beta}_4$
`mleref_tf`	20.5997	-0.0070	0.5354	-0.2181	-0.4663	-0.4706
`glm`	0.0135	-0.0497	0.5461	-0.2051	-0.4498	-0.4520

Table 6-7.: Processing time and estimates of the coefficients of the Poisson regression model obtained with R functions: `mlereg_tf, glm`.

6.0.5. Flexible Weibull extension regression model

The flexible Weibull extension (FWE) distribution is a distribution with two parameters μ and σ proposed by Bebbington, Lai, and Zitikis (2007). Its probability density function is given by:

$$f(y|\mu,\sigma) = \left(\mu + \frac{\sigma}{y^2}\right) e^{\left(\mu y - \frac{\sigma}{y}\right)} \exp\left(-e^{\mu y - \frac{\sigma}{y}}\right), \tag{6-6}$$

where $y > 0$, $\mu > 0$ and $\sigma > 0$. The FWE regression model assumes that the response variable Y has a FWE distribution as follows:

$$\begin{aligned} y_i &\sim FWE(\mu_i, \sigma_i), \\ \log(\mu_i) &= \boldsymbol{X}_i^\top \boldsymbol{\beta} = \beta_0 + \sum_{j=1}^{p} \beta_j X_{ij}, \\ \log(\sigma_i) &= \boldsymbol{Z}_i^\top \boldsymbol{\rho} = \rho_0 + \sum_{j=1}^{p} \rho_j Z_{ij}, \end{aligned} \tag{6-7}$$

where $\boldsymbol{X}_i$, $\boldsymbol{Z}_i$ denote the vectors of explanatory variables, and $\boldsymbol{\beta}$, $\boldsymbol{\rho}$ denote the vectors of regression parameters.

To illustrate the use of the **estimtf** package we generate $n = 50$ observations from the FWE regression model (6-8) using the `rFWE` function of the **RelDists** package which contains multiple distributions that are very useful for reliability analysis (Hernandez, Usuga, Patino, Mosquera, & Urrea, 2021).

$$
\begin{aligned}
y_i &\sim FWE(\mu_i, \sigma_i), \\
\log(\mu_i) &= \beta_{10} + \beta_{11}x_{1i} = 1.21 - 3 \times x_{1i}, \\
\log(\sigma_i) &= \beta_{20} + \beta_{21}x_{2i} = 1.26 - 2 \times x_{2i}, \\
x_1 &\sim U(0,1), \\
x_2 &\sim U(0,1),
\end{aligned}
\tag{6-8}
$$

where $i = 1, 2, \ldots 50$. In Table **6-1** we observe the data for the response variable Y and the explanatory variables X_1 and X_2 of the FWE regression model presented above.

Observation	**y**	**x1**	**x2**	**Observation**	**y**	**x1**	**x2**
1	0.526	0.313	0.942	26	0.569	0.431	0.878
2	1.238	0.346	0.201	27	1.926	0.716	0.556
3	0.436	0.421	0.695	28	0.927	0.381	0.484
4	0.725	0.695	0.604	29	0.384	0.003	0.732
5	0.839	0.174	0.821	30	0.717	0.242	0.292
6	0.526	0.374	0.596	31	0.929	0.775	0.798
7	0.768	0.427	0.975	32	0.601	0.952	0.97
8	3.035	0.491	0.008	33	0.882	0.42	0.266
9	0.79	0.953	0.676	34	2.775	0.416	0.059
10	0.973	0.139	0.5	35	0.899	0.418	0.281
11	1.737	0.614	0.306	36	1.035	0.231	0.348
12	0.998	0.093	0.161	37	4.031	0.848	0.218
13	1.657	0.823	0.913	38	1.353	0.523	0.134
14	0.547	0.117	0.661	39	1.089	0.384	0.432
15	3.96	0.887	0.234	40	1.985	0.451	0.038
16	0.688	0.116	0.131	41	0.221	0.08	0.967
17	0.515	0.643	0.65	42	0.61	0.488	0.188
18	1.84	0.878	0.468	43	2.678	0.785	0.458
19	0.949	0.203	0.395	44	0.903	0.09	0.289
20	0.988	0.314	0.577	45	1.991	0.71	0.543
21	0.255	0.512	0.705	46	2.979	0.708	0.448
22	1.307	0.16	0.095	47	3.624	0.875	0.068
23	3.67	0.662	0.047	48	0.319	0.728	0.858
24	4.189	0.888	0.87	49	2.343	0.818	0.937
25	1.885	0.688	0.014	50	1.361	0.532	0.706

Table 6-8.: Data with 50 observations on 3 variables from the FWE regression model (6-8).

The following is the code written in the R programming language for estimating the coefficients $\beta_{10}, \beta_{11}, \beta_{20}$ and β_{21} using the `mlereg_tf` function of the **estimtf** package. In this case, we have to create an R function that contains the probability density function of the FWE distribution (6-6). On the other hand, as in the previous applications, we set the initial values for the coefficients as 0.5.

```r
# Call required libraries
library(estimtf)

# Define the probability density function of the FWE distribution
dFWE <- function(y, mu, sigma){
  (mu + sigma/(y^2))*(exp(mu * y - sigma/y))*exp(-exp(mu * y - sigma/y))
}

# Use mlereg_tf function to estimate the coefficients
# data: an R data frame containing the data presented in Table 6-9
estimation <- mlereg_tf(ydist = y ~ dFWE,
                        formulas = list(mu = ~ x1, sigma = ~x2),
                        available_distribution = FALSE,
                        data = data,
                        initparam = list(mu = 0.5, sigma = 0.5),
                        optimizer = "AdamOptimizer",
                        link_function = list(mu = "log", sigma = "log"),
                        hyperparameters = list(learning_rate = 0.01),
                        maxiter = 10000)
```

Code 6.12: R code for the estimation of coefficients of the FWE regression model using the `mlereg_tf` function.

In Code 6.13 the R output obtained by estimating the coefficients $\beta_{10}, \beta_{11}, \beta_{20}$ and β_{21} of the FWE regression model using the `mlereg_tf` function is presented. From the summary we observe that the maximum likelihood estimates of these parameters are $\hat{\beta}_{10} = 1.442$, $\hat{\beta}_{11} = -3.092$, $\hat{\beta}_{20} = 1.671$ and $\hat{\beta}_{21} = -2.411$. By comparing these estimates with the real values of the parameters $\beta_{10} = 1.21, \beta_{11} = -3, \beta_{20} = 1.26$ and $\beta_{21} = -2$ we observe that the estimates obtained using the `mlereg_tf` function are similar to these values even though the sample size is not very large which allows us to conclude that the estimation process was successful.

```
# print the results
summary(estimation)

## Number of observations: 50
## TensorFlow optimizer: AdamOptimizer
## ---------------------------------------------------------------
## Distributional parameter: mu
## ---------------------------------------------------------------
##             Estimate. Std..Error Z.value Pr...z..
## (Intercept)    1.4420     0.1863   7.741  9.9e-15 ***
## x1            -3.0927     0.3507  -8.819  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## ---------------------------------------------------------------
## Distributional parameter: sigma
## ---------------------------------------------------------------
##             Estimate. Std..Error Z.value Pr...z..
## (Intercept)    1.6713     0.1951   8.568  < 2e-16 ***
## x2            -2.4114     0.3521  -6.849 7.42e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## ---------------------------------------------------------------
```

Code 6.13: Summary of the estimates of coefficients of the FWE regression model using the `mlereg_tf` function.

In Table **6-9** we compared the estimates obtained from the different R functions and their respective execution times. From Table **6-9**, we observed that the estimates of most of these coefficients obtained with the **`gamlss`** function are closer to the real values than those obtained with the `mlereg_tf` function. In addition, the execution time of the **`gamlss`** function is significantly lower than the execution time of the **`mlereg_tf`**. Based on these results, it can be concluded that to estimate the coefficients $\beta_{10}, \beta_{11}, \beta_{20}$ and β_{21} of this FWE regression model (6-8), the **`gamlss`** function would be preferred because of the quality of its estimates and its efficiency.

R function	**Execution time (seconds)**	$\hat{\beta}_{10}$	$\hat{\beta}_{11}$	$\hat{\beta}_{20}$	$\hat{\beta}_{21}$
`mleref_tf`	65.76	1.442	-3.092	1.671	-2.411
`gamlss`	0.44	1.073	-2.742	1.249	-2.065

Table 6-9.: Processing time and estimates of the coefficients of the FWE regression model obtained with R functions: `mlereg_tf, gamlss`.

In this last application we observed the higher execution time while using the **estimtf** package. As mentioned earlier, it is possible to try to decrease this time by increasing the learning rate and/or decreasing the maximum number of iterations. Therefore, in Table **6-10** we compared the absolute error of the estimates of each parameter obtained in different scenarios in which we used learning rates higher than 0.01 and maximum number of iterations lower than 10000 which were chosen originally for the estimation process. Regarding the execution time, we observed that the increase in the value of the learning rate actually contributed to the decrease in the execution time. On the other hand, we observed that as the maximum number of iterations decreases, the execution time decreases as well regardless of the learning rate value.

Also, it is important to notice that the highest errors are obtained when using learning rates lower than 0.5 and a number of iterations lower than or equal to 500. This shows that the selection of the learning rate must depend on the number of iterations or vice versa. That is, as we decrease the learning rate we must increase the number of iterations of the estimation process to avoid affecting the quality of the estimations. Finally, in Table **6-10** we observed that most of the absolute errors of the estimates obtained with the different values for the learning rate and the maximum number of iterations, do not vary much with respect to the reference scenario in which we used a learning rate of 0.01 and 10000 iterations. This indicates that for this case in particular we can use other values for these hyperparameters and get good estimates in less time.

With the applications shown in this chapter we can state that it is possible to use the **estimtf** package to find the maximum likelihood estimates of the parameters of multiple distributions or regression models using real data sets. In the case of distributional parameters, with this package we obtained very similar estimates to the ones obtained with functions like `nlm`, `nlminb` and `optim`, which are very popular and have been traditionally used for this task. Finally, in the case of the Poisson regression model we obtained estimates very similar to those of the `glm` function, which is widely used to fit generalized linear models, and in the case of the FWE regression model we concluded that the performance in estimating the coefficients of the `gamlss` function was superior to the performance of the function `mlereg_tf` in terms of the estimates error and execution time.

`learning_rate`	`maxiter`	Execution time (seconds)	β_{10}	β_{11}	β_{20}	β_{21}
	100	9.012	1.518	2.665	0.628	1.514
	500	17.407	1.574	2.304	0.391	0.533
0.01	1000	28.107	1.218	1.919	0.161	0.138
	5000	69.272	0.232	0.093	0.411	0.411
	10000	**65.760**	**0.232**	**0.092**	**0.411**	**0.411**
	100	10.356	1.632	2.327	0.468	0.666
	500	19.228	0.019	0.356	0.325	0.394
0.05	1000	24.726	0.235	0.098	0.412	0.411
	5000	25.731	0.235	0.098	0.412	0.411
	10000	26.182	0.235	0.098	0.412	0.411
	100	4.141	1.359	2.053	0.207	0.198
	500	9.232	0.236	0.100	0.412	0.411
0.1	1000	9.651	0.236	0.101	0.412	0.411
	5000	10.077	0.236	0.100	0.412	0.411
	10000	18.436	0.236	0.100	0.412	0.411
	100	5.155	0.146	0.009	0.383	0.538
	500	7.595	0.237	0.105	0.412	0.410
0.5	1000	7.885	0.237	0.105	0.412	0.410
	5000	8.090	0.237	0.105	0.412	0.410
	10000	17.323	0.237	0.105	0.412	0.410

Table 6-10.: Absolute error of estimates of β_{10}, β_{11}, β_{20} and β_{21} obtained using the `mlereg_tf()` function and differente values for the learning rate and the maximum number of iterations.

7. Conclusions, recommendations and future work

This chapter provides some conclusions based on the work presented above and recommendations for the use of the **estimtf** package, as well as proposals for future work focused primarily on improving the performance and results obtained with the package.

7.1. Conclusions

In this work we presented the **estimtf** package. This package allows R users to find the maximum likelihood estimates of parameters of probability distributions and regression models using the TensorFlow optimizers. Our main goal was to introduce the main characteristics, functionalities and advantages of the **estimtf** package and illustrate the use of the `mle_tf` function and the `mlereg_tf` function using real datasets. On the other hand, we used a simulation study to show that there are some variables such as the learning rate that can affect the results obtained with this package and that it is possible to obtain good estimates using both functions `mle_tf` and `mlereg_tf`.

Although the TensorFlow library and in particular the TensorFlow optimizers are mostly used to train neural network models, with this work we showed that they can also be implemented for estimating distributional and regression parameters. To achieve this, it was necessary to design an iterative estimation process in which the value of the parameters change from one iteration to another according to an update rule that depends on the optimizer. We were able to implement this estimation process in R through the **estimtf** package taking advantage of packages such as **tensorflow** and **tfprobability**.

One of the main advantages of the **estimtf** package is the possibility of estimating parameters of distributions that are not necessarily implemented in R. In this case, the user must provide an R object of class function with the mass/density function of the distribution of interest. On the other hand, unlike optimization functions such as `optim` or `nlm`, when using the functions from the **estimtf** package, the user does not have to compute the log-likelihood function in order to find the maximum likelihood estimates.

Through a simulation study it was shown that the estimates obtained with the `mle_tf` function and the `mlereg_tf` function are affected by the sample size n, the TensorFlow optimizer and the initial value of the learning rate for the selected optimizer when estimating parameters of the Poisson distribution and the EEG distribution. For both distributions included in the study, we can conclude that when using the Adam optimizer and the RMSProp optimizer, the majority of the estimates obtained with these functions were close to the actual values of the parameters for most learning rate and sample size combinations, especially with large sample sizes. However, for some parameters, the error of the estimates is very high when $n < 50$. On the other hand, with the Adagrad optimizer, only with a learning rate of 0.1, we obtained good estimates. Which lead us to conclude that for distributions like Poisson and EEG, when using the Adagrad optimizer is recommended to set a value higher than 0.01 for the initial learning rate.

In the application section we observe that the maximum likelihood estimates of the parameters of the EEG distribution, the distribution for instantaneous failures and the transmuted Rayleigh distribution, obtained with the `mle_tf` function are very similar to the ones obtained with the `optim, nlm` and `nlminb` functions which use optimization methods such as L-BFGS-B, the Newton-type algorithm and PORT routines respectively and that are very popular and frequently used to solve optimization problems. Also, the estimates of the coefficients of the Poisson regression model obtained with the `mlereg_tf` function are very similar to the ones obtained with the `glm` function which is a very known and widely used function to fit generalized linear models.

Although in the simulation study and in the applications chapter, parameters of only a few probability distributions and linear regression models were estimated, it is important to emphasize that the performance in terms of precision of the estimates of the `mle_tf` and `mlereg_tf` functions in most of the cases was very good. This allows us to begin to trust the estimates provided by the **estimtf** package and to conclude that it is indeed possible to use the TensorFlow optimizers to solve the maximum likelihood problem. This gives R users the possibility to access a reliable tool with which they can find the maximum likelihood estimates of parameters of multiple distributions and regression models in an intuitive and simple way while taking advantage of a very powerful library such as TensorFlow.

Finally, according to the results presented in previous sections it is not possible to conclude that the optimization method implemented in the **estimtf** package is better in terms of precision and efficiency than other traditional methods such as quasi-Newton methods or PORT routines. However, with this work we not only managed to achieve one of the main objectives which was to evaluate whether it is possible to use TensorFlow optimizers to find the maximum likelihood estimates of distributional and regression parameters. We also managed to show that the performance of these algorithms in estimating these parameters is good under certain conditions.

7.2. Recommendations

As the learning rate decreases, the changes in the values of the parameters from one iteration to another are less significant and therefore, the time required for the algorithm to converge is longer. With this in mind, it is recommended to increase the number of iterations of the optimization process as the initial learning rate value is smaller.

As in the previous sections we concluded that the optimizer and the initial value of the learning rate affect the quality of the estimates, it is important to know very well the way in which each optimizer updates the values of the parameters during the optimization process to choose correctly the initial value of the learning rate. For example, in the case of the Adagrad optimizer the user must provide an initial value for the learning rate greater than 0.01 in order to obtain accurate estimates. If required, the user can try with different combinations of the optimizer and its hyperparameters values to determine which one gets the best results.

When estimating parameters from distributions not included in Table **4-1**, the user must provide the mass/density function of the distribution of interest. To avoid getting errors during the estimation process it is important to make sure that the function has the right arguments which are mentioned in the **estimtf** package section. Also, the user must avoid adding curly brackets when writing the mass/density function.

The optimization process may fail because the parameters take values outside their space. To avoid this when estimating distributional parameters, the user should provide limits to the parameters to impose restrictions during the estimation process. On the other hand, when estimating regression parameters the user must determine if it is required to apply a link function to all or some of the parameters to be estimated. Also, we recommend the user to try with different optimizers, with different values for the learning rate and to check if there is any problem with the input data. Finally, when working with a distribution not included in Table **4-1**, if the process keeps failing even after following all the recommendations, it is possible that some invalid computation is being performed when evaluating the log-likelihood function. Therefore the user must review the function and try to make changes to avoid these numerical problems.

7.3. Future work

From the application section we conclude that the execution time of the `mle_tf` function and the `mlereg_tf` function is greater than the execution time of functions such as `optim, nlminb, nlm` and `glm`. We acknowledge that the execution time is a very important factor when choosing a function to carry out a specific task. Therefore, we consider that it is crucial to work on improving the efficiency of the functions included in the **estimtf** package in order to provide R users accurate results in less time. One of the ideas we have to improve not only the efficiency of the package, but also the quality of its estimates is by combining TensorFlow's optimization algorithms that use the descent gradient method with other optimization methods such as Newton-Raphson.

The simulation study presented earlier was designed to determine if the sample size, the selected optimizer and the initial value for the learning rate had an effect on the maximum likelihood estimates obtained with the `mle_tf` function and the `mlereg_tf` function. As there are other variables involve in the estimation process such as the maximum number of iterations and the values for the optimizer hyperparameters other than the learning rate, it would be interesting to determine if these variables have an effect on the estimates obtained with both functions. Also, as in the simulation study we only included the Poisson distribution and the EEG distribution, we will continue to study different distributions to determine the values of the hyperparameters from which good estimates are obtained and to replicate these conditions in the estimation of parameters of similar distributions.

For now, the **estimtf** package has eight available distributions for which the user must only provide the name of the distribution. In order to simplify the implementation of the the `mle_tf` function and the `mlereg_tf` function, we intend to continue increasing the list of available distributions so that the user does not have to worry about providing the mass/-density function of the distribution of interest. Also, we consider important to improve the provided summary of the estimates by adding more information about the estimation process and metrics such as the AIC for model selection.

Another really important aspect of a package is its documentation. We have been and we will continue working in improving the documentation related to package dependencies, package functions, its inputs, outputs and general functionalities. In order to improve the user experience when using the functions of the **estimtf** package, it is essential to provide information about the errors that may occur during the estimation process. For this reason, we plan to include in the package manual a list of errors for each function of the **estimtf** package, the cause of these errors and recommendations to solve them.

In the simulation study, only the `mle_tf` and the `mlereg_tf` functions of the **estimtf** package were used to estimate the parameters of the Poisson and EEG distributions. However, we consider it important to compare the results of these functions in terms of error and precision of the estimates and in terms of execution time, with the results of other similar functions available in R. For this reason, we plan to extend this simulation study by including other functions, which will allow us to determine more clearly if there is any significant difference in the performance of these functions and of the **estimtf** package functions.

Finally, due to the fact that in optimization problems such as maximum likelihood estimation, situations may arise in which the optimization algorithm fails due to for example poor specification of the initial point, it is important to perform a stability and sensitivity analysis of the optimization algorithm implemented in the `mle_tf` and `mlereg_tf` in different extreme situations and determine how its performance is affected in these conditions.

A. Appendix: estimtf package manual

In this appendix, we present the manual of the **estimtf** package. This manual contains detailed information of the arguments and outputs of each function included in the package as well as some examples of the use of these functions which can be run in R.

Package ‘estimtf’

October 2, 2021

Type Package

Title Estimation of Distributional and Regression Parameters using TensorFlow

Version 0.1.0

Author Sara Garcés [aut, cre],
Freddy Hernández [ctb]

Maintainer Sara Garcés <`sgarcesc@unal.edu.co`>

Description
Provides functions to find the Maximum Likelihood Estimates of parameters from probability distributions and linear regression models using the TensorFlow optimizers. The optimization process included in the main functions of this package use the 'tensorflow' package <https://CRAN.R-project.org/package=tensorflow> and the 'tfprobability' package <https://CRAN.R-project.org/package=tfprobability>.

License GPL-3

Encoding UTF-8

LazyData true

URL `https://github.com/SaraGarcesCespedes/estimtf`

BugReports `https://github.com/SaraGarcesCespedes/estimtf/issues`

RoxygenNote 7.1.1

Depends R (>= 3.5.0)

Imports tensorflow,
dplyr,
reticulate,
stringr,
tfprobability,
keras,
purrr,
stats

Suggests knitr,
rmarkdown,
markdown

VignetteBuilder knitr

R topics documented:

`mlereg_tf` *mlereg_tf function*

Description

Function to find the Maximum Likelihood Estimates of regression parameters using TensorFlow.

Usage

```
mlereg_tf(
  ydist = y ~ Normal,
  formulas,
  data,
  available_distribution = TRUE,
  fixparam = NULL,
  initparam = NULL,
  link_function = NULL,
  optimizer = "AdamOptimizer",
  hyperparameters = NULL,
  maxiter = 10000,
  tolerance = .Machine$double.eps
)
```

Arguments

`ydist`	an object of class "formula" that specifies the distribution of the response variable. The default value is `y ~ Normal`. The available distributions are: `Normal`, `Poisson`, `Binomial`, `Weibull`, `Exponential`, `LogNormal`, `Beta` and `Gamma`. If you want to estimate parameters from a distribution different to the ones mentioned above, you must provide the name of an object of class function that contains its probability mass/density function. This `R` function must not contain curly brackets other than those that enclose the function.
`formulas`	a list containing objects of class "formula". Each element of the list represents the linear predictor for each of the parameters of the regression model. The linear predictor is specified with the name of the parameter and it must contain an `~`. The terms on the right side must be separated by `+`.
`data`	a data frame containing the response variable and the covariates.
`available_distribution`	logical. If `TRUE`, the distribution of the response variable is one of the following distributions: `Normal`, `Poisson`, `Binomial`, `Weibull`, `Exponential`, `LogNormal`, `Beta` and `Gamma`.

`fixparam`	a list containing the fixed parameters of the model only if they exist. The parameters values and names must be specified in the list.
`initparam`	a list with the initial values of the regression coefficients to be estimated. The list must contain the regression coefficients values and names. If you want to use the same initial values for all regression coefficients associated with a specific parameter, you can specify the name of the parameter and the value. If NULL the default initial value is zero.
`link_function`	a list with names of parameters to be linked and the corresponding link function name. The available link functions are: `log`, `logit`, `squared` and `identity`.
`optimizer`	a character indicating the name of the TensorFlow optimizer to be used in the optimization process. The default value is `'AdamOptimizer'`. The available optimizers are: `"AdadeltaOptimizer"`, `"AdagradDAOptimizer"`, `"AdagradOptimizer"`, `"AdamOptimizer"`, `"GradientDescentOptimizer"`, `"MomentumOptimizer"` and `"RMSPropOptimizer"`.
`hyperparameters`	a list with the hyperparameters values of the selected TensorFlow optimizer. If the hyperparameters are not specified, their default values will be used in the oprimization process. For more details of the hyperparameters go to this URL: https://www.tensorflow.org/api_docs/python/tf/compat/v1/train
`maxiter`	a positive integer indicating the maximum number of iterations for the optimization algorithm. The default value is `10000`.
`tolerance`	a small positive number. When the difference between the loss value or the parameters values from one iteration to another is lower than this value, the optimization process stops. The default value is `.Machine$double.eps`.

Details

`mlereg_tf` computes the log-likelihood function based on the distribution specified in `ydist` and linear predictors specified in `formulas`. Then, it finds the values of the regression coefficients that maximizes this function using the TensorFlow opimizer specified in `optimizer`.

The `R` function that contains the probability mass/density function must not contain curly brackets. The only curly brackets that the function can contain are those that enclose the function, that is, those that define the beginning and end of the `R` function.

Value

This function returns the estimates, standard errors, Z-score and p-values of significance tests of the regression model coefficients as well as some information of the optimization process like the number of iterations needed for convergence.

Note

The `summary,print,plot_loss` functions can be used with a `mlereg_tf` object.

Author(s)

Sara Garcés Céspedes <`sgarcesc@unal.edu.co`>

Examples

```
#--------------------------------------------------------------------------------
# Estimation of coefficients of a Poisson regression model

# Data frame with response variable and covariates
counts <- c(18,17,15,20,10,20,25,13,12)
outcome <- gl(3,1,9)
treatment <- gl(3,3)
data <- data.frame(treatment, outcome, counts)

# Use the mlereg_tf function
estimation_1 <- mlereg_tf(ydist =  counts ~ Poisson,
                          formulas = list(lambda = ~ outcome + treatment),
                          data = data,
                          initparam = list(lambda = 1.0),
                          optimizer = "AdamOptimizer",
                          link_function = list(lambda = "log"),
                          hyperparameters = list(learning_rate = 0.1))

# Get the summary of the estimates
summary(estimation_1)

#--------------------------------------------------------------------------------
# Estimation of coefficients of a linear regression model with one fixed parameter

# Data frame with response variable and covariates
x <- runif(n = 1000, -3, 3)
y <- rnorm(n = 1000, mean = 5 - 2 * x, sd = 3)
data <- data.frame(y = y, x = x)

# Use the mlereg_tf function
estimation_2 <- mlereg_tf(ydist = y ~ Normal,
                          formulas = list(mean = ~ x),
                          data = data,
                          fixparam = list(sd = 3),
                          initparam = list(mean = list(Intercept = 1.0, x = 0.0)),
                          optimizer = "AdamOptimizer",
                          hyperparameters = list(learning_rate = 0.1))

# Get the summary of the estimates
summary(estimation_2)

#--------------------------------------------------------------------------------
# Estimation of parameter lambda of the Instantaneous Failures distribution

# Create an R function that contains the probability density function
pdf <- function(y, lambda) { (1 / ((lambda ^ 2) * (lambda - 1))) *
                             (lambda^2 + y - 2*lambda) * exp(-y/lambda) }

# Data frame with response variable
y <-  c(3.4, 0.0, 0.0, 15.8, 232.8, 8.8, 123.2, 47, 154, 103.2, 89.8,  12.2)
data <- data.frame(y)

# Use the mlereg_tf function
estimation_3 <- mlereg_tf(ydist = y ~ pdf,
                          formulas = list(lambda = ~1),
                          data = data,
                          initparam = list(lambda = rnorm(1, 5, 1)),
                          available_distribution = FALSE,
                          optimizer = "AdamOptimizer",
                          hyperparameters = list(learning_rate = 0.1),
                          maxiter = 10000)

# Get the summary of the estimates
summary(estimation_3)
```

`mle_tf` *mle_tf function*

Description

Function to find the Maximum Likelihood Estimates of distributional parameters using TensorFlow.

Usage

```
mle_tf(
  x,
  xdist = "Normal",
  fixparam = NULL,
  initparam,
  bounds = NULL,
  optimizer = "AdamOptimizer",
  hyperparameters = NULL,
  maxiter = 10000,
  tolerance = .Machine$double.eps
)
```

Arguments

`x`	a vector containing the data to be fitted.
`xdist`	a character indicating the name of the distribution of interest. The default value is `'Normal'`. The available distributions are: `Normal`, `Poisson`, `Binomial`, `Weibull`, `Exponential`, `LogNormal`, `Beta` and `Gamma`. If you want to estimate parameters from a distribution different to the ones mentioned above, you must provide the name of an object of class function that contains its probability mass/density function. This `R` function must not contain curly brackets other than those that enclose the function.
`fixparam`	a list containing the fixed parameters of the distribution of interest only if they exist. The parameters values and names must be specified in the list.
`initparam`	a list with initial values of the parameters to be estimated. The list must contain the parameters values and names.
`bounds`	a list with lower and upper bounds for each parameter to be estimated. The list must contain the parameters names and vectors with the bounds. The default value is NULL.

`optimizer`	a character indicating the name of the TensorFlow optimizer to be used in the estimation process The default value is `'AdamOptimizer'`. The available optimizers are: `"AdadeltaOptimizer"`, `"AdagradDAOptimizer"`, `"AdagradOptimizer"`, `"AdamOptimizer"`, `"GradientDescentOptimizer"`, `"MomentumOptimizer"` and `"RMSPropOptimizer"`.
`hyperparameters`	a list with the hyperparameters values of the selected TensorFlow optimizer. If the hyperparameters are not specified, their default values will be used in the oprimization process. For more details of the hyperparameters go to this URL: https://www.tensorflow.org/api_docs/python/tf/compat/v1/train
`maxiter`	a positive integer indicating the maximum number of iterations for the optimization algorithm. The default value is `10000`.
`tolerance`	a small positive number. When the difference between the loss value or the parameters values from one iteration to another is lower than this value, the optimization process stops. The default value is `.Machine$double.eps`.

Details

`mle_tf` computes the log-likelihood function of the distribution specified in `xdist` and finds the values of the parameters that maximizes this function using the TensorFlow optimizer specified in `optimizer`.

The `R` function that contains the probability mass/density function must not contain curly brackets. The only curly brackets that the function can contain are those that enclose the function, that is, those that define the beginning and end of the `R` function.

Value

This function returns the estimates, standard errors, Z-score and p-values of significance tests of the parameters from the distribution of interest as well as some information of the optimization process like the number of iterations needed for convergence.

Note

The `summary,print,plot_loss` functions can be used with a `mle_tf` object.

Author(s)

Sara Garcés Céspedes <`sgarcesc@unal.edu.co`>

Examples

```
#--------------------------------------------------------------------------------
# Estimation of parameters mean and sd of the normal distribution

# Vector with the data to be fitted
x <- rnorm(n = 1000, mean = 10, sd = 3)

# Use the mle_tf function
estimation_1 <- mle_tf(x,
                        xdist = "Normal",
                        optimizer = "AdamOptimizer",
                        initparam = list(mean = 1.0, sd = 1.0),
                        hyperparameters = list(learning_rate = 0.1))
```

```
# Get the summary of the estimates
summary(estimation_1)

#----------------------------------------------------------------------------
# Estimation of parameter lambda of the Instantaneous Failures distribution

# Create an R function that contains the probability density function
pdf <- function(X, lambda) { (1 / ((lambda ^ 2) * (lambda - 1))) *
                             (lambda^2 + X - 2*lambda) * exp(-X/lambda) }

# Vector with the data to be fitted
x <-  c(3.4, 0.0, 0.0, 15.8, 232.8, 8.8, 123.2, 47, 154, 103.2, 89.8,  12.2)

# Use the mle_tf function
estimation_2 <- mle_tf(x = x,
                       xdist = pdf,
                       initparam = list(lambda = rnorm(1, 5, 1)),
                       bounds = list(lambda = c(2, Inf)),
                       optimizer = "AdamOptimizer",
                       hyperparameters = list(learning_rate = 0.1),
                       maxiter = 10000)

# Get the summary of the estimates
summary(estimation_2)
```

`plot_loss` *plot_loss function*

Description

Function to display a graph that contains the loss value computed in each iteration of the optimization process performed using the `mle_tf` function or using the `mlereg_tf` function.

Usage

```
plot_loss(object, ...)
```

Arguments

`object`	an object of class `MLEtf` for which the construction of a graph with the loss values is desired.
`...`	additional arguments affecting the constructed graph.

Details

`plot_loss.MLEtf` function displays a graph of the loss value, which correspond to the negative log-likelihood computed in each iteration of the optimization process.

Author(s)

Sara Garcés Céspedes <`sgarcesc@unal.edu.co`>

Examples

```
#--------------------------------------------------------------
# Estimation of both normal distrubution parameters

# Generate a sample from the normal distribution
x <- rnorm(n = 1000, mean = 10, sd = 3)

# Use the plot_loss function
plot_loss(mle_tf(x,
                 xdist = "Normal",
                 optimizer = "AdamOptimizer",
                 initparam = list(mean = 1.0, sd = 1.0),
                 hyperparameters = list(learning_rate = 0.1)))
```

`print.MLEtf` *print.MLEtf function*

Description

Function to display the estimates of parameters from probability distributions using the `mle_tf` function or parameters from regression models using the `mlereg_tf` function.

Usage

```
## S3 method for class 'MLEtf'
print(x, ...)
```

Arguments

`x`	an object of class `MLEtf` for which a visualization of the estimates is desired.
`...`	additional arguments affecting the displayed estimates.

Details

`print.MLEtf` function displays the estimates of parameters from probability distributions and regression models.

Author(s)

Sara Garcés Céspedes <`sgarcesc@unal.edu.co`>

Examples

```
#--------------------------------------------------------------
# Estimation of both normal distrubution parameters

# Generate a sample from the normal distribution
x <- rnorm(n = 1000, mean = 10, sd = 3)

# Use the print function
print(mle_tf(x,
             xdist = "Normal",
             initparam = list(mean = 1.0, sd = 1.0),
             optimizer = "AdamOptimizer",
             hyperparameters = list(learning_rate = 0.1)))
```

`summary.MLEtf` *summary.MLEtf function*

Description

Function to produce result summaries of the estimates of parameters from probability distributions using the `mle_tf` function or parameters from regression models using the `mlereg_tf` function.

Usage

```
## S3 method for class 'MLEtf'
summary(object, ...)
```

Arguments

`object`	an object of class `MLEtf` for which a summary is desired.
`...`	additional arguments affecting the summary produced.

Details

`summary.MLEtf` function displays estimates and standard errors of parameters from statistical distributions and regression models. Also, this function computes and displays the Z-score and p-values of significance tests for these parameters.

Author(s)

Sara Garcés Céspedes <`sgarcesc@unal.edu.co`>

Examples

```
#--------------------------------------------------------------
# Estimation of both normal distrubution parameters

# Generate a sample from the normal distribution
x <- rnorm(n = 1000, mean = 10, sd = 3)

# Use the summary function
summary(mle_tf(x,
               xdist = "Normal",
               optimizer = "AdamOptimizer",
               initparam = list(mean = 1.0, sd = 1.0),
               hyperparameters = list(learning_rate = 0.1)))
```

Index

B. Appendix: R code used in the simulation study

In this appendix, we present the R code used in the simulation study for all the scenarios.

B.1. Scenario 1

```r
# Function to generate a sample of size n of the Poisson distribution
gen_data <- function(lambda, n) {

  datos <- rpois(n, lambda = lambda)

  return(datos)
}

# Function to replicate gen_data function
auxiliar <- function(n) {
        lambda <- 2.91
        datos <- gen_data(lambda, n)
        return(datos)
}
```

Code B.1: Functions to generate data from the Poisson distribution with parameter λ.

```r
# Function to perform one simulation
one_simul <- function(n, learning_rate, optimizer) {
    # Generate the random samples
    datos <- auxiliar(n)

    # Use the mle_tf function to estimate parameters of the Poisson
   distribution
    estimation_2 <- try(mle_tf(x = datos,
                              xdist = "Poisson",
                              initparam = list(lambda = 0.5),
                              optimizer = optimizer,
                              hyperparameters =list(
                                learning_rate=learning_rate),
                                maxiter = 10000),
                         silent = TRUE)

    # Get the estimates
    if (class(estimation_2)[1] == "try-error") {
            estimates <- rep(NA, 2)
    } else {
            estimates <- c(estimation_2$outputs$estimates)
    }
    results <- c(estimates)
    result <- cbind(t(results), n, learning_rate, optimizer)

    # Save the results
    write(x = t(result), file = 'simul_sincov_poisson.txt', ncol = 5,
          append=TRUE)
}
```

Code B.2: Function to perform one simulation for each of the 54 cases in scenario 1.

```
# Function to replicate the one_simul function
mult_simul <- function(x) {
    n <- x[1]
    n <- as.numeric(n)
    learning_rate <- x[2]
    learning_rate <- as.numeric(learning_rate)
    optimizer <- x[3]
    # replicate nrep times the one_simul function
    res <- replicate(n=nrep,
                     expr=one_simul(n = n,
                                    learning_rate = learning_rate,
                                    optimizer = optimizer))
}

# Specify the parameters for the simulation
n <- c(20, 50, 100, 200, 500, 1000)
learning_rate <- c(0.1, 0.01, 0.001)
optimizer <- c("AdamOptimizer", "RMSPropOptimizer", "AdagradOptimizer")
nrep <- 1000
values <- expand.grid(n = n,
                      learning_rate = learning_rate,
                      optimizer = optimizer)
values$optimizer <- as.character(values$optimizer)

# Start simulation
apply(values, 1, mult_simul)
```

Code B.3: Function to replicate 1000 times the one_simul function for scenario 1.

B.2. Scenario 2

```r
# Function to generate a sample of size n of the Poisson regression model
gen_data <- function(lambda, x1, n) {

  datos <- rpois(n, lambda = lambda)
  datos_final <- c(datos, x1)

  return(datos_final)
}
# Function to replicate gen_data function
auxiliar <- function(n) {
        beta0 <- -0.42
        beta1 <- 0.58
        x1 <- rnorm(n, 2.5, 0.6)
        lambda <- exp(beta0 + beta1 * x1)
        datos <- gen_data(lambda, x1, n)
        return(datos)
}
```

Code B.4: Functions to generate data from the Poisson regression model with coefficient β_0 and β_1.

```r
# Function to perform one simulation
one_simul <- function(n, learning_rate, optimizer) {
    # Generate the random samples
    datos <- replicate(n=n_repeticiones, expr=auxiliar(n= 1))
    datos <- as.data.frame(t(datos))
    colnames(datos) <- c("y", "x1")

    # Use the mlereg_tf function to estimate parameters of the Poisson
   regression model
    formulas <- list(lambda = ~x1)
    estimation_2 <- try(mlereg_tf(ydist = y ~ Poisson,
                                  formulas = formulas,
                                  data = datos,
                                  available_distribution = TRUE,
                                  fixparam = NULL,
                                  initparam=list(lambda= 0.5),
                                  link_function=list(
                                  lambda="log"),
                                  optimizer = optimizer,
                                  hyperparameters = list(
                                  learning_rate = learning_rate),
                                  maxiter = 10000),
                        silent = TRUE)
    # Get the estimates
    if (class(estimation_2)[1] == "try-error") {
            estimates <- rep(NA, 2)
    } else {
            estimates <- c(estimation_2$outputs$estimates)
    }
    results <- c(estimates)
    result <- cbind(t(results), n, learning_rate, optimizer)

    # Save the results
    write(x = t(result), file = 'simul_cov_poisson.txt', ncol = 6,
          append=TRUE)
}
```

Code B.5: Function to perform one simulation for each of the 54 cases in scenario 2.

```
# Function to replicate the one_simul function
mult_simul <- function(x) {
    n <- x[1]
    n <- as.numeric(n)
    learning_rate <- x[2]
    learning_rate <- as.numeric(learning_rate)
    optimizer <- x[3]
    # replicate nrep times the one_simul function
    res <- replicate(n=nrep,
                     expr=one_simul(n = n,
                                    learning_rate = learning_rate,
                                    optimizer = optimizer))
}

# Specify the parameters for the simulation
n <- c(20, 50, 100, 200, 500, 1000)
learning_rate <- c(0.1, 0.01, 0.001)
optimizer <- c("AdamOptimizer", "RMSPropOptimizer", "AdagradOptimizer")
nrep <- 1000
values <- expand.grid(n = n,
                      learning_rate = learning_rate,
                      optimizer = optimizer)
values$optimizer <- as.character(values$optimizer)

# Start simulation
apply(values, 1, mult_simul)
```

Code B.6: Function to replicate 1000 times the `one_simul` function for scenario 2.

B.3. Scenario 3

```
# Function to generate a sample of size n of the EEG distribution
gen_data <- function(gamma, beta, n) {
  gamma <- gamma
  beta <- beta

  # Define the probability density function
  f <- function(x) {(beta * gamma * exp(-beta * x)) / (1 -
     (1 - gamma) * exp(-beta * x))^2}
  # Define the cumulative distribution function
  Fa <- function(x) {integrate(f, 0, x)$value}
  Fa <- Vectorize(Fa)
  # Inverse of the cumulative distribution function
  F.inv <- function(y){uniroot(function(x){Fa(x) - y}, interval=c(0, 1),
   extendInt = "upX")$root}
  F.inv <- Vectorize(F.inv)
  # Generate a random sample from U(0,1) and evaluate in F.inv
  Y <- runif(n, 0, 1)
  datos <- F.inv(Y)

  return(datos)
}

# Function to replicate gen_data function
auxiliar <- function(n) {
    gamma <- 2
    beta <- 4
    datos <- gen_data(gamma, beta, n)
    return(datos)
}
```

Code B.7: Functions to generate data from the EEG distribution with parameters β and γ.

```
# Function to perform one simulation
one_simul <- function(n, learning_rate, optimizer) {
    # Generate the random samples
    datos <- auxiliar(n)
    # Define the probability density function of the EEG distirbution
    pdf <- function(X, beta, gamma) {
    (beta*gamma*exp(-beta * x))/(1 -(1 - gamma)*exp(-beta * x))^2
    }
    # Use the mle_tf function to estimate parameters of the EEG
   distribution
    estimation_2 <- try(mle_tf(x = datos,
                               xdist = pdf,
                               initparam = list(beta = 0.5,
                               gamma = 0.5),
                               optimizer = optimizer,
                               hyperparameters = list(
                               learning_rate = learning_rate),
                               maxiter = 10000),
                        silent = TRUE)

    # Get the estimates
    if (class(estimation_2)[1] == "try-error") {
            estimates <- rep(NA, 2)
    } else {
            estimates <- c(estimation_2$outputs$estimates)
    }
    results <- c(estimates)
    result <- cbind(t(results), n, learning_rate, optimizer)

    # Save the results
    write(x = t(result), file = 'simul_sincov.txt', ncol = 6, append=TRUE)
}
```

Code B.8: Function to perform one simulation for each of the 54 cases in scenario 3.

```
# Function to replicate the one_simul function
mult_simul <- function(x) {
    n <- x[1]
    n <- as.numeric(n)
    learning_rate <- x[2]
    learning_rate <- as.numeric(learning_rate)
    optimizer <- x[3]
    # replicate nrep times the one_simul function
    res <- replicate(n=nrep,
                     expr=one_simul(n = n,
                                    learning_rate = learning_rate,
                                    optimizer = optimizer))
}
# Specify the parameters for the simulation
n <- c(20, 50, 100, 200, 500, 1000)
learning_rate <- c(0.1, 0.01, 0.001)
optimizer <- c("AdamOptimizer", "RMSPropOptimizer", "AdagradOptimizer")
nrep <- 1000
values <- expand.grid(n = n, learning_rate = learning_rate,
                      optimizer = optimizer)
values$optimizer <- as.character(values$optimizer)

# Start simulation
apply(values, 1, mult_simul)
```

Code B.9: Function to replicate 1000 times the `one_simul` function for scenario 3.

B.4. Scenario 4

```
# Function to generate a sample of size n of the EEG distribution
gen_data <- function(gamma, beta, x1, x2, n) {
  gamma <- gamma
  beta <- beta

  # Define the probability density function
  f <- function(x) {(beta * gamma * exp(-beta * x)) / (1 -
     (1 - gamma) * exp(-beta * x))^2}
  # Define the cumulative distribution function
  Fa <- function(x) {integrate(f, 0, x)$value}
  Fa <- Vectorize(Fa)
  # Inverse of the cumulative density function
  F.inv <- function(y){uniroot(function(x){Fa(x) - y}, interval=c(0, 1),
   extendInt = "upX")$root}
  F.inv <- Vectorize(F.inv)
  # Generate a random sample from U(0,1) and evaluate in F.inv
  Y <- runif(n, 0, 1)
  datos <- F.inv(Y)
  datos_final <- c(datos, x1, x2)
  return(datos_final)
}

# Function to replicate gen_data function
auxiliar <- function(n) {
        beta0 <- 0.5
        beta1 <- 1.5
        beta2 <- 2
        beta3 <- -3
        x1 <- runif(n, 0, 1)
        x2 <- runif(n, 0, 1)
        beta <- exp(beta0 + beta1 * x1)
        gamma <- exp(beta2 + beta3 * x2)
        datos <- gen_data(gamma, beta, x1, x2, n)
        return(datos)}
```

Code B.10: Functions to generate data from the EEG distribution with parameters β and γ which are in terms of linear predictors.

```
# Function to perform one simulation
one_simul <- function(n_repeticiones, learning_rate, optimizer) {
    # Generate the random samples
    datos <- replicate(n = n_repeticiones,
                       expr = auxiliar(n = 1))
    datos <- as.data.frame(t(datos))
    colnames(datos) <- c("y", "x1", "x2")

    # Use the mlereg_tf function to estimate parameters of the EEG
   distribution
    formulas <- list(beta = ~x1, gamma = ~x2)
    estimation_2 <- try(mlereg_tf(ydist = y ~ pdf,
                                  formulas = formulas,
                                  data = datos,
                                  available_distribution = FALSE,
                                  fixparam = NULL,
                                  initparam = list(beta=0.5, gamma = 0.5),
                                  link_function = list(
                                  beta = "log", gamma = "log"),
                                  optimizer = optimizer,
                                  hyperparameters = list(learning_rate =
                                  learning_rate),
                                  maxiter = 10000),
                        silent = TRUE)

    # Get the estimates
    if (class(estimation_2)[1] == "try-error") {
            estimates <- rep(NA, 2)
    } else {
            estimates <- c(estimation_2$outputs$estimates)
    }
    results <- c(estimates)
    result <- c(results, n_repeticiones, learning_rate, optimizer)

    # Save the results
    write(x = t(result), file = 'simul_cov.txt', ncol = 8, append = TRUE)
}
```

Code B.11: Function to perform one simulation for each of the 54 cases in scenario 4.

```
# Function to replicate the one_simul function
pdf <- function(y, beta, gamma) {(beta * gamma * exp(-beta * y)) / (1 - (1
    - gamma) * exp(-beta * y))^2}

mult_simul <- function(x) {
        n <- x[1]
        n <- as.numeric(n)
        learning_rate <- x[2]
        learning_rate <- as.numeric(learning_rate)
        optimizer <- x[3]
        optimizer <- as.character(optimizer)
        # replicate nrep times the one_simul function
        res <- replicate(n = nrep,
                         expr = one_simul(n = n,
                         learning_rate = learning_rate,
                         optimizer = optimizer))
}

# Specify the parameters for the simulation
n <- c(20, 50, 100, 200, 500, 1000)
learning_rate <- c(0.1, 0.01, 0.001)
optimizer <- c("AdamOptimizer", "RMSPropOptimizer", "AdagradOptimizer")
nrep <- 1000
values <- expand.grid(n = n,
                      learning_rate = learning_rate,
                      optimizer = optimizer)
values$optimizer <- as.character(values$optimizer)

# Start simulation
apply(values, 1, mult_simul)
```

Code B.12: Function to replicate 1000 times the `one_simul` function for scenario 4.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, M., J. Isard, ... Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation.*

Adamidis, K., Dimitrakopoulou, T., & Loukas, S. (2005). On an extension of the exponential-geometric distribution. *Statistics Probability Letters*, *73*, 259-269.

Agresti, A. (2015). *Foundations of linear and generalized linear models.* Wiley.

Allaire, J., & Tang, Y. (2021). tensorflow: R interface to "tensorflow" [Computer software manual]. Retrieved from `https://github.com/rstudio/tensorflow` (R package version 2.2.0.9000)

Bakouch, H., Dey, S., Ramos, P., & Louzada, F. (2017). Binomial-exponential 2 Distribution: Different Estimation Methods with Weather Applications. *TEMA (São Carlos)*, *18*, 233 - 251.

Bebbington, M., Lai, C.-D., & Zitikis, R. (2007). A flexible weibull extension. *Reliability Engineering System Safety*, *92*(6), 719-726.

Bengio, Y. (2012). *Practical recommendations for gradient-based training of deep architectures.*

Bolker, B., & R Development Core Team. (2020). bbmle: Tools for general maximum likelihood estimation [Computer software manual]. Retrieved from `https://CRAN.R-project.org/package=bbmle` (R package version 1.0.23.1)

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. *Proc. of COMPSTAT*.

Boyd, S., & Vandenberghe, L. (2004). *Convex optimization.* Cambridge University Press.

Byrd, R., Lu, P., Nocedal, J., & Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal of Scientific Computing*, *16*, 1190–1208.

Bélisle, C. J. (1992). Convergence theorems for a class of simulated annealing algorithms on Rd. *Journal of Applied Probability*, 885–895.

Commenges, D., Jacqmin-Gadda, H., Proust-Lima, C., & Guedj, J. (2006). A newton-like algorithm for likelihood maximization: The robust-variance scoring algorithm. *Arxiv math/0610402*.

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, *39*(1), 1–38.

Devore, J. (2016). *Probability and statistics for engineering and the sciences.* Cengage

Learning. Retrieved from https://books.google.com.co/books?id=UouECwAAQBAJ

Dey, S., Raheem, E., & Mukherjee, S. (2017). Statistical properties and different methods of estimation of transmuted rayleigh distribution. *Revista Colombiana de Estadística*, *40*, 165 - 203. Retrieved from http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0120-17512017000100008&nrm=iso

Do, Q., Son, T., & Chaudri, J. (2017). Classification of asthma severity and medication using tensorflow and multilevel databases. *Procedia Computer Science*, *113*, 344-351.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*, 2121-2159.

Fisher, R. A. (1922). On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London, A*, *222*, 309–368.

Fox, P. A., Hall, A. P., & Schryer, N. L. (1978). The port mathematical subroutine library. *ACM Trans. Math. Softw.*, *4*(2), 104–126.

Galeone, P. (2019). *Hands-on neural networks with tensorflow 2.0: understand tensorflow, from static graph to eager execution, and design neural networks* (1st ed.). Packt Publishing.

Garcés, S., & Hernández, F. (2021). estimtf: Estimation of distributional and regression parameters using tensorflow [Computer software manual]. Retrieved from https://github.com/SaraGarcesCespedes/estimtf (R package version 0.1.0)

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning.* MIT Press. (http://www.deeplearningbook.org)

Henningsen, A., & Toomet, O. (2011). maxlik: A package for maximum likelihood estimation in R. *Computational Statistics*, *26*(3), 443-458.

Hernandez, F., Usuga, O., Patino, C., Mosquera, J., & Urrea, A. (2021). Reldists: Estimation for some reliability distributions within gamlss framework [Computer software manual]. Retrieved from https://ousuga.github.io/RelDists/ (R package version 1.0.0)

Hernández, F., & Usuga, O. (2019). Manual de R [Computer software manual]. Retrieved from https://fhernanb.github.io/Manual-de-R/

Ihaka, R., & Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, *5*(3), 299–314.

Karlis, D., & Xekalaki, E. (2003). Choosing initial values for the em algorithm for finite mixtures. In *Comput. stat. data anal.*

Keydana, S. (2020). tfprobability: Interface to "tensorflow probability" [Computer software manual]. Retrieved from https://CRAN.R-project.org/package=tfprobability (R package version 0.11.0.0)

Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. *International Conference on Learning Representations.*

Kissell, R., & Poserina, J. (2017). Chapter 4 - advanced math and statistics. In R. Kissell & J. Poserina (Eds.), *Optimal sports math, statistics, and fantasy* (p. 103-135). Academic Press. Retrieved from https://www.sciencedirect.com/science/article/

pii/B9780128051634000049

Legendre, A. M. A. M. (1805). *Nouvelles méthodes pour la détermination des orbites des comètes [microform] / par a.m. legendre.* Paris: F. Didot.

Ling, M. (2018). A comparison of estimation methods for generalized gamma distribution with one-shot device testing data..

Little, T. (2014). *The oxford handbook of quantitative methods* (No. v. 1). Oxford University Press.

Louzada, F., Ramos, P. L., & Perdoná, G. (2016). Different estimation procedures for the parameters of the extended exponential geometric distribution for medical data. *Computational and Mathematical Methods in Medicine*, *2016*.

Mai Anh, T., Bastin, F., & Frejinger, E. (2014). On optimization algorithms for maximum likelihood estimation.

Merovci, F. (2013). Transmuted rayleigh distribution. *Austrian Journal of Statistics*, *42*(1), 21-31. Retrieved from https://www.ajs.or.at/index.php/ajs/article/view/vol42%2C%20no1-2

Millar, R. (2011). *Maximum likelihood estimation and inference: With examples in R, SAS and ADMB.* Wiley.

Mosquera, J., & Hernandez, F. (2019). Estimationtools: Maximum likelihood estimation for probability functions from data sets [Computer software manual]. Retrieved from https://CRAN.R-project.org/package=EstimationTools (R package version 1.2.1)

Mullen, K., Ardia, D., Gil, D., Windover, D., & Cline, J. (2011). DEoptim: An R package for global optimization by differential evolution. *Journal of Statistical Software*, *40*(6), 1–26. Retrieved from http://www.jstatsoft.org/v40/i06/

Muralidharan, K., & Khabia, A. (2014). Some statistical inferences on inlier(s) models. *International Journal of System Assurance Engineering and Management*, *8*.

Nash, J. C. (2014). Nonlinear parameter optimization using R tools..

Nelder, J., & Mead, R. (1965). A simplex method for function minimization. *Comput. J.*, *7*, 308-313.

Nelder, J. A., & Wedderburn, R. W. M. (1972). Generalized linear models. *Journal of the Royal Statistical Society. Series A (General)*, *135*(3), 370–384.

Nesterov, Y. (2014). *Introductory lectures on convex optimization: A basic course* (1st ed.). Springer Publishing Company, Incorporated.

Nocedal, J., & Wright, S. (2006). *Numerical optimization.* Springer New York. Retrieved from https://books.google.at/books?id=VbHYoSyelFcC

Pawitan, Y. (2013). *In all likelihood: Statistical modelling and inference using likelihood.* OUP Oxford.

Pearson, K. (1936). Method of moments and method of maximum likelihood. *Biometrika*, *28*(1/2), 34–59.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural*

Networks, *12*(1), 145 - 151.

Ramos, P., & Louzada, F. (2019). A distribution for instantaneous failures. *Stats*, *2*, 247-258.

R Core Team. (2021). R: A language and environment for statistical computing [Computer software manual]. Vienna, Austria. Retrieved from `https://www.R-project.org/`

Rigby, R. A., & Stasinopoulos, D. M. (2005). Generalized additive models for location, scale and shape. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, *54*(3), 507–554. Retrieved from `http://www.jstor.org/stable/3592732`

Rizzo, M. (2007). *Statistical computing with R.* Chapman & Hall/CRC.

Ross, S. M. (2006). *Simulation, fourth edition.* USA: Academic Press, Inc.

RStudio. (2020). *Tensorflow for R.* Retrieved 24-06-2020, from `https://tensorflow.rstudio.com/`

Ruder, S. (2016). *An overview of gradient descent optimization algorithms.*

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*, 533-536.

Sawant, A., Bhandari, M., Yadav, R., Yele, R., & Bendale, S. (2018). Brain cancer detection from mri: a machine learning approach (tensorflow). *International Research Journal of Engineering and Technology (IRJET)*, *5*(4).

Schnabel, R. B., Koonatz, J. E., & Weiss, B. E. (1985). A modular system of algorithms for unconstrained minimization. *ACM Trans. Math. Softw.*, *11*(4), 419–440.

Stasinopoulos, D., Rigby, R., Heller, G., Voudouris, V., & De Bastiani, F. (2017). *Flexible regression and smoothing: Using gamlss in R.*

Stigler, S. M. (1981). Gauss and the invention of least squares. *The Annals of Statistics*, *9*(3), 465–474.

Storvik, G. (2011). Numerical optimization of likelihoods : Additional literature for stk 2120..

Sweeting, T. J. (1980). Uniform asymptotic normality of the maximum likelihood estimator. *The Annals of Statistics*, *8*(6), 1375–1381.

TensorFlow. (2020). *Tensorflow core v2.2.0.* Retrieved 11-06-2020, from `https://www.tensorflow.org/`

Variani, E., Bagby, T., McDermott, E., & Bacchiani, M. (2017). End-to-end training of acoustic models for large vocabulary continuous speech recognition with tensorflow. In *Interspeech.*

Wickham, H. (2015). *R packages* (1st ed.). O'Reilly Media, Inc.

Wilks, D. S. (2019). Chapter 4 - parametric probability distributions. In D. S. Wilks (Ed.), *Statistical methods in the atmospheric sciences (fourth edition)* (Fourth Edition ed., p. 77-141). Elsevier.

Yang, X.-S. (2021). Chapter 1 - introduction to algorithms. In X.-S. Yang (Ed.), *Nature-inspired optimization algorithms (second edition)* (Second Edition ed., p. 1-22). Academic Press. Retrieved from `https://www.sciencedirect.com/science/article/`

pii/B9780128219867000081

Zakerzadeh, H., & Dolati, A. (2009). Generalized lindley distribution. *Journal of Mathematical Extension*, *3*, 1-17.

Zeiler, M. (2012). Adadelta: An adaptive learning rate method. , *1212*.