Final project

DOUBLE BACHELOR IN MATHEMATICS AND BUSINESS ADMINISTRATION

Facultat de Matemàtiques i Informàtica
Facultat d'Economia i Empresa

Universitat de Barcelona

Recurrent Neural Networks for Churn Prediction

Author: Montserrat Comas Turró

Directors: **Vitrià Marca, Jordi**
Department of Mathematics and Computer Science
Torra Porràs, Salvador
Department of Econometrics, Statistics and Spanish Economy

Barcelona, June 2018

Abstract

This project is based on a probabilistic Deep learning model called WTTE-RNN that applies recurrent neural networks along with survival analysis in order to model the distribution of time between specific events. The main motivation of the application of survival analysis is its adjustment to recurrent events, unlike the basic hypothesis of this theory which assumes that the existence of one event implies the end of data entry. In order to understand the main parts that constitute the model, an extensive section of this project addresses Deep learning and Survival Analysis. The approach of the model as a business tool for churn prediction is also important, in order to show how the knowledge acquired during the Mathematics degree can serve as a tool in the business strategy direction and so as a link with the Business degree.

2010 Mathematics Subject Classification. 68T05, 68T20, 68W25, 68W40

Acknowledgements

I would like to thank Jordi Vitrià for helping me through difficulties as well as his patience and unconditional support over these last months. I also thank Salvador Torra for his predisposition for tutoring part of this project and finding links with the business field. Finally my thanks to my family and friends, all of whom supported and helped me during these years of university when I needed it most.

Contents

Introduction

Customer churn is one of the main concerns in all business sectors. For many companies, customer retention policies are not focused on a clear target segment, which can make marketing decisions less efficient. For this reason and with the objective of better understanding their own client, the study of customer churn has been a common procedure lately, based on different mechanisms.

This phenomenon has affected sectors such as communications, banking and insurance, retail, distribution and tourism among others. While in each sector the churn effect varies, this is still an issue in all of them. As companies strive to attract new customers, they are aware of the rising troubles to achieve it due the increasing competition. In addition, it has been proven that, in the long term, customer loyalty is more efficient as well as being cheaper than applying resources to look for new clients.

Before performing any quantitative analysis, it is necessary to analyze the different factors that are promoting customer churn in our company, in order to be able to provide different approaches to any proposed final model for this study. It is necessary to study what types of customers we have, what may cause their abandonment, which sales channels are most recurrent, as well as other variables could be directly added to the model. In the meantime, others could provide different associations of samples that may improve the performance of the chosen model.

The proposed model in this project is capable of working with what could be considered as two different types of business depending on the behavior of their clients. First of all, consider the retail sector which, given the current circumstances and the unstoppable growth of the use of big data, is a scenario with great potential. On the one hand, we have customers susceptible to capture many incentives and information from the current media, from which they are also likely to offer a lot of information about their preferences when consuming. This is what information technologies are trying to take advantage of since the data of each of us is increasing its value over

time. Hence, companies must find a way to encourage their clients and exchange information for mutual interests.

All these matters are trying to be managed nowadays and that is why many models have been born for years. Those may differ in different aspects though, in this project, two main distinctions are remarked: how is churned defined and if deterministic or probabilistic models are chosen to predict this behaviour. Customer churn may be interpreted as a binary solution representing if the customer left the company or not. However, another perspective is one of the main characteristics of the model we expose: a churn definition that depends on the time to the next client event. The idea is about predicting it as a measure of "how churned" a customer is. Instead of how other authors approach this time in a deterministic way by using algorithms such as Decision Trees or Random Forest, the proposed model tries to predict a probabilistic distribution that models this time we are trying to predict.

Once the idea of finding a distribution of the time was exposed and taking into account that the data of the business we may have would be composed of customers tracking up to the present, the model we use must manage with *censored data*. This concept refers to when we can't assert if something will happen or not in the future because, somehow, the tracking of information we use to train a model couldn't finish and thus only a bound of time can be fixed. Due to the desired approach of time between events along with the need of dealing with censored data, *Survival Analysis* came into the model as a main chapter. This subfield of statistics was created to provide solutions for medicine problems such as how much lifetime or how much time until a tumor appearance was left for a sick patient and it is adapted to the customer churn problem.

In contrast with one of the main assumptions in Survival Analysis which implies that there is one single event per sequence, in one of the real life scenarios we consider as a business type, which is the main motivation of this project application, customers life is composed of recurrent purchases. Taking this into account, a mechanism that deals with time dependencies and recurrent events had to be used. This is why *Recurrent Neural Networks* are implemented. These architectures belong to *Deep Learning* algorithms and they are becoming increasingly popular in the latest 20 years; this is why this project is dedicated to them to a large extend.

Finally, from the combination of Survival Analysis and Recurrent Neural Networks, Egil Martinsson built up a model called *The Weibull Time To Event - Recurrent Neural Network (WTTE-RNN)* [14]. This project is mainly based in under-

standing the pieces that compose his model as well as its implementation in the case we had real business data. The available online implementations that were used, [21][17], work with a generated dataset and another real one, but this second case refers to a Jet Engines database whose failure tries to be predicted. Two analogies to hypothetical business data are made based on this in order to establish which procedure one should follow in case they would belong to a company database, together with how to interpret the given output prediction.

In order to achieve all these objectives, this project is divided in four chapters. In *Preliminaries*, all the necessary concepts to understand the main pieces in WTTE-RNN are exposed, based on the big blocks *Machine Learning* and *Survival Analysis*. In the second chapter you can find what the model is compound of, regarding hypothesis, objectives and its basic structure. The third chapter, named *Experiments* is based on the implementation of the algorithm using *Python*, divided in two types of businesses: non-contractual and contractual ones; this distinction differs in the type of data we work with and thus how the *loss-function* of the model changes. This chapter is also destined to interpret the results. Finally, a short recap of what this project meant is done in the conclusions.

Chapter 1

Preliminaries

1.1 Machine Learning

This chapter is mainly inspired in the first three weeks of [18]. Machine learning is a field of computer science used for training computers how to learn. With some input data, the aim of each algorithm is to progressively improve its own performance, which is determined by several parameters without being specifically programmed. A more modern definition of what a machine learning algorithm does is given by Tom M. Mitchell:

Definition 1.1. *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T (measured by P) improves with E.*

Example 1.2. *An algorithm that learns how to play checkers. In this case, the variables T, P, E are the task of playing checkers, the probability that the program wins the next game and the experience of playing many games, respectively.*

Machine learning problems can be assigned to Supervised or Unsupervised learning, depending on if we already know the correct output or not, respectively. Unsupervised learning approaches problems without any idea (or little) of how results should look like and thus, there is no feedback based on the predicted results.

1.1.1 Model representation

Observation 1.3. *Sets' dimensions (parameters, input data, output data) vary depending on the algorithm we refer to.*

Let $D \in (X, Y)$ where X denotes the space of input values and Y denotes the space of output values. Let $x^{(i)}, y^{(i)}$ be the input (features) and output (target) variables

that we want to predict, respectively. For $i = 1, ..., m$, the list of pairs $(x^{(i)}, y^{(i)})$ are the training examples that compose the training set, where $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)})$ are the n input features in the i^{th} training example.
Our problem is, given a training set, to learn the values in a tensor of parameters θ that define the hypothesis $h_\theta : X \to Y$ s.t. $h_\theta(x)$ is a good predictor for the output variable y.

1.1.2 Cost function

Definition 1.4. *Let θ in K be the parameters that the algorithm is trying to set. A cost function*

$$J(\theta) : K \to \mathbb{R}$$

is used to measure the accuracy of our hypothesis. In regression problems, it usually takes an average difference between the predicted results (given when applying the input values to the hypothesis function) and the real output values.

Some of the most known applications are regression, classification or clustering.

1.1.3 Training

This section goes step by step through some essential concepts that compose the process of training an algorithm.

Suppose we are using Multivariate Linear Regression. In this case, and with the notation we defined above, consider the following model representation:
Let m be the number of training examples and n the number of features. The hypothesis function is defined as

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n \tag{1.1}$$

Observation 1.5. *We assume $x_0^{(i)} = 1$ so we can simplify matrix operations. Later, it will also be helpful to see what happens with ANN notation.*

Using matrix representation, this can be represented as

$$h_\theta(x) = \begin{bmatrix} \theta_0 & \theta_1 & \cdots & \theta_n \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \theta^T x$$

which will be much more helpful when we want to program it in a code-efficient way.

In the case of multivariate linear regression it is common to use the half squared error as a cost function:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

Gradient Descent

Once the hypothesis and cost function are defined, the next step is to estimate θ, for what Gradient Descent algorithm is used. The main idea is about finding the θ parameters of the hypothesis function that fit the data as well as possible, which is the same as trying to reduce the cost function value with respect to θ. Thus, the best situation is when the cost function is at the minimum value it can take, given our input and output dataset.

To reach this objective, we take the derivatives of our cost function and a parameter α that will determine the length of the steps taken to the minimum. Derivatives tell us the direction to move towards to, so the idea is to take the ones with the steepest descent and a step 'length' defined by α.

We carry on with the assumption of multivariate regression as example. Then, the derivatives:

$$\frac{\partial}{\partial\theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$$

Suppose we start with a first guess for the hypothesis parameters. Then, gradient descent iterates the following equations while h_θ becomes more and more accurate:

$$\begin{aligned}
\theta_0 &= \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \\
\theta_j &= \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x_j^{(i)}
\end{aligned}$$

Actually, when gradient descent looks at every single example of the training set, it is called *batch gradient descent.*

Observation 1.6. *Note that gradient descent is susceptible to local minima. Anyway, in this case, $J(\theta)$ is a convex function so gradient descent always converges.*

Proposition 1.7. *(Feature scaling) Gradient descent can speed up by having each of the input values in roughly the same range.*

To prevent the parameters descend quickly on small ranges and slowly on large ranges, the input variables have to be modified so they are roughly the same. This

prevents gradient descent to inefficiently oscillate down the optimum. The most used transformations are feature scaling and mean normalization.

Usually when training a model, many inefficiencies may have to be managed. One of the main problems is **overfitting**.

Definition 1.8. *Given a training set used for the training of an algorithm, when the resulting parameters end up fitting to closely to this particular set but failing to fit the test set we say there's an overfitting problem.*

When overfitting is affecting our hypothesis, the most popular option is reducing the weight of some terms that may be responsible of the issue in this function. Actually, it's common to regularize all parameters effect in the cost function, and this process is called *regularization*. The regularized cost function in this case looks like:

$$\frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2$$

Using the regularized cost function, we can reduce overfitting. The λ parameter, known as the *regularization parameter*, is responsible of determining how much the contribution of the parameters is affected.

1.2 Deep Learning

In practical terms, deep learning is just a subfield of machine learning. Most machine learning methods work well because of human-designed representations of the data; in the first step one tries to describe the data with features a CPU can understand and pass that information to a learning algorithm that finally optimizes weights to improve the final prediction. One of the main distinctions we can do with deep learning is the representation learning; this models attempt to automatically learn good features and representations in multiple levels of the process. A deep learning model will constantly analyze data with a logic structure that tries to simulate how human brains take conclusions. To achieve it, layered structured algorithms are used: artificial neural networks (ANN).

1.2.1 Artificial Neural Networks

Consider the machine learning problem in the previous section but now with a non-linear hypothesis function $h_{\Theta,b}(x)$. The main structure in neural nets are neurons (figure 1.1), computational units that take inputs which are channelled to outputs.

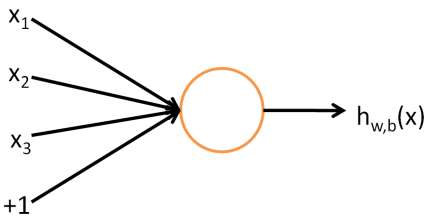


Figure 1.1: Source: [22]

It consists in the input layer that goes into another node (layer 2 or output layer) which finally outputs the predicted value with the corresponding hypothesis, also named activation function. Having this basic idea of how a neuron is defined, we can expand it to higher dimensions so we can properly define neural networks.

The first type of developed neural network is the regular Feedforward Neural Network (FNN), also called Dense or Fully-connected NN. It does not take into account any particular structure but it will be useful to have a general idea of neural networks architectures.

Definition 1.9. *The components of a neural network are:*

- n_l*: number on layers without counting the input layer.*
- $l \in \{1, ..., n_l\}$*: layers.*
- s_l*: number of units/neurons in layer* l*.*
- $b_i^{(l)}$*: bias term of layer* l *(analogous to the* θ_0 *term in 1.1.3).*
- M*: number of training examples.*
- N*: number of features of the input variables.*
- $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, ..., x_N^{(i)})$ *where* $i \in \{1, 2, ..., M\}$*: input variables.*
- $y^{(i)}$ *where* $i \in \{1, 2, ..., M\}$*: output variables to be predicted.*
- $\hat{y}^{(i)}$ *where* $i \in \{1, 2, ..., M\}$*: output of the network.*
- $\Theta^{(l)}$*: matrix of weights assigned to relations from layer* l *to layer* $l+1$*.* $\Theta_{p,q}^{(l)}$ *is the multiplicative factor for unit* $q \in \{1, 2, ..., s_l\}$ *that affects to unit* $p \in \{1, 2, ..., s_{l+1}\}$*.*
- f*: activation function. This one may differ from one layer to another but, to simplify how neural nets work, let's assume it is the same for every layer.*
- $a_i^{(j)}$*: activation of unit* i *in layer* j*. Let* $a^{(1)} = x$*.*

Definition 1.10. (Feedforward Neural Networks) *A FNN is a neural network formed by one input layer, one or more hidden layers and one output layer. From the first hidden layer on, the neurons from one layer are directly connected with the ones before and after them with two types of relationship: a weight feature and an activation feature.*

Let us show how the relationships mentioned in definition (1.10) work. This process is called *forward propagation*. We will use vectorized notation since ML looks for the most efficient implementations in practice and it will also be better to summarize information:

$$\begin{aligned} a^{(1)} &= x \\ z^{(2)} &= \Theta^{(1)} x + b^{(1)} \cdot 1 \\ a^{(2)} &= f(z^{(2)}) \\ &\vdots \\ z^{(n_l)} &= \Theta^{(n_l-1)} a^{(n_l-1)} + b^{(n_l-1)} \cdot 1 \\ a^{(n_l)} &= f(z^{(n_l)}) \end{aligned}$$

Notice that each row of $\Theta^{(l)}$ contains the parameters needed for each unit in layer $l+1$, i.e.

$$z_i^{(l+1)} = \Theta_{i,1}^{(l)} a_1^{(l)} + \Theta_{i,2}^{(l)} a_2^{(l)} + \cdots + \Theta_{i,s_l}^{(l)} a_{s_l}^{(l)} + b_i^{(l)}$$

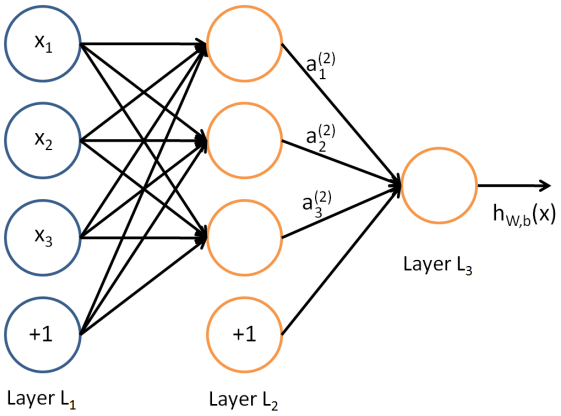


Figure 1.2: Source: [22]

Figure 1.2 represents a FNN with a particular architecture: one single output unit given by $h_{W,b}$, which represents $f(z^{(n_l)})$ in the equations.

In [9], Ian Goodfellow states that "a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly". Let us mention other authors' perspective of this statement like [2] and [11] where it has been technically defined and proved.

Definition 1.11. *(**Squashing function**) A function* $\Psi : \mathbb{R} \to [a, b]$ *being* $a, b \in \mathbb{R}$ *is a squashing function if it holds:*

- *It is a decreasing function*
- $\Psi \xrightarrow{+\infty} b$ *and* $\Psi \xrightarrow{-\infty} a$

Theorem 1.12. (Universal Approximation) *Let* $f : \mathbb{R}^n \to \mathbb{R}^m$ *:* $n, m < \infty$ *be a continuous function in K and K is compact. There exists a single FNN composed of an output layer with a linear activation function and a hidden layer with a squashing activation function that can approximate* f *with a randomly small error* $\epsilon > 0$*, assuming that the neural network has enough neurons in the hidden layer.*

Observation 1.13. *It has also been proven that ANN with a big number of hidden layers are very hard to train for time and overfitting issues and vanishing gradient.*

Activation functions

Some functions are significantly useful in ML as activation functions. The most popular ones are listed below, what will be also useful to comfortably expand on other following sections.

- **Sigmoid** (or Logistic Activation) function. The main reason why we use it is because its image exists in $(0, 1)$ so it is commonly applied with probability outputs.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

  The **Softmax** function is the generalization of the sigmoid function, used to *squeeze* a K-dimensional vector z of real values into a K-dimensional vector of values in $[0, 1]$ with the functionality

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \quad j = 1, ..., K$$

- **Tanh** function. It has similar properties with the sigmoid function but its range goes from -1 to 1.

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

1.2.2 Backpropagation

As well as gradient descent was previously introduced, now backpropagation algorithm comes in to minimize the loss function af an ANN. It is a way of computing gradients applying the chain rule to a recursive method.

Backpropagation starts from the idea of using batch gradient descent. Notice how each step is what we defined as gradient descent but in some way of adapted to neural network structures. Now, the goal is to minimize the cost function $J(\Theta, b)$ as a function of Θ and b. Given a training example $(x^{(i)}, y^{(i)})$, the *half squared error* is

$$J(\Theta, b; x^{(i)}, y^{(i)}) = \frac{1}{2}||h_{\Theta,b}(x^{(i)}) - y^{(i)}||^2$$

and the relative cost function associated to the network is

$$J(\Theta, b) = \left[\frac{1}{m}\sum_{i=1}^{m} J(\Theta, b; x^{(i)}, y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(\Theta_{ji}^{(l)}\right)^2$$

The second term of the equation is the *regularization term* or *weight decay* and it tends to prevent overfitting reducing the magnitude of the weights. The multiplicative factor λ, named *weight decay parameter* controls the relative importance of Θ and the bias term.

Once the cost function is defined, backpropagation tries to minimize it using the following procedure:

1. Initialize each parameter $\Theta_{ij}^{(l)}$ and each $b^{(l)}$. This should be randomly and avoid 0's to all parameters since this would cause $\Theta_{ij}^{(1)}$ and $a_i^{(2)}$ be the same for every i and any input x.

2. Apply gradient descent:

$$\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta, b)$$
$$b_i^{(l)} = b_i^{(l)} - \alpha\frac{\partial}{\partial b_i^{(l)}}J(\Theta, b)$$

   At this point, one notices that this partial derivatives can be time consuming because of the dimensions that the neural network could have. This is why backpropagation is used.

3. Use backpropagation. Assume we are using the notation defined in proposition (1.9) and definition (1.10).

   The first step is called the *forward pass*: given the training examples (x, y), compute the activations througout all the layers of the network including the final output $h_{\Theta,b}(x)$. Then, compute the values $\delta_i^{(l)}$; these are the *error terms* responsible of measuring how much each node i in layer l contributes in possible errors of the output. The first one will be $\delta_i^{(n_l)}$, since we have the real output value. See the following equations to understand the process that comes now:

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} ||y^{(i)} - h_{\Theta,b}(x)|| = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

   For each node i in layer l, s.t. $l \in \{n_l - 1, n_l - 2, ..., 2\}$

$$\delta_i^{(l)} = \Big(\sum_{j=1}^{s_{l+1}} \Theta_{ji}^{(l)} \delta_j^{(l+1)} \Big) \cdot f'(z_i^{(l)})$$

   Once the error terms are calculated, the partial derivatives can be given as:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta, b; x, y) = a_j^{(l)} \cdot \delta_i^{(l+1)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(\Theta, b; x, i) = \delta_i^{(l+1)}$$

1.2.3 Automatic Differentiation

In the last sections, one of the main subjects was how to optimize machine learning and deep learning algorithms and that was why gradient descent and backpropagation were introduced. Anyway, as all this models are implemented in computer programs, Automatic differentiation (AD) comes in given that it simplifies computers' work regarding derivatives. This section is inspired in [10], besides the notation we use.

Methods for computing derivatives can be classified in four categories:

- **Manual computation and coding**.
- **Numerical differentiation**: this method refers to the application of finite difference approximation. Given $f : \mathbb{R}^n \to \mathbb{R}$, $\nabla f = \left(\frac{\partial f}{\partial x_1}, ... \frac{\partial f}{\partial x_n} \right)$ can be approximated by:

$$\frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + h \cdot e_i) - f(x)}{h}$$

where e_i is the *i-th* unit vector and $h > 0$ is a small step size. This method is easy regarding implementation but it can imply increasing errors when the number of dimensions also increases. There exist many techniques for improving numerical differentiation but in practice it is still inefficient regarding computational matters.

- **Symbolic differentiation**. It is the automatic manipulation of expressions in order to obtain derivative expressions (Grambmeier and Kaltofen, 2003), that is put in practice applying differentiation transformations like

$$\begin{aligned}\frac{d}{dx}(f(x)+g(x)) &= \frac{d}{dx}f(x)+\frac{d}{dx}g(x)\\ \frac{d}{dx}(f(x)\cdot g(x)) &= \Big(\frac{d}{dx}f(x)\Big)\cdot g(x)+f(x)\cdot\Big(\frac{d}{dx}g(x)\Big)\end{aligned}$$

  One of the positive sides of symbolic differentiation is that it can compute analytical solutions of extremes, for example when $\frac{d}{dx}f(x) = 0$, what could simplify the final equation. Even then, this process can't be efficient regarding runtime because the expressions can get exponentially larger than the derivative they represent.

  Example: given $f(x) = 25x(x^2-1)(3-2x)^3(1+x+4x^2)^2$ and the multiplication chain rule, this method would develop $f(x)$ as:

$$\begin{aligned}25x(x^2-1)(3-2x)^3(1+x+4x^2)^2+\\ 25x(2x)(3-2x)^2(1+x+4x^2)^2-\\ 150x(x^2-1)(3-2x)^2(1+x+4x^2)^2+\\ 50x(x^2-1)(3-2x)^3(1+x+4x^2)(1+8x)\end{aligned}$$

  while the simplified form is

$$-25(3-2x)^2(320x^7-192x^6-256x^5+21x^4-24x^3+52x^2+4x+3)$$

- **Automatic differentiation**.

The basis of AD comes from the idea of simplifying computations by only storing the values of intermediate sub-expressions in memory. It concerns us since in many machine learning methods the numerical evaluation of derivatives are needed. It takes the idea of symbolic differentiation applied to elementary operations while keeping intermediate numerical results.
Considering a finite set of elementary computations, every equation is built as a combination of them. This set is formed by binary arithmetic operations, the unary sign switch and transcendental functions such as the exponential, the logarithm

and the trigonometric functions. Once this is established, every operation can be differentiated through basic steps of the chain rule. AD has two variants, the forward and the reverse mode.

The following notation is used in [10]. Consider a function $f : \mathbb{R}^n \to \mathbb{R}^m$ constructed by intermediate variables v_i such that

- $v_{i-1} = x_i$, $i = 1, ..., n$ the input variables.
- v_i, $i = 1, ..., l$ the intermediate variables, built from elementary operations.
- $y_{m-i} = v_{l-i}$, $i = m-1, ...0$ the output variables.

Forward Mode

This type of AD is the most simple and it can actually be matched with the application of the classical chain rule to each intermediate variable. Taking any dimension x_i of the input variable in order to obtain the derivative of the p-th dimension of the output, the process consists in assigning

$$\dot{v_k} = \frac{\partial v_k}{\partial x_i}$$

to each v_k, until

$$\dot{y_j} = \frac{\partial y_j}{\partial x_i}$$

can be computed based on the intermediate variables. This process will naturally provide us with the Jacobian of the function f.

Each forward pass of AD is initialized by setting one single $\dot{x}_i$ to 1 and the rest to zero, i.e. setting $\dot{x} = e_i$ where e_i is the i-th unit vector. Given specific input values $x = a$, the Jacobian computation equals

$$\dot{y_j} = \frac{\partial y_j}{\partial x_i}, \quad j = 1, ..., m$$

evaluated at a.

Reverse mode

This mode corresponds to generalized backpropagation given that it propagates derivatives backward from a given output. In order to do this, each v_i is complemented with an adjoint

$$\bar{v_i} = \frac{\partial y_j}{\partial v_i}$$

. In case we would apply backpropagation, y would be the scalar corresponding to the loss function. What reverse AD does requires two timesteps:

- Run the original function forward, computing v_i values and recording the computational dependencies.
- Calculate derivatives by propagating adjoints $\bar{v}_i$ in reverse.

To clear up how this mode is implemented, the chosen example in [10] is attached; it considers the function $f(x_1, x_2) = ln(x_1) + x_1x_2 - sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$. The reverse mode results are expressed in figure 1.3: the left table shows the values resulting from the forward evaluation, used in the next forward step represented in the right table.

Forward Primal Trace

$v_{-1} = x_1$		$= 2$
$v_0 = x_2$		$= 5$
$v_1 = \ln v_{-1}$		$= \ln 2$
$v_2 = v_{-1} \times v_0$		$= 2 \times 5$
$v_3 = \sin v_0$		$= \sin 5$
$v_4 = v_1 + v_2$		$= 0.693 + 10$
$v_5 = v_4 - v_3$		$= 10.693 + 0.959$
$y = v_5$		$= 11.652$

Reverse Adjoint (Derivative) Trace

$\bar{x}_1 = \bar{v}_{-1}$		$= \mathbf{5.5}$
$\bar{x}_2 = \bar{v}_0$		$= \mathbf{1.716}$
$\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}}$	$= \bar{v}_{-1} + \bar{v}_1 / v_{-1}$	$= 5.5$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0}$	$= \bar{v}_0 + \bar{v}_2 \times v_{-1}$	$= 1.716$
$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}}$	$= \bar{v}_2 \times v_0$	$= 5$
$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0}$	$= \bar{v}_3 \times \cos v_0$	$= -0.284$
$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1}$	$= \bar{v}_4 \times 1$	$= 1$
$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3}$	$= \bar{v}_5 \times (-1)$	$= -1$
$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4}$	$= \bar{v}_5 \times 1$	$= 1$
$\bar{v}_5 = \bar{y}$	$= 1$	

Figure 1.3: Reverse mode automatic differentiation example where $y = f(x_1, x_2)$

The most important advantage of the reverse mode is that it makes the process of backpropagation less costly to evaluate since only one application of this method is sufficient to compute the full gradient $\nabla f = \left(\frac{\partial f}{\partial x_1}, ... \frac{\partial f}{\partial x_n}\right)$. Neural networks and many other machine learning algorithms involve gradients with respect to many parameters that would make the forward mode very expensive to evaluate in terms of operations count.

1.3 Recurrent Neural Networks

This section explanations and pictures are basically from the Stanford lecture [6][7]. Also some clarifications from [19] were useful

Since neural networks want to simulate human thinking in some way, it is obvious that the effect of experience should be captured in several problems. Some of the first motivations for using RNN were language models; every time we read any type

of text, everything in a sentence makes sense because of almost everything that was read in the sentences before. This is also adaptable to information recorded taking into account its time sequence.

Depending on the input and the output shapes we work with, many different architectures can be seen in this type of structures. In figure 1.4, consider the red blocks as the input data and the blue ones as the output.
First of all, one must be aware that the inputs of a RNN are more complex than the inputs of a FNN. The architectures we are now focusing on contain loops that allow to store and use data from earlier steps of an input and hence, to better predict the upcoming output in case there are time dependencies.
The key concept in RNN is the hidden state h_t, represented as the green blocks in 1.4. They are the piece of the structure responsible of transferring the required 'past' information to the output and tie the weights between timesteps, i.e. they express how previous steps condition the following ones.

1.3.1 Architecture

The vanilla neural network is the most basic type of recurrent neural network, shown in the left structure of figure 1.4. It receives an input, which is a fixed size object and always fed to the same set of hidden layers to finally produce a single output. Still in some kinds of ML algorithms we want more flexibility in the types of data a model can process. Once we move to this idea of RNN we have a lot more opportunities with the types of input and output data that our networks can handle.

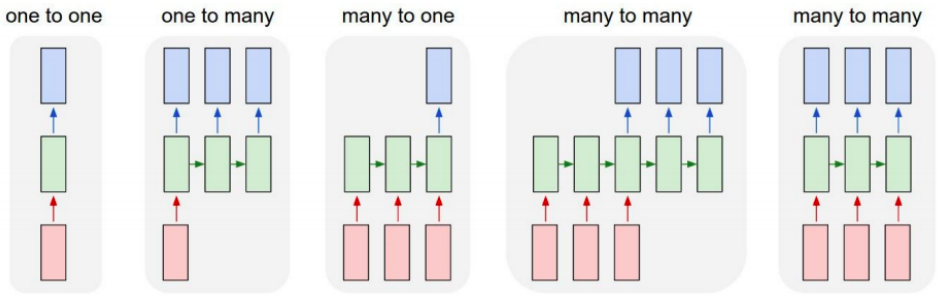


Figure 1.4: Types of RNN structure (Source: [13])

Once we are aware our output and input sequences can have variable length, let us focus in the RNN recurrent core cell, which takes some input x to later feed that input into the RNN which manages the internal hidden state effect. To sum up, the process is based in reading an input, updating the hidden state and producing

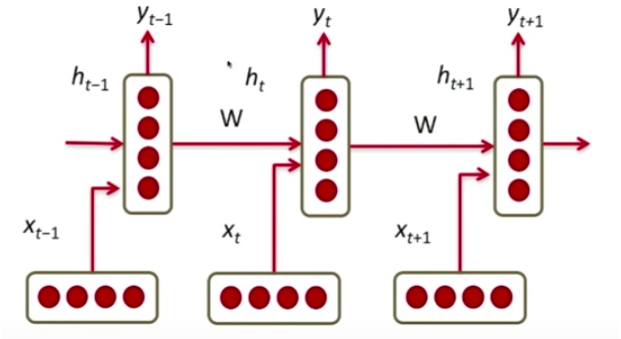
an output; in depth, the hidden state is updated eveytime the RNN reads a new input and then it is fed back to the model for the next time it reads an input, usually also producing an output at every timestep. Let us explain this process more mathematically:
To process a sequence of vectors x at timestep t, a functional form f_W of these relations is applied with the recurrence formula

$$h_t = f_W(h_{t-1}, x_t)$$

where h_t is the updated hidden state and f_W depend on some weights W, while it accepts the previous hidden state and the input of the current step. If we want to produce an output after every step, we could attach fully connected layers that read this h_t at every timestep. One of the most important assumptions is that we use the same function f and weights W at every timestep of the computation.
For the simplest form, which is the vanilla RNN, there is a single hidden state for which the $\tanh$ function is usually the chosen non-linearity in the system. In this case, we say

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$\hat{y}_t = W_{hy}h_t$$

We can think of RNN in two ways: having a h_t that feeds back to itself recurrently, that visually corresponds to figure 1.5b or unrolling this computational graph for multiple timesteps as in figure 1.5a.This second option makes the data flow throw the hidden states and the input, output and weights a little bit more clear. W in figure 1.5 refers to both W_{hh} and W_{xh}, the first one referring to the weights that define the relation between hidden states while W_{xh} defines the relation between h_{t-1} and x_t.
The first step consists in defining h_0 (usually initialized to 0), then we will have some input x_t and they both go into the f_W function producing the next hidden state h_1. This process is repeated while we receive the next input until we consume all the input vectors, shown figure 1.6.
The way in which the W matrix is added in the computational graph is due to emphasize that for each grey block f_W we are reusing the same weights at every timestep of the computation while there will be unique hidden states and inputs. Since how gradient flows in backpropagation when you reuse the same node multiple times in a computational graph, during the backwards pass you end up summing gradients into the W matrix when computing $\frac{\partial L}{\partial W}$, where L represents the loss function. So when we think about the backpropagation for this model then we will have a separate gradient flowing throwgh each of those timesteps and the final gradients for W will be the sum of all those individual per timestep gradients.



(a) Unrolled RNN. (Source:[13]

(b) Rolled RNN (Source: [6])

Figure 1.5: Two ways of seeing recurrent neural networks

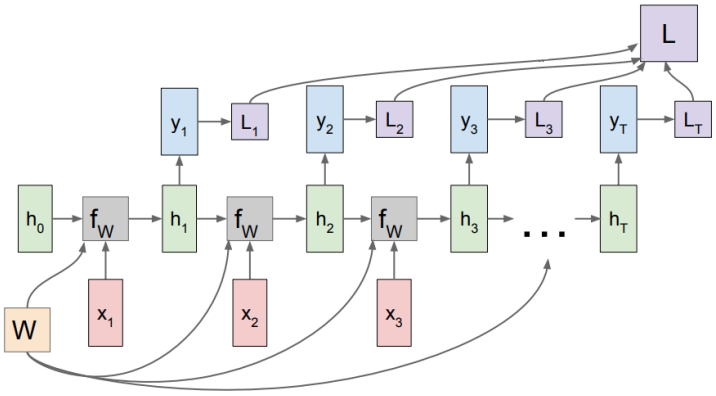


Figure 1.6: Computational graph for a RNN (Source: [6])

As mentioned before, at every timestep h_t might fit into some other little neural network that can produce an output. Then we have a special scalar loss for each timestep whose sum will produce a final loss. In order to train the model, in back-propagation we need to compute the gradient of the final loss with respect to W, so we will have the loss flowing from that final loss into each of these timesteps and then, each of those timesteps will compute a local gradient on the weights.

Backpropagation through time

This idea that we have a sequence and we produce an output at every timestep and then finally compute some loss is sometimes called backpropagation through time because you are imagining that in the forward pass you are stepping forward through time and then during the backward pass you are sort of going backward

through it to compute all the gradients. This can actually be problematic when trying to train with sequences that are very long, for example, training language modelling with the entire text of Wikipedia.Every time we made a gradient step we would have to make a forward pass through the entire text of all Wikipedia, then make a backward pass and then make a single gradient update. This would be super slow and the model would never converge while additionally would take a ridiculous amount of memory.

In practice, people do sort of approximation called truncated backpropagation through time. The idea is that even though our input sequence is very long, when training the model we will step forward for some number of steps, compute a loss only over these subset of data and then backpropagate through this subsequence so now we could make a gradient step. When we repeat the process, we still have the hidden state the hidden state that was computed from the previous batch and and we can carry this h_t forward time so the forward pass will be exactly the same. In the next gradient step we will backpropagate only through the second batch and this process will continue. This procedure is analogous to stochastic gradient descent in the case of sequences.

1.3.2 Issues with backpropagation: LSTM and GRU

So far we have talked about this idea of a single RNN layer, so we need to think a little bit more carefully about what exactly happen to this models when we are tying to train them. Taking a vanilla cell:

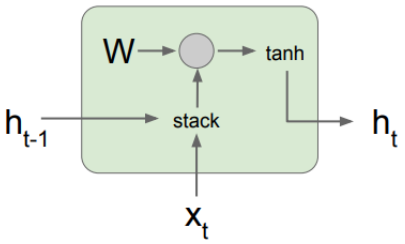


Figure 1.7: Vanilla cell (Source: [6])

we can picture the basic functional form of a RNN. Again, we assume the non-linearity expressed by the tanh function. The computation of the hidden state is as

follows:

$$\begin{aligned} h_t &= \tanh(W_{hh}h_{t-1} + W_{xt}x_t) \\ &= tanh\Big((W_{hh}\, W_{hx} \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\Big) \\ &= \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right) \end{aligned}$$

What happens in this architecture when doing the backward pass when we try to compute gradients? During the backward pass, we will receive derivative of loss with respect to h_t and when passing through the cell we will need to compute derivative of loss with respect to h_{t-1}. When one computes this, what is happening is shown as the red path in figure 1.8; flow will go backwards through the tanh gate and then to the matrix multiplication gate. When implementing this matrix multiplication gate, you end up multiplying by the transposed matrix of W (see **??**). That means that every time we backpropagate through one of this vanilla cells, we end up multiplying by some part of the weight matrix. When stitching many of these RNN cells in sequences, the gradient flow through a sequence of these layers.

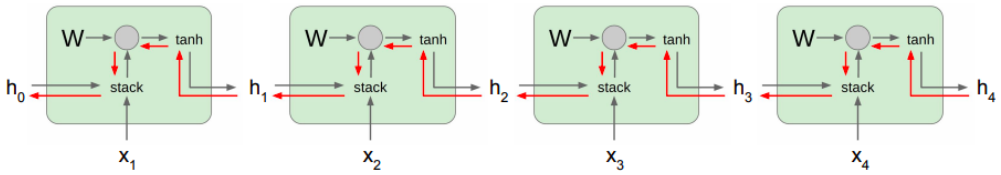


Figure 1.8: (Source: [6])

For several timesteps multiplying by the same over and over again, gradients will explode or vanish depending on weight magnitudes. Assume w_{ij} the largest singular value of the matrix, then two cases can appear:

- If $w_{ij} > 1$, exploding gradients
- If $w_{ij} < 1$, vanishing gradients

The deepening on these concepts is out of the scope of this project, see [9] for more detailed information. These problems make the learning impossible, which solution was found moving to a more complicated RNN structure.

To deal with the problem of long term dependencies, a neural network architecture was found in 1997 which is king of used by everyone now 20 years later: LSTM networks.

LSTM networks

Long-Short Term Memory networs define a slightly fancier recurrence relation for RNN. Some of its main characteristics are:

- Helps aliviate the problem of gradients
- Has better gradient flow properties
- Mantains two hidden states at every step

Vanilla uses recurrence relation to update the hidden state at every step. Now in LSTM we maintain two hidden layers:

- h_t: hidden state analogue to the h_t we had before.
- c_t: the **cell state**. It is an internal kept inside of the LSTM that does not really get exposed to the 'outside world'.

This kind of RNN is composed of 4 gates f, i, g and o. Given the previous hidden state h_t and the given current input x_t, it stacks them multiplying by a big matrix W to compute the four different gates; which all have the same size as the hidden state with the following functionality:

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

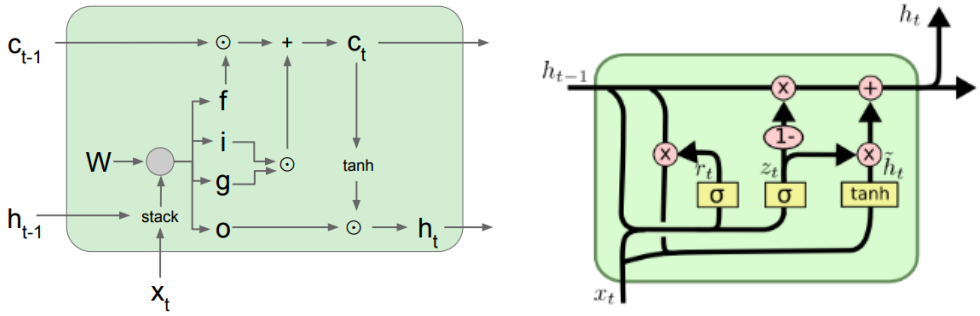
$$h_t = o \odot \tanh(c_t)$$

where $\odot$ represents the element wise multiplication. To define what does each gate do in a friendly way, say

- i: the **input gate**. It measures how much do we want to input into our cell
- f: the **forget gate**. It measures how much we want to forget from the cell of the precious step.
- o: the **output gate**. It expresses how much do we want to write into the input cell.
- g: the **input transformation gate**. Is says how much do we want to reveal our cell to the outside.

The selected function in each case helps to clearly understand how do they work. The first three ones will take values between 0 and 1 as σ representing the sigmoid function, so in case the values would take the extreme value, each gate would represent the decision of including or not the information for the next step in the way each of them represents. In the g gate case, its value will be between -1 and 1.

To sum up, the key to the LSTMs is the cell state, so it kind of runs straight through the entire chain meaning only some minor linear interactions to let useful information just flow along the sequence.



(a) Long-Short Term Memory (Source: [6])

(b) Gated Recurrent Unit (Source: [19])

Figure 1.9: Improvements for the problem of Long-Term dependencies

GRU networks

The Gated Recurrent Unit, introduced by Cho, et al. in 2014 combines the forget and input gates into a single *update gate*. It also merges the cell state and hidden layer making some other changes. The resulting model is simpler than the standard LSTM and has been increasingly popular. Each cell of a GRU network is built as shown in figure 1.9b, where

- $z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$
- $r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$
- $\tilde{h}_t = \tanh(W \cdot [r_t \odot h_{t-1}, x_t])$
- $h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$

1.4 Survival Analysis

Almost all concepts in this section are the same as the ones Egil Martinsson mentioned in [14]. Some statements may vary a little since they were not in the main

scope of this project and there was no need to study them that deeply.

Survival analysis refers to a set of methods that, given some input data, provide the time until the occurrence of the event of interest as the output. Its name comes from trying to predict deaths or some diseases. Usually time is measured with discrete times as days, weeks or months, so a common case could be predicting months until a patient develops cancer.

These models have three main characteristics:

- The response: waiting time until the event of interest occurs. These quantities are the ones WTTE-RNN tries to predict.
- Censoring: some observations may be censored (in some observed cases, the event has already occurred but not in some others).
- Explanatory variables: also called features, whose effect on the output we want to model.

1.4.1 Waiting times

Let $T \in [0, \infty)$ be a non negative random variable representing the time between events, known as *waiting time*; to understand it as it was originally used, we may refer to them as 'survival time' until 'death' respectively in this section.
The main idea of survival analysis is predicting this waiting times under the assumption that when death occurs no events take place anymore. That is why we first go over all the functions we should use to estimate time to one single event and later adapt it to the case when there are recurrent events (several events for one single case, several survival times). The problem and functions are first defined for continuous time, without loss of generality.

1.4.2 Censoring

Imagine that the observations in our data refer to patients, all of them tracked for T timesteps. We can divide this in three different cases:

(i) Patients who died at timestep $t \leq$ T.

(ii) Patients who are still alive at T, i.e. they did not die for the duration of the study.

(iii) Patients who dropped out of the study before T and did not die within that time.

All we know in cases (i) and (ii) is that their waiting time exceeds their last observed time, i.e. they did not die for the duration of the the study. These are called right censored observations.

Definition 1.14. *Observations are called censored when the information about their survival time is incomplete; the most commonly encountered form is right censoring.*

Definition 1.15. *A data point is right censored if it is above a certain value but it is unknown by how much.*

Observation 1.16. *Let y_t^n be the time to event (expressed in timesteps) for patient n at timestep t and T_n the last observed timestep for patient n. For any $t \leq T_n$ after which an event has not occured yet, we say that y_t^n is right censored and we could fix a lower bound $y_t^n \geq T_n - t = \tilde{y}_t^n$.*

Observation 1.17. *Censoring represents a missing type of data, but an important assumption is usually made in survival analysis: censoring is random and non-informative in order to avoid bias.*

To later save this information in the dependent variable of our model, it will be composed of two parts: (y_t^n, u_t^n) where y_t^n is the time to death for patient n at timestep t and u_t^n is the indicator that records if death happened or not (i.e. if the observation is censored or not, being 0 or 1 respectively). Moreover, we will assume we have *non-informative censoring*.

1.4.3 The Survival, Hazard and Cumulative Hazard functions

Two functions dependent on time are estimated: the Hazard and Survival functions. They are key concepts in survival analysis since they describe the distribution on event times. They provide alternative but equivalent characterizations of the distribution of T, and for how they are named, you can easily imagine what they represent.

Assume T continuous random variable with PDF $f(t)$ and CDF $F_T(t) = P(T \leq t)$. As CDF represents the probability that the event occurred by duration t, it is convenient to work with its complementary.

Definition 1.18. (Survival function) *A survival function*

$$S_T(t) : [0, \infty) \to \mathbb{R}$$

is defined as

$$S_T(t) = P(t < T) = 1 - F_T(t) = \int_t^\infty f_T(x)dx \tag{1.2}$$

The survival function gives, for every time, the probability of surviving, i.e. the probability of non experiencing and event, up to that time.

Definition 1.19. (Hazard function(HF)) *A hazard function*

$$\lambda(t) = \lim_{h \to 0} \frac{P(t < T \leq t + h | T > t)}{h} \tag{1.3}$$

is a conditional density with the properties:

1. $0 \leq \lambda(t)\ \forall t \geq 0$
2. $\int_0^\infty \lambda(w)dw = \infty$

Observation 1.20. *The numerator of expression 1.3 is the conditional probability that the event will take place in $(t, t+h]$ given the condition that it has not occurred before, while h is the with of the interval. Dividing them, we obtain the rate of event occurrence per unit of time. Thus, the limit when $h \to 0$ can be seen as the instantaneous event rate (also known as failure rate).*

Proposition 1.21. *The expression 1.3 is equivalent to*

$$\lambda(t) = \frac{f(t)}{S(t)} \tag{1.4}$$

Proof.

$$\begin{aligned} &P(t < T \leq t + h | T > t) = \frac{P(\{t < T \leq t + h\} \cap \{T > t\})}{P(T \geq t)} = \frac{P(t < T \leq t + h)}{P(T \geq t)} \Rightarrow \\ &\Rightarrow \lambda(t) = \lim_{h \to 0} \frac{P(t < T \leq t + h)}{h} \cdot \frac{1}{S(t-)} = \frac{f(t)}{S(t-)} \end{aligned} \tag{1.5}$$

where $S(t-) = \lim_{s \to t} S(s)$. As $S(t)$ is contiunous, we can state

$$S(t-)) = S(t)$$

. □

Definition 1.22. (Cumulative Hazard Function (CHF)) *Given $\lambda(t)$ a hazard function, its integral*

$$\Lambda(t) = \int_0^t \lambda(w)dw$$

is called Cumulative Hazard Function.

Proposition 1.23. $\Lambda(t)$ *induces a CDF for a positive random variable* T *with CDF*

$$P(T < t) = F_T(t) = 1 - e^{-\Lambda(t)}$$

and PDF:

$$f_T(t) = e^{-\Lambda(t)}\lambda(t)$$

Proof. First of all, we prove that $F_T(t)$ holds CDF properties:

(i) $\lambda(w) \geq 0 \Rightarrow \int_0^t \lambda(w)dw = \Lambda(t)$ is an increasing function. Since $F_T(t) = 1 - e^{-\Lambda(t)}$, $F_T(t)$ in an increasing function $\forall t \geq 0$.

(ii) $\lim_{t\to-\infty} \Lambda(t) = 0$ and $\lim_{t\to\infty} \Lambda(t) = \infty$. Since $F_T(t) = 1 - e^{-\Lambda(t)}$, $\lim_{t\to-\infty} F_T(t) = 0$ and $\lim_{t\to\infty} F_T(t) = 1$.

(iii)

$$f_T(T) = \frac{\partial}{\partial t}(1 - e^{-\Lambda(t)}) = e^{-\Lambda(t)}\lambda(t)$$

□

Whenever we think about the data we may work with, notice there will be cases when the event has not occurred yet (censored data). Thus, the fundamental idea is to take the portion of the distribution that is below a certain threshold t and to flip it left-to-right onto the positive portion of the horizontal axis. To do that, we consider the random variable defined in 1.24 as Y_t.

Definition 1.24. (Conditional Excess Distribution) *Name* $Y_t = \{T - t\}|\{T > t\}$ *the conditional excess distribution of* T *at* t*. Then, if* T *has* F_T *as CDF, the CDF of* Y_t *is*

$$F(t, s) = 1 - e^{-(\Lambda(t+s)-\Lambda(t))}$$

Proof.

$$F(t,s) = P(Y_T \leq s) = P(T \leq t + s|T > t) =$$

$$= \frac{P(T \leq t + s \cap T > t)}{P(T > t)} = \frac{F_T(t+s) - F_T(t)}{1 - F_T(t)} =$$

$$= \frac{S_T(t) - S_T(t+s)}{S_T(t)} = \frac{e^{-\Lambda(t+s)} - e^{-\Lambda(t)}}{e^{-\Lambda(t)}} =$$

$$= 1 - e^{-(\Lambda(t+s)-\Lambda(t))}$$

□

Observation 1.25. *Given $F(t,s)$, the PDF of Y_t is obtained:*

$$f(t,s) = \frac{\partial}{\partial s}\frac{F_T(t+s)-F_T(t)}{1-F_T(t)} = \frac{f_T(t+s)}{1-F_T(t)} = \frac{f_T(t+s)}{S_T(t)} = e^{-\Lambda(t+s)-\Lambda(t))}\lambda(t+s).$$

Following definition (1.18), $1-F(t,s) = S(t,s)$.

We are interested in situations where events may take place after some of them already did, what we call recurrent events. To be able to work with that, the concepts previously defined need to be adapted to this case.

Definition 1.26. (Recurrent Cumulative Hazard Function (RCHF)) *Let $s \in [0,\infty)$ be the coming time. $\forall x \in \mathbb{R}$ the RCHF is a function $R : \mathbb{R}^m \times \mathbb{R}_+ \to \mathbb{R}_+$ s.t.*

- $\frac{\partial R}{\partial S}(x,s) \geq 0 \ \forall s \geq 0$
- $R(x,s) \to \infty$ *as* $s \to \infty$
- $R(x,s) = 0 \ \forall s \leq 0$

Notice that, from the previous definition of the hazard function, we can understand $\frac{\partial R}{\partial s}(x,s) = R_s(x,s)$ as the hazard function for events over the coming time s.

Definition 1.27. *A RCHF is symmetric if $R(t,s) = R(0,t+s) - R(0,t)$.*

Observation 1.28. *Given a symmetric RCHF*

- *There is a CDF $F_T(t)$ such that $F(t,s) = 1 - e^{-R(t,s)}$*
- $F(0,t) = F_T(t)$
- $F(t,s) = e^{-R(t,s)} \cdot R_s(t,s)$
- $R(t,s) = \Lambda(t+s) - \Lambda(t)$

R is defined so we can relate the information we have at time t to future events at time $t+s$.
Commonly, businesses are saving their customers purchases (or log-in's, clicks, etc.) times with discrete units. Anyway, if they had the exact time in the shape DD-MM-YY-hh-mm-ss it may be also more comfortable to turn this data into a discrete-time set. Because of this, let us show how these functions look for discrete cases in the last section.

Definition 1.29. (Step Cumulative Hazard Function (SCHF)) *Let Λ be some CHF and R a symmetric RCHF defined as above. SCHF $d(t)$ is defined as the HF integrated in $[t,t+1)$ $\forall t \in \{0,1,...,n\}$:*

$$d(t) = R(t,1) = \Lambda(t+1) - \Lambda(t) = \int_t^{t+1} \lambda(w)dw \tag{1.6}$$

1.4.4 Log-Likelihood

Let $\mathcal{L}(t,\theta)$ be the likelihood function known as the joint PDF of the sample $T = \{T_1, ..., T_n\}$ - denoted as $f_{T|\theta}(t)$ (PMF if discrete variable) -. We are interested in using a likelihood that works as a loss function for the model, depending on θ and use it to maximize the probability of $T = t$ given θ, which will be the same as minimizing the log-likelihood $log(\mathcal{L}(t,\theta))$.

Survival analysis states that the likelihood function to manage censored data is

$$\mathcal{L}(t_i,\theta) = \prod_{t_i \in unc.} P(T = t_i|\theta) \prod_{t_i \in l.c.} (T < t_i|\theta) \prod_{t_i \in r.c.} (T > t_i|\theta) \prod_{t_i \in i.c.} (t_{i,l} < T < t_{i,r}|\theta) \tag{1.7}$$

where $unc., l.c., r.c., i.c.$ mean uncensored, left censored, right censored and interval censored data respectively. We are only interested in uncensored and right censored observations, so it is also useful to know that, given the survival functions we defined in this section,

- $P(T = t_i|\theta) = f(t_i|\theta)$
- $P(T > t_i|\theta) = S(t_i|\theta)$

Theorem 1.30. (Likelihood for right-censored data) *Let T be a random waiting-time parameterized by θ. Under the assumption of non-informative right censoring we can write the likelihood as:*

$$\mathcal{L}(t,\theta) = \begin{cases} P(T = t_i|\theta) = f(t_i) & \text{if uncensored} \\ P(T > t_i|\theta) = S(t_i) & \text{if right censored} \end{cases}$$

This turns out to result in a log-likelihood with the following shape:

$$log\mathcal{L} = \sum_{i=1}^{n} \Big(u_i \cdot log(\lambda(t_i)) - \Lambda(t_i) \Big)$$

Proof. Under the following assumption:

$$\mathcal{L}(t,\theta) = \prod_{t_i \in unc.} P(T = t_i|\theta) \prod_{t_i \in r.c.} P(T > t_i|\theta)$$

Let u_i be the indicator for censoring: 0 if it is censored or 1 if it is not. Then,

$$\mathcal{L}(t,\theta) = \prod_{t_i} P(T = t_i|\theta)^{u_i} P(T > t_i|\theta)^{1-u_i}.$$

$$\begin{aligned} log\big(\mathcal{L}(t,\theta)\big) &= u_i \sum_{i=1}^{n} log\big(P(T=t_i|\theta)\big) + (1-u_i) \sum_{i=1}^{n} log\big(P(T>t_i|\theta)\big) \\ &= u_i \sum_{i=1}^{n} log\big(f(t_i)\big) + (1-u_i) \sum_{i=1}^{n} log\big(S(t_i)\big) \\ &= u_i \sum_{i=1}^{n} log\big((e^{-\Lambda(t_i)}\lambda(t_i)\big) + (1-u_i) \sum_{i=1}^{n} log\big(e^{-\Lambda(t_i)}\big) \\ &= \sum_{i=1}^{n} \Big(u_i log\lambda(t_i) - \Lambda(t_i)\Big) \end{aligned}$$

□

Observation 1.31. (Likelihood for right-censored data, discrete case) *Let T with CHF Λ be a positive random variable and T_d its induced discrete random variable s.t. $T_d = t \Leftrightarrow T \in [t, t+1)$. Then*

$$\mathcal{L}_d(t,\theta) = \begin{cases} P(T=t_i|\theta) & \text{if uncensored} \\ P(T>t_i|\theta) & \text{if right censored} \end{cases}$$

Where $p(t)$ is the probability mass function. In this case, we have

$$log\mathcal{L}_d = \sum_{i=1}^{n} \Big(u_i \cdot log(e^{d(t)} - 1) - \Lambda(t_i + 1)\Big)$$

Proof.

$$\begin{aligned} \mathcal{L}_d(t,\theta) &= \prod_{t_i} P(T_d = t_i|\theta)^{u_i} \cdot P(T_d > t_i|\theta)^{1-u_i} \\ &= \prod_{t_i} P(T \in [t_i, t_i+1)|\theta)^{u_i} \cdot S_d(t_i+1)^{1-u_i} \\ &= \prod_{t_i} (S(t_i) - S(t_i+1))^{u_i} \cdot S_d(t_i+1)^{1-u_i} \\ &= \prod_{t_i} (e^{-\Lambda(t_i)} - e^{-\Lambda(t_i+1)})^{u_i} \cdot e^{-\Lambda(t_i+1)(1-u_i)} \\ &= \prod_{t_i} (e^{-\Lambda(t_i)} - e^{-\Lambda(t_i+1)})^{u_i} \cdot e^{\Lambda(t_i+1)(u_i-1)} \\ &= \prod_{t_i} (e^{-\Lambda(t_i)+\Lambda(t_i+1)} - 1)^{u_i} \cdot e^{-\Lambda(t_i+1)} \\ &= \prod_{t_i} (e^{d(t)} - 1)^{u_i} \cdot e^{-\Lambda(t_i+1)} \end{aligned}$$

$$\begin{aligned} log\big(\mathcal{L}_d(t,\theta)\big) &= \sum_{i=1}^{n} \Big(log\Big((e^{d(t)}-1)^{u_i}\Big) + log\Big(e^{-\Lambda(t_i+1)}\Big)\Big) \\ &= \sum_{i=1}^{n} \Big(u_i \cdot log(e^{d(t)} - 1) - \Lambda(t_i+1)\Big) \end{aligned}$$

□

1.4.5 Weibull distribution

The Weibull distribution is a continuous probability distribution named after the Swedish mathematician Waloddi Weibull. He originally proposed the distribution as a model for material breaking strength, but recognized the potential of the distribution in his 1951 paper *A Statistical Distribution Function of Wide Applicability*. Today, it's commonly used to analyze life data and model failure times.

In WTTE-RNN the Weibull distribution and its discrete variant are used. Even its parametrization may vary a little bit depending on the source we look for it, the following equations will be assumed from now on as the ones corresponding to the Weibull distribution.

Given $t \in [0, \infty), \alpha \in (0, \infty)$ (called *scale parameter*) and $\beta \in (0, \infty)$ (called *shape parameter*), a random variable $T \sim$Weibull(α, β) with:

- Cumulative distribution function:

$$F(t) = 1 - exp\left[-\left(\frac{t}{\alpha}\right)^{\beta} \right] = 1 - e^{-\Lambda(t)}$$

- Probability density function:

$$f(x) = \frac{\beta}{\alpha}\left(\frac{t}{\alpha}\right)^{\beta-1} exp\left[-\left(\frac{t}{\alpha}\right)^{\beta} \right] = \lambda(t) \cdot e^{-\Lambda(t)}$$

- Cumulative Hazard function:

$$\Lambda(t) = \left(\frac{t}{\alpha}\right)^{\beta}$$

- Hazard function:

$$\lambda(t) = \left(\frac{t}{\alpha}\right)^{\beta-1} \cdot \frac{\beta}{\alpha}$$

- Survival function:

$$S(t) = e^{-(\frac{t}{\alpha})^{\beta}}$$

- Cumulative mass function:

$$P(T_d \leq t) = P(T \leq t+1) = 1 - exp\left[-\left(\frac{t+1}{\alpha}\right)^{\beta} \right]$$

- Probability mass function:

$$P(T_d = t) = P(t \leq T \leq t+1) = exp\left[-\left(\frac{t}{\alpha}\right)^{\beta} \right] - exp\left[-\left(\frac{t+1}{\alpha}\right)^{\beta} \right]$$

Chapter 2

The model: WTTE-RNN

The aim of this chapter is to deeply describe the essential problem we want to face and how the model tries to solve it. We need to talk about initial hypothesis, define variables, the objective function and how concepts mentioned before are adapted to this particular case.

2.1 Churn prediction

Companies may need to know which is their customer churn-rate or churn probabilities for particular clients. Anyway, before trying to predict these indicators, the most important step is to properly define what churn means, and that does not mean there is a particular definition for it but that some assumptions must be fixed. Who do we define as churned customers? The ones that haven't had activity for a declared period of time? Alternatively, depending on the type of data, we may have a signal for when they really left and that moment is what we would like to predict.

We can consider that what churned means tends to be related with the idea of defining b_t within a preset timeframe τ defined as:

$$b_t = \begin{cases} 1 & \text{if event in } \quad [t, t+\tau) \\ 0 & \text{if no event in } \quad [t, t+\tau) \\ \text{unknown} & \text{otherwise (censored).} \end{cases}$$

So after defining churned customers as the ones who won't purchase in τ timesteps, the target value to predict would be b_t.

2.1.1 Censoring

Considering data-types that could be applied to this model, notice we could talk about right censored data. In this case we want to predict future without having all the information we need to train our algorithm at one hundred per cent.

Our motivation will always be to predict future values based on the observed past (from when we first saw the customer up until now), one of the reasons why we are using survival analysis. For training, a lower bound will be given for right censored data.

2.1.2 Some applied methods nowadays

First of all, from a **deterministic standpoint** of the customer churn problem, many algorithms have been trying to predict churn defining it as a binary variable. In order to predict it, the most known machine learning algorithms are **Logistic Regression**, **Decision Trees** and **Random Forests**. This methods are out of the scope of this project since they address the prediction from a different perspective than probabilistic models. Documentation about them can be found in []

Shopify **??** is an ecommerce platform for online stores and retail point-of-sale systems. To help their merchants, one of their aims is to handle the problem of determining customer churn, and so, customer retention. One of the main distinctions they do to start facing the problem is between the type of business they focus on. The problem potentially differs when we talk about a contractual business or a non-contractual business since in the first case there's no need of managing with censored data because we know when clients dissolve their contract.

They also stress the importance of how do we describe customer churn and that time measure tends to be discrete in all kind of businesses. Due to the limitations that churn definition can involve, for example an incorrect definition or a binary approach to the solution, **probabilistic models** are becoming commonly used given that they provide a new way to think about customer churn. Two parameters are associated to customer behaviour: the rate of purchasing, λ, and the probability of churn event, p. The idea is to predict them and so to create unobservables from probability distributions.

In order to distinguish between different types of businesses, Shopify's idea is to apply this estimation of parameters under the assumption that different businesses imply different distributions for them. To deeply understand how these methods are run, see [3].

Why WTTE-RNN?

As some of the mentioned options may be a great solution, WTTE-RNN is predicting the time to next event, what turns the output data into an understandable measure that can be interpreted as churned or not depending on how much time to event is considered a churned result, what makes the model more adaptable to different cases. This approach could be interpreted as predicting 'non-churn'; while predicting the time to next event, which could be the next purchase of a customer, we are actually predicting their future behaviour and those who have longer times can be read as 'more-churned' ones.

Another pro of this method is that it considers the sequential order of events as an informative feature (because of RNN). This means that the data we may use as input for this method should belong to a kind of business to which we can relate this aspect. Further on, we are going to see some aspects that the model needs to deal with:

(i) Work with recurrent events

(ii) Handle time varying covariates

(iii) Learn temporal patterns

(iv) Handle sequences of varying length

(v) Learn with censored data

(vi) Make flexible predictions

2.2 Objective function: the Weibull loss

As survival analysis is a main tool to find the parameters that give us the distribution that fit the most the behaviour of the waiting times, to achieve it, our aim is to maximize the log-likelihood.

Let α and β be the parameters of the Weibull-distribution. They will be treated as functions of the input data $x_{0;t}$, so the Cumulative Weibull Hazard (CWH) looks like:

$$\hat{R}(x_{0;t}, y_t) = \left(\frac{y_t}{\alpha(x_{0;t})}\right)^{\beta(x_{0;t})} \tag{2.1}$$

and the Weibull Hazard (WH):

$$\hat{R}_s(x_{0;t}, y_t) = \left(\frac{\beta(x_{0;t})}{\alpha(x_{0;t})}\right)\left(\frac{y_t}{\alpha(x_{0;t})}\right)^{\beta(x_{0;t})-1} \tag{2.2}$$

Assume we have one single sequence of data. Having this and remembering theorem (1.30), our main goal becomes finding α and β such that:

$$\max_{\alpha,\beta>0} \mathcal{L}(\alpha,\beta,y,u,x) := \sum_{t=1}^{T} \left(u_t \cdot \left[\beta(x_{0:t}) \cdot log\Big(\frac{y_t}{\alpha(x_{0:t})}\Big) + log(\beta(x_{0:t})) \right] - \left(\frac{y_t}{\alpha(x_{0:t})}\right)^{\beta(x_{0:t})} \right) \tag{2.3}$$

for continuous data and

$$\max_{\alpha,\beta>0} \mathcal{L}_d(\alpha,\beta,y,u,x) := \sum_{t=1}^{T} \left(u_t \cdot \Big[log(e^{d(t)} - 1) \Big] - \left(\frac{y_t + 1}{\alpha(x_{0:t})}\right)^{\beta(x_{0:t})} \right) \tag{2.4}$$

for discrete data (observation 1.31).

Proof.

$$\begin{aligned}
\mathcal{L}(\alpha,\beta,y,u,x) &= \sum_{t=1}^{T} \Big(u_t log\lambda(y_t) - \Lambda(y_t) \Big) \\
&= \sum_{t=1}^{T} \left[u_t \Big((\beta - 1) log\Big(\frac{y_t}{\alpha}\Big) + log\Big(\frac{\beta}{\alpha}\Big) \Big) - \Big(\frac{y_t}{\alpha}\Big)^{\beta} \right] \\
&= \sum_{t=1}^{T} \left(u_t \Big((\beta - 1) logy_t - (\beta - 1) log\alpha + log\beta - log\alpha \Big) - \Big(\frac{y_t}{\alpha}\Big)^{\beta} \right) \\
&= \sum_{t=1}^{T} \left(u_t \Big(\beta(logy_t - log\alpha) + log\beta - log(y_t) \Big) - \Big(\frac{y_t}{\alpha}\Big)^{\beta} \right)
\end{aligned}$$

Since this will be maximized with respect to α and β, and t is not a function of any of these two parameters, we can continue the procedure dismissing $u_t \cdot logy_t$, what let us use the representation

$$\mathcal{L}(\alpha,\beta,y,u,x) \propto \sum_{t=1}^{T} \left(u_t \Big(\beta(logy_t - log\alpha) + log\beta \Big) - \Big(\frac{y_t}{\alpha}\Big)^{\beta} \right)$$

For the discrete case:

$$\begin{aligned}
\mathcal{L}_d(\alpha,\beta,y,u,x) &= \sum_{t=1}^{T} \Big(u_i \cdot log(e^{d(t)} - 1) - \Lambda(t_i + 1) \Big) \\
&= \sum_{t=1}^{T} \left[u_t \cdot log\Big[exp\Big(\Lambda(y_t + 1) - \Lambda(y_t) \Big) \Big] - \Lambda(y_t + 1) \right] \\
&= \sum_{t=1}^{T} \left[u_t \cdot log\left[exp\left(\Big(\frac{y_t + 1}{\alpha}\Big)^{\beta} - \Big(\frac{y_t}{\alpha}\Big)^{\beta} \right) \right] - \Big(\frac{y_t + 1}{\alpha}\Big)^{\beta} \right]
\end{aligned}$$

□

Since α and β will be the outputs of the Recurrent Neural Network, being W its parameters, the objective functions of the model are the same as (2.3) and (2.4) but instead of maximizing with respect to α and β, doing it with respect to W. We refer to Weibull "loss" since we need a loss function to minimize, which will be $-\mathcal{L}$.

2.2.1 Gradients

Since the method is implemented with Keras, which implies that gradients are computed with Automatic Differentiation, using activation functions from the package guarantees that there won't be any problem with the involved derivatives. To ensure that the chosen loss function has a unique maxima, the gradients of the Weibull-loss with respect to its parameters are verified:

$$\mathcal{L} \propto u \cdot (\beta \cdot log\Big(\frac{t}{\alpha}\Big) + log\beta) - \Big(\frac{t}{\alpha}\Big)^{\beta}$$

$$\begin{aligned}\frac{\partial}{\partial\beta}\mathcal{L} &= u\Big(log\Big(\frac{t}{\alpha}\Big) + \frac{1}{\beta}\Big) - log\Big(\frac{t}{\alpha}\Big) \cdot \Big(\frac{t}{\alpha}\Big)^{\beta} \\ &= \frac{u}{\beta} + log\Big(\frac{t}{\alpha}\Big)\Big[u - \Big(\frac{t}{\alpha}\Big)^{\beta}\Big] \\ \frac{\partial}{\partial\beta^2}\mathcal{L} &= -u \cdot \frac{1}{\beta^2} - log\Big(\frac{t}{\alpha}\Big)^2 \cdot \Big(\frac{t}{\alpha}\Big)\end{aligned}$$

$$\begin{aligned}\frac{\partial}{\partial\alpha}\mathcal{L} &= -u \cdot \beta \cdot \frac{1}{\alpha} + \beta \cdot t^{\beta} \cdot \frac{1}{\alpha^{\beta+1}} \\ &= \frac{\beta}{\alpha} \cdot (-u + \Big(\frac{t}{\alpha}\Big)^{\beta}) \\ \frac{\partial}{\partial\alpha^2}\mathcal{L} &= \frac{\beta}{\alpha^2} \cdot (u - (\beta+1) \cdot \Big(\frac{t}{\alpha}\Big)^{\beta})\end{aligned}$$

$$\frac{\partial}{\partial\beta\partial\alpha}\mathcal{L} = \frac{1}{\alpha}\Big(- u + \Big(\frac{t}{\alpha}\Big)^{\beta}\Big[1 + \beta \cdot log\Big(\frac{t}{\alpha}\Big)\Big]\Big)$$

For the discrete case, being θ the parameters of the Weibull distribution, the only problem we can observe is that, since the derivative of the log-likelihood is

$$u \cdot \frac{e^{d(t)}}{e^{d(t)} - 1} \cdot \frac{\partial d(t)}{\partial\theta} - \frac{\partial\Lambda(t+1)}{\partial\theta} = u \cdot \frac{\Lambda_\theta(t+1) - \Lambda_\theta(t)}{1 - e^{-d(t)}} - \Lambda_\theta(t+1)$$

it may cause problems when $e^{d(t)} - 1$ or $1 - e^{-d(t)}$ are near 0. Anyway, using proper initialization of the parameters for the experiments, there was no issue found regarding this.

Though it is out of the scope of this project to dig deeply on this matter, it has been proven how a location-scale-transformation for the Weibull variable and the application of the Gumbel distribution provide a maximum likelihood estimation (MLE) that is unique for multiple censored data. In addition, this MLE can be found using standard algorithms due to the fact that there are no other local maxima [8].

2.3 The optimization problem

Despite the objective of this method will be used for two different approaches of the initial problem (meaning that some initial assumptions and data may change a little), how we mathematically define its components won't be an issue.

All the elements needed for the problem are defined in the following lines with their respective notation, assuming that the time variable is discrete:

- $n \in \{1, 2, ..., N\}$: sequence number.
- $t \in \{1, 2, ..., T\}$: timesteps of a sequence. ($t^{(i)} \in \{1, 2, ..., T_i\}$ s.t. $i \in \{1, 2, ..., n\}$ timesteps of sequence i).
- $\{1, 2, ..., M\} \in J$: set of features.
- $y_t^{(n)} \in [0, \infty)$: time to event at timestep t for sequence n.
- $u_t^{(n)} \in \{0, 1\}$: failure indicator at timestep t for sequence n. 0 if timestep t is right censored or 1 otherwise.
- $x_t^{(n)} \in \mathbb{R}^J$: vector of features at timestep t for sequence n.
- $x_{0:t} \in \mathbb{R}^{t \times M}$: sequence of feature vectors from start until timestep t.
- $\hat{R}(x_{0:t}, y_t) = \Lambda(y_t) \in [0, \infty)$: estimated Recurrent Cumulative Hazard Function evaluated at t.
- $\alpha(x_{0:t})$, $\beta(x_{0:t}) \in (0, \infty)$: positive functions. They will be the predicted parameters of the Weibull distribution.
- Θ: parameters of the RNN.
- The objective function:

$$\max_{\Theta} \mathcal{L}_d(\alpha, \beta, y, u, x) := \sum_{n=1}^{N} \sum_{t=1}^{T_n} \left(u_t \cdot \left[log(e^{d(t)} - 1) \right] - \left(\frac{y_t + 1}{\alpha(x_{0:t})} \right)^{\beta(x_{0:t})} \right)$$

In order to adapt this function to a loss that can be used by the RNN, we will focus on an objective function we wish to minimize:

$$\min_{\Theta} \mathcal{L}_d(\alpha, \beta, y, u, x) := -\sum_{n=1}^{N}\sum_{t=1}^{T_n}\left(u_t \cdot \left[log(e^{d(t)} - 1)\right] - \left(\frac{y_t + 1}{\alpha(x_{0:t})}\right)^{\beta(x_{0:t})}\right)$$

Now we have all the information about the data we may want to analyse and which loss function may help to find a good solution, we are ready to consider to which neural network architecture it is assigned.

2.3.1 RNN structure

The chosen architecture for the model is based on:

- Input values of given shape: x_i are the input vectors of a sequence at timestep i. To encode it, this is written in a matrix which rows are timesteps and columns are the features of the input.
- ANN: here LSTM or GRU can be chosen and, depending on the dimensions of the data, we could make several experiments to find which specific structure fits better. After this one, a dense layer is added in order to obtain two values, to which the activation function is applied.
- Output values: these are the parameters of the Weibull distribution we try to predict. We refer to them as *output values* instead of *parameters* to avoid confusions with the neural network parameters. Since α and β must be positive (because of the definition of Weibull parameters' domain), Exponential and SoftMax functions are chosen as activation functions in the last dense layer and thus provide α and β respectively.

Using Python language and Keras API for the RNN implementation 2.1, this process is implemented with the code in figure. Callbacks [12] are functions to be applied at given stages of the training procedure. They can be used to give a view on internal states and statistics of the model during training as well as to reduce the learning rate when a metric has stopped improving among other utilities. The Sequential model is a linear stack of layers, that will be the ones added in the following lines: the Masking layer that takes care of the varying length of the sequences (in the preprocessing of the data they will have been modified so this layer can skip those steps of the sequences filled with a mask value), the GRU layer explained in section 1.3.2, a fully connected layer that outputs 2 values and finally the activation functions, which are implemented in the created Python package for this specific model [16].

```
reduce_lr = callbacks.ReduceLROnPlateau(monitor='loss',
                                        factor  =0.5,
                                        patience=50,
                                        verbose=0,
                                        mode='auto',
                                        epsilon=0.0001,
                                        cooldown=0,
                                        min_lr=1e-8)

nanterminator = callbacks.TerminateOnNaN()
history = callbacks.History()
weightwatcher = WeightWatcher(per_batch =False,per_epoch= True)
n_features = x_train.shape[-1]

def base_model():
    model = Sequential()
    model.add(Masking(mask_value=mask_value,input_shape=(None, n_features)))
    model.add(GRU(3,activation='tanh',return_sequences=True))
    return model
def wtte_rnn():
    model = base_model()

    model.add(TimeDistributed(Dense(2)))
    model.add(Lambda(wtte.output_lambda,
                     arguments={"init_alpha":init_alpha,
                                "max_beta_value":4.0,
                                "alpha_kernel_scalefactor":0.5}))

    loss = wtte.loss(kind='discrete',reduce_loss=False).loss_function
    model.compile(loss=loss, optimizer=adam(lr=.01,clipvalue=0.5),sample_weight_mode='temporal')
    return model
```

Figure 2.1: Structure definition

In this particular case, the *adam optimizer* is used, with a learning rate of 0.01 (see section 2.3.2).

```
K.set_value(model.optimizer.lr, 0.01)
model.fit(x_train, y_train,
          epochs=100,
          batch_size=100,
          verbose=1,
          validation_data=(x_test, y_test,sample_weights_test),
          sample_weight = sample_weights_train,
          callbacks=[nanterminator,history,weightwatcher,reduce_lr])

plt.plot(history.history['loss'],    label='training')
plt.plot(history.history['val_loss'],label='validation')
plt.legend()
plt.show()
weightwatcher.plot()
```

Figure 2.2: Model implementation

Once the model structure is defined it comes the moment of the implementation. This step is coded as in figure 2.2. The first arguments to fit into the model reffer to the training dataset, divided into input and output values, which must be of a given shape (see section 3.1). Then, given the chosen optimizer, the number of epochs and the batch size must be set. The verbose function shows how the model is progressing while the iterations are going forward. Finally, some graphics are plotted in order

to visually check how the main variables behaved through the process. The first one is shown in figure 3.3; the evaluations of the loss-function give a useful idea of when overfitting starts and can help us to decide how any epochs are necessary to obtain the best solution.

2.3.2 Gradient descent optimization algorithms

Three optimizers were tested while the experiments so they may appear in future repositories about implementations of the method as well as others. This is the motivation why Stochastic gradient descent (SGD), a variant of gradient descent, is exposed in this section.

Given a model loss function $J(\theta)$ parameterized by its parameters $\theta \in \mathbb{R}^d$, which are updated in the opposite direction of the gradient of the objective function through backpropagation with a learning rate α. SGD performs parameter update for each training example $(x^{(i)}, y^{(i)})$:

$$\theta = \theta - \alpha \cdot \nabla_\theta J(x^{(i)}, y^{(i)}; \theta)$$

This method performs frequent updates that imply a high variance, which cause the objective function to fluctuate heavily at each iteration. This property enables it to jump to new points that may lead faster to a better local minimum but this may also complicate convergence to the exact minimum as it will continue overshooting. Anyway, it has been proven that with a slow decrease of the learning rate, SGD shows the same convergence behaviour as batch gradient descent. In order to find better performances of this optimizer, many gradient descent optimization algorithms have been created lately, most of them adapting the learning rate to the parameters depending on their occurring frequency and relevance for the loss function.

The tested optimizers when coding were **Adam**, **Nadam** and **RMSprop**. For specific documentation about these optimizers check [5]. As they are based on SGD, we must specify the batch size (how many training examples are used in a batch) and the number of epochs (number of iterations going through all the data) we want to use in the implementation.

Chapter 3

Experiments

Because of the lack of real data, this part of the project was basically about testing how already existing experiments were working in order to understand how to implement it in the case we had an interesting dataset to analyse. It was also a tool to understand how little modifications may change their results; to see how layer dimensions and amount of training data can modify a network performance.

Moreover, since a part of this project objective was to find how mathematics knowledge can be applied in business problems, two different scenarios are proposed. Both of them can be solved with WTTE-RNN although data should be treated in a slightly different way and also the cost function varies.

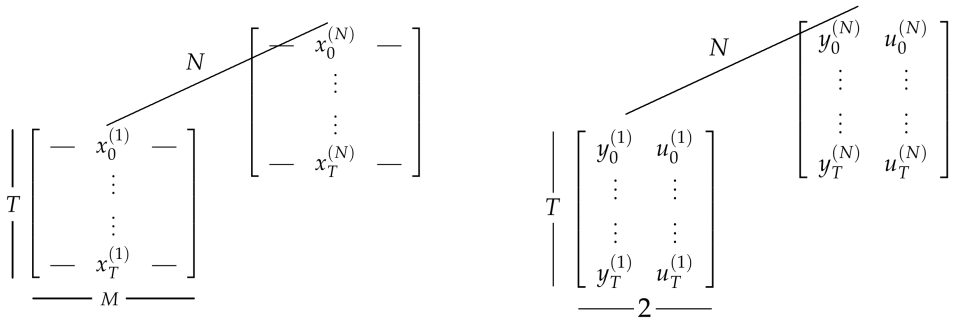
The basic tool to program neural networks in Python is the Keras API together with Tensorflow backend. It was developed to provide an easy and efficient way to implement any neural network architecture.

3.1 Data Structure

Since Recurrent Neural Networks get sequences as input, objects called *tensors* are used. One can think about them as matrices of several dimensions. To prepare the data so the model can learn from it, assuming it has

- N sequences ($n \in \{1, 2, ..., N\}$ sequence number),
- M features,
- T timesteps,

we must adapt it in a tensor of shape $(N \times T \times M)$. In some cases, M may be $M + 1$ (see why in the next section).



(a) Input data

(b) TTE and not-censored indicator

Figure 3.1: Training data shapes

The odds are that sequences have varying length. To solve that, T is fixed according to some criteria (like the longest length of all sequences or a reasonable time lapse we think is enough for the prediction). Sequences that are shorter than T, say their length is T_i, will be filled with a *mask value* in their last $T - T_i$ timesteps. Keras already handles this; a mask value is given to a layer that automatically skips them.

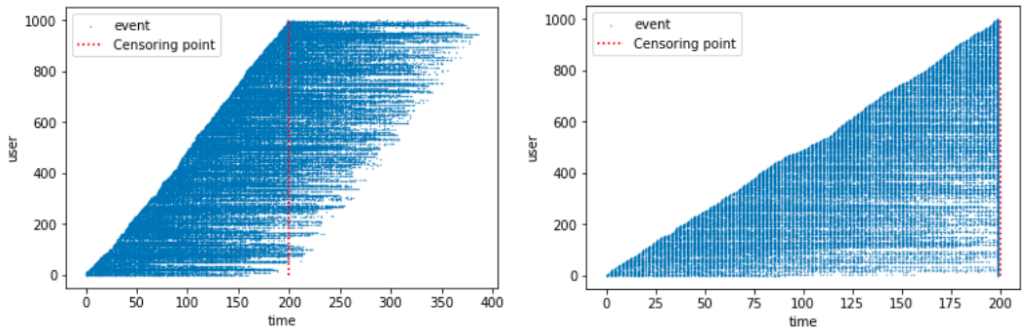
3.1.1 Non-contractual businesses: recurrent events structure

Ecommerce business are the perfect example for this type of problem and are specially concerned about censoring. Egil Martinsson generated a dataset of random sequences of varying length which could be related to an Ecommerce database (ref). It has one single feature but the implementation would be more or less the same in the case we had M features.

Suppose the events we are referring to are purchases. These data assumes we only have information when the customer is buying something; in the case that it wasn't like that, a good option would be adding an additional feature to the input vector with 1's or 0's if there was a purchase or not, respectively. This would give valuable additional information to the neural network, since other features may have the same value while the information they give is not be the same (e.g. clicks on products before a purchase or before a log-out without purchase).

To generate this kind of data and prepare it for the network, a python package was created by Egil Martinsson so all the steps we are describing below have an easy implementation that is available in [17]:

1. Fix the maximum number of sequences, maximum sequence length and latest time a sequence can start (1000, 200 and 200 respectively in this case).
   This sequences are already generated taking into account that the time variable is discrete. In case the data would be expressed in continuous time, the discrete time unit should be fixed so the data could be transformed. Moreover,



(a) Initially generated data

(b) Censored data

Figure 3.2: Censoring simulation. Each blue dot represents an event taking place. (Source: [17])

   this dataset has information for when there's an event, what means that the timesteps in between will be filled with $0's$ since the neural network considers each input step as one time unit going forward.
   To input all the information to the network, there will be a feature being 1 if there was an event or this filled 0's when there was not. It is a way of showing that the real feature value equals 0 because of that, considering that it could also be 0 when a purchase is made.

2. Simulate censoring. Assume the information after time T is 'the future'. This indicates in which part of the log-likelihood equation the data affects. The censoring point is represented in 3.2 as the vertical dot line.

3. Prepare the data so it can fit as input in the neural network. This means storing it as explained in the beginning of this chapter, according to what figure 3.1 shows. In one hand, those sequences shorter than 200 will be filled with the appropriate mask value so the network can skip those steps that are empty. In the other hand, because of how data is generated, the times between events need to be calculated so they can be used for the training and validation process.

Once these steps are completed, the data is ready so we can try little modifications until we find the specific structure that gives better results.

3.1.2 Contractual businesses: time to failure prediction

The second scenario appeals to a different customer database type. This case could belong to a bank's clients database, in the way that they really know if their customers are churned or not, for example when they cancel their account. Notice that now the data is not censored and what the algorithm will try to predict is the time until a customer decides to leave.

There is an available dataset about jet engines that was used to check if the model was able to predict times for not censored data. The loss function changes in this case because we could get rid of the fraction of the likelihood function that refers to censored observations. Although this dataset couldn't naturally be associated to a client database because of its values, its shape and how the author combines it with the model in order to find the best approximation shows how useful it could be in the case we had real data.

How the data is shaped is a key aspect in this kind of problem. While one has to take into account that it should fit in the shapes shown in figure 3.1, a set of training examples that contains N customers with sequences of length T can be partitioned in T different sequences having lengths $1, 2, ..., T$ for each client; this is used with a really interesting objective. Knowing the time to event at each step, the number of training examples can go from N to $N \times T$, what offers more opportunities of reaching a good approximation because you don't tell the algorithm that those subsequences belong to the same customer; the network will try to make predictions based on the cumulative information from 1 to T timesteps. This was altered by modifying Gianmario Spacagna's implementation, so the next lines are destined to describe the dataset and the two ways of treating it.

Dataset characteristics:

- 100 engines in the training set and 100 engines in the test set. $x_i^{(j)}$ is the input vector for engine j at timestep i.
- Set J of 24 features for each engine and $\sigma_i \in J$ the features $i \in \{1, 2, .., 24\}$. Given this notation, the structure of an input vector for any engine and any timestep is $x = (\sigma_1, ..., \sigma_{24})$.
- Sequences have varying length.
- The last information of each engine in the training set means the failure of itself at the next timestep.

Dataset preparation by Gianmaria Spacagna:

1. Dismiss constant features. After this step, $J = \{\sigma_i : i \in \{1, 2, .., 17\}\}$.

2. Normalize the other features such that $\sigma_i \in [-1, 1]$ $\forall i \in \{1, 2, .., 17\}$ depending on the minimum and maximum value they originally take.

3. Partition of the sequences. Assume a lookback period of 100 timesteps at most. Then, depending on each sequence length, the first dimension of the input tensor will be defined:

   Assume an input sequence x_n with T_n timesteps

   - If $T_n = 100$, 100 matrices M_i $i \in \{1, .., 100\}$ are filled. M_i is filled with x_i of the first i timesteps in the corresponding rows and with a mask value in the remaining $100 - i$ rows. In the last step of the preparation, the full matrix is filled with x_i $\forall i \in \{1, ..., 100\}$.
   - If $T_n < 100$, T_n matrices M_i $i \in \{1, .., T_n\}$ are filled. M_i is filled in the same way as before only with the difference that the process ends earlier; there won't be a matrix whose rows will be completely full of feature values.
   - If $T_n > 100$, T_n matrices M_i $i \in \{1, .., T_n\}$ are filled. This time, note that every M_j with $j > T_n$ will be completely full. Moreover, they won't contain the full sequence of feature states but the last 100 timesteps from timestep $T_n - j$.

   Note that each matrix M_t represents the variable $x_{0:t}$ defined in section 2.3.

4. Build T_n 2-d vectors containing the TTE and the censoring indicator for each subsequence, starting from T_n for the step-1 sequence and decreasing 1 by 1 for the following.

   Note that because of this way of storing the data, a many-to-one RNN is enough (it means just a tiny modification when implementing it in Python). This is why the real output values must be stored in a vector for each sequence instead of in matrix. Moreover, as data is not censored, the respective indicator will always be 1.

Modifications in data preparation:

1. Constant features are not dismissed. Since the network will choose which ones are valuable, it should not pay attention to them and we can omit this step. The others are also standardized satisfying $\sigma_i \in [-1, 1]$.

2. The last 200 steps of each sequence are put in a 200×24 matrix so this time there is only one matrix per sequence that dismisses the first $T_n - 200$ timesteps. This was implemented in order to check how the performance of the RNN was affected without creating all the possible subsequences, so using all the past information for predicting. Moreover, in this case the RNN is many-to-many type, so the output tensor should also be different from the other case.

3. The output values are stored in a 200×2 tensor. The censoring indicator is again 1 for every case. The y_t value equals 1 for the last step and increases while stepping upwards in the matrix.

4. Sequences shorter than 200 are filled with the suitable mask value in the rows corresponding to the remaining steps.

3.2 Output interpretation

If any data scientist would want to apply this method in a company, as well as the duty of clarifying which kind of data is required, in practice the most important part is the interpretation of the results so the people in charge can understand it and apply the measures they believe that are necessary. One can not find a solution with an acceptable final loss-function value and just give its absolute value and the parameters of a Weibull distribution. Because of that, the goal of this section is to interpret these magnitudes.

First of all, at the moment the iterations of the algorithm finish, there appears a plot where we can see how the values of the loss-function progress (figure 3.3), evaluated on the test and training set. This plot is mainly useful to detect overfitting and decide how many iterations are necessary to obtain the better results.

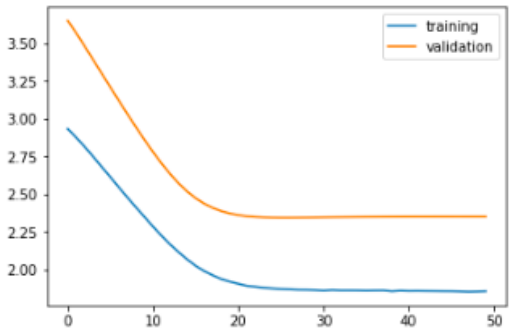


Figure 3.3: Solution for the implemented version of case 3.1.1

3.2.1 Weibull distribution

The assumption that the loss function value could be a sign of how well the model works is usually false. It is undeniable that it is a great tool to check if the model is learning in the appropriate way or if there is overfitting. Anyway, it does not have further utility since the metric that it represents is hardly applicable to business cases. Considering that the Weibull was chosen because of its unimodal density function and how its parameters describe the variable behaviour among other reasons, these magnitudes are going to be the main key concepts for the results interpretation.

Parameters

Once the network finishes the training process, when some input data enters in the structure we will know which parameters are assigned to its estimated TTE distribution. For this reason it is essential to understand what behaviour they determine. It was mentioned in 1.4.5 that the Weibull parameters α and β are also known as *scale* and *shape* parameters respectively. Let us specify why this nomenclature is used:

- The scale parameter α: it determines how wide the interval over which the density function is distributed is. As shown in figure 3.4, the bigger α the wider distribution.

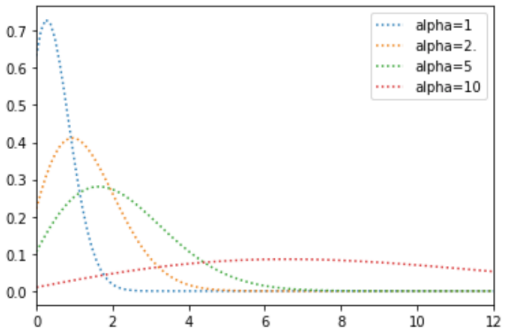


Figure 3.4: PMF given several values of α and $\beta = 2$ (evaluated on continuous range)

  Thus if the algorithm is performing well, for greater TTE's greater α's should be given as result.

- The shape parameter β: given a fixed value of α, the values β can take will modify the shape of the density function depending on:

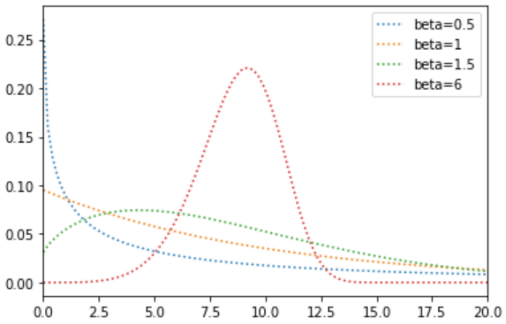


Figure 3.5: PMF given several values of β and $\alpha = 10$ (evaluated on continuous range)

- $\beta < 1$. This situation is related with a decreasing rate of event thus, i.e. the probability of experiencing an event decreases over time and the mode takes it highest value at $t = 0$; this argument is deeply analyzed in the next subsection.
- $\beta = 1$. In this case, the rate of event will be constant, so it would not be a really informative variable.
- $\beta > 1$. At some point, the rate of event will increase and there will be an specific time around which the density function peaks.

In this case, the relation between both parameters' magnitudes will also define how much gathered the area of the function is. Given α, for nearer magnitudes of β to that value we could define a smaller interval of TTE's of high probability. This is an interesting point since in a business case this could be interpreted as less risk.

Mode and percentiles

This analysis of the output is completely related with the parameter results and what was mentioned above. Since the mode is the moment with highest probability of experiencing an event, it will be a great magnitude for interpreting the results. The absolute value of its probability is highly conditioned by the distribution parameters, so we must find another metric to try to ensure an interval that maximizes the chance of getting a right guess.

The graphics in figure 3.6 belong to the Weibull representations in figure 3.5 for $\beta = 0.5, 1.5$ and 6 respectively. As we can appreciate, the represented statistics interpretation may vary depending on the shape of the distribution. We can always state that the mode corresponds to the TTE of higher probability so it is a good

option to consider it as the value to take risks around to. An interval around which the probability is still relatively high can be the percentiles when the distribution is significantly S-shaped like in the first case of figure 3.6. In any case, when $\beta < 1$ the only measure that we can consider representative is the mode, since the percentiles can go much further than the highest probability TTE. This is exposed in 3.2.2 since this issue appears when we distinguish between the two different kinds of businesses.

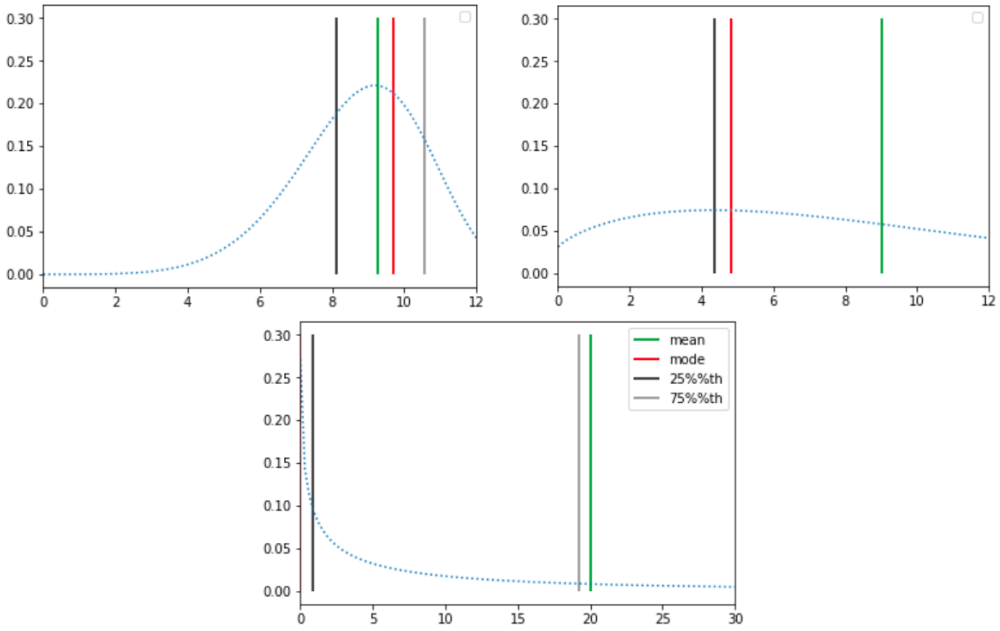


Figure 3.6: Statistics with different parameters of the Weibull

3.2.2 Results' summary

As mentioned before, the big difference between the two distinguished business models is the recurrence or not of events. The pictured results correspond to sequences of the test set that were picked randomly; the aim of this section is to basically dig deeply on how the concepts mentioned in 3.2 should be put in practice.

In the first case, we must emphasize on the fact that while sequences may have 100 timesteps length the recurrent TTE's don't go further than 10, what will help to understand figure 3.7: The rate of event is always decreasing in this case while time goes by since $\beta < 1$ in every case. This is because the generated events were spread in tiny timebreaks so, given the shown shapes a Weibull distribution can take, the mean values will be much higher than 0 while the mode will always stay there. If we had a dataset we could relate with the same kind of behaviour, because of what we

just exposed, the best statistic to take into account would be the mode. Depending on the what each time unit means, percentiles may be a magnitude that could help when taking risks in order to fix a time interval that the company could assume.

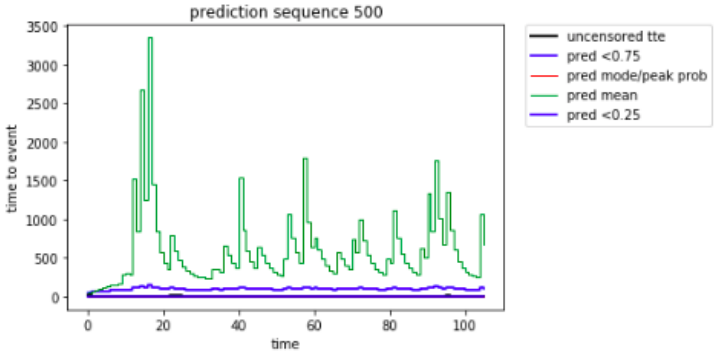


Figure 3.7: Resulting statistics of random sequence of the generated dataset (Source: [17])

In the other hand, when we talk about contractual businesses the TTE will always be decreasing since it is unique and nearer while time goes forward. The plots of this data are much more expressive as it is shown in figure 3.8

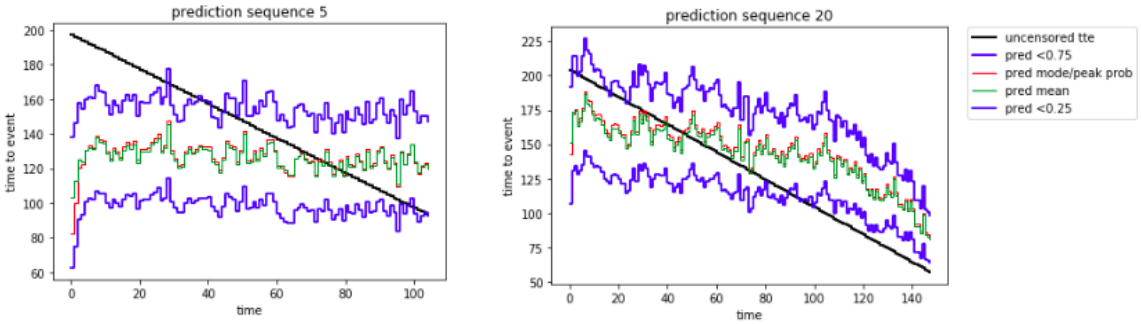


Figure 3.8: Predictions on the Jet Engine database applying the modifications explained in 3.1.2 to Gianmario Spacagna's implementation

In the prediction of the 5-th sequence we can appreciate that the model is not performing well. By the way, the mode and mean obtained for sequence 20 could be accepted as the best indicator. Note that these two statistics tend to be similar in this cases; this may be related with S-shaped distributions where the peak is more or less centered. Anyway, the natural process could be checking the possible shapes of the distribution taking into account the output values of the network; that will always be helpful to choose which statistics are more representative in each case.

Chapter 4

Conclusions

Our attention has been focused in the prediction of customer churn as a time to event approach. Without any previous knowledge of which methods could be applied to find a solution, Egil Martinsson's suggestion, summarized in [15], seemed interesting since he mentioned the statistical concept of Survival Analysis and the Deep Learning architectures known as Neural Networks as interesting tools to deal with it.
In this work we have been mainly focused on the documentation about what Deep Learning is among with its algorithms' characteristics, focusing on Recurrent Neural Networks. Moreover, we learned how to interpret statistical variables of Survival analysis, taking them into account as a resource to create an effective model that predicts the time between events.

With regard to Survival Analysis, it has allowed us to better assimilate and find applicability to concepts used during a Mathematics bachelor, more specifically in the field of statistics. Additionally, given that it was initially applied in the medicine field, the adjustment of its models to the business area contributes much of the value of the studied model since, together with the choice of the Weibull as the distribution to predict, it provides flexibility to the solutions.
Regarding Neural Networks, despite my interest in them, I started from scratch in this topic. This part of the learning was the most difficult for me due to many basic concepts I never worked with before as well as with the Python language and the algorithms implementation with this type of coding.

In the future, it would be interesting to dig deeply into several sections of these project since they have wide applicability potential in the real world. Additionally, experimenting with real data could provide me with better analytical and programming skills in order to find solutions for common problems in many business.

Bibliography

[1] Batten, Dayne. *Demo Weibull Time-to-event Recurrent Neural Network in Keras* [en línia] (2017) [consulta: abril 2018] `https://github.com/daynebatten/keras-wtte-rnn`

[2] Cybenko, G. Approximation by superpositions of a sigmoidal function. A:*Math. Control Signal Systems*, 1989, 2: 303. ISSN: 1435-568X.

[3] Davinson-Pion, Cam. *How Sopify Merchants can Measure Retention* [en línia] (2017). [consulta: juny 2018] `https://shopifyengineering.myshopify.com/blogs/engineering/how-shopify-merchants-can-measure-retention`

[4] Despa, Simona. *What is Survival Analysis?*[en línia]. [consulta: abril 2018] `https://www.cscu.cornell.edu/news/statnews/stnews78.pdf`

[5] Epelbaum, Thomas. *Deep learning: Technical introduction* [en línia] (2017) [consulta: juny 2018] `https://arxiv.org/pdf/1709.01412v2.pdf`

[6] Fei-Fei Li & Justin Johnson & Serena Yeung. *Recurrent Neural Networks* [en línia] (2018) Course c231n, Stanford University. [consulta: març, abril i juny de 2018] `http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf`

[7] Fei-Fei Li & Justin Johnson & Serena Yeung. *Recurrent Neural Networks* [en línia] (2018) Course c231n, Stanford University. [consulta: març, abril i juny de 2018] `https://www.youtube.com/watch?v=6niqTuYFZLQ`

[8] FW Scholz and Boeing Phantom Works. *Maximum likelihood estimation for type I censored Weibull data including covariates.* [en línia] (1996) `http://faculty.washington.edu/fscholz/DATAFILES498B2008/ISSTECH-96-022.pdf`

[9] Goodfellow, Et Al. and Yoshua Bengio and Aaron Courville. Deep Learning [en línia] (2016) [consulta: maig 2018] `http://www.deeplearningbook.org`

[10] Gunes Baydin, Atilim and Barak A. Pearlmutter and Alexey Andreyevich Radul and Jeffrey Mark Siskind. *Automatic Differentiation in Machine Learning* [en línia] (2018), Journal of Machine Learning Research. [consulta: juny 2018] http://jmlr.org/papers/v18/17-468.html

[11] Hornik, K.; Stinchcombe, M.; White, H. Multilayer feedforward networks are universal approximators. A:*Neural Networks*. 2, p. 359-366, 1989.

[12] Keras Documentation. *Callbacks* [en línia] [consulta: abril 2018] https://keras.io/callbacks/

[13] Manning, Chris & Socher, Richard. *Recurrent Neural Networks and Language Models* [en línia] (2017). Course cs224n, Stanford University. [consulta: abril 2018] https://www.youtube.com/watch?v=KeqepPKrY8&t=1094s

[14] Martinsson, Egil. *WTTE-RNN Weibull Time To Event Recurrent Neural Network* [en línia] (Gothenburg, Sweden, 2016) [consulta: febrer a juny de 2018] https://ragulpr.github.io/assets/draft_master_thesis_martinsson_egil_wtte_rnn_2016.pdf

[15] Martinsson, Egil. *WTTE-RNN - Less hacky churn prediction* [en línia] (2016) [consulta: febrer i març de 2018] https://ragulpr.github.io/2016/12/22/WTTE-RNN-Hackless-churn-modeling/

[16] Martinsson, Egil. *WTTE-RNN (Python Implementation / API)* [en línia] (2016) [consulta: febrer a maig de 2018] https://github.com/ragulpr/wtte-rnn/tree/master/python

[17] Martinsson, Egil. *Data-pipeline simple example.* [en línia] (2017) [consulta: febrer a juny 2018] https://github.com/ragulpr/wtte-rnn-examples/blob/master/examples/hello-world-datapipeline.ipynb

[18] Ng, Andrew. Coursera. *Machine Learning* [en línia]. Stanford University [consulta: març i abril de 2018] https://www.coursera.org/learn/machine-learning/home/welcome

[19] Olah, Christopher. Colah's Blog. *Understanding LSTM Networks* [en línia]. [consulta: maig i juny 2018] http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[20] Rodríguez, G. (2007). *Lecture Notes on Generalized Linear Models.*[en línia][consulta: abril 2018] http://data.princeton.edu/wws509/notes/

[21] Spacagna, Gianmario. *Deep Time-to-Failure.* [en línia] (2018) [consulta: febrer a juny 2018] https://github.com/gm-spacagna/deep-ttf/blob/master/notebooks/Keras-WTT-RNN%20Engine%20failure.ipynb

[22] UFLDL Tutorial, Stanford University. *Multi-Layer Neural Network* [en línia]. [consulta: juny 2018] http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/

Appendix

In order to implement the following code, one must first install the *wtte-rnn* package as it is suggested in [16]. Everything in this section is inspired in the implementations you can find in [17] and [21].

In the following pages you can see the code that belongs to the generation of the distributions plotted in section 3.2.1 with the title **Weibull plots**. Later on, in **Proves Jet Engine** there is the implementation whose results generated the plots in figure 3.8. It contains, step by step, the reading of the data and its preprocessing along with the implementation of what is detailed in 2.3.1 and 3.1.2.

The Jet Engine datasets are taken from [1]. You can find the refference to this path to the training and test datasets mentioned in the code as *path*, that should be replaced by *https://raw.githubusercontent.com/daynebatten/keras-wtte-rnn/master*.

Weibull plots

In [ ]:
```
%matplotlib inline
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from six.moves import xrange

import numpy as np
import wtte.weibull
import matplotlib.pyplot as plt
```

In [ ]:
```
np.random.seed(1)

beta  = 2.
y = np.linspace(0,12,100)
plt.plot(y,wtte.weibull.pmf(t=y,a=1.,b=beta),':',label = 'alpha=1')
plt.plot(y,wtte.weibull.pmf(t=y,a=2.,b=beta),':',label = 'alpha=2.')
plt.plot(y,wtte.weibull.pmf(t=y,a=3.,b=beta),':',label = 'alpha=5')
plt.plot(y,wtte.weibull.pmf(t=y,a=10.,b=beta),':',label = 'alpha=10')
plt.legend()
plt.xlim(0,y.max())
plt.title('Discrete Weibull distribution. beta = 2')
plt.show()
```

In [ ]:
```
alpha  = 10.
y = np.linspace(0,20,100)
plt.plot(y,wtte.weibull.pmf(t=y,a=alpha,b=0.5),':',label = 'beta=0.5')
plt.plot(y,wtte.weibull.pmf(t=y,a=alpha,b=1.),':',label = 'beta=1')
plt.plot(y,wtte.weibull.pmf(t=y,a=alpha,b=1.5),':',label = 'beta=1.5')
plt.plot(y,wtte.weibull.pmf(t=y,a=alpha,b=6.),':',label = 'beta=6')
plt.legend()
plt.xlim(0,y.max())
plt.title('Discrete Weibull distribution. alpha = 10')
plt.show()
```

In [ ]:
```
alpha = 10.
b1=0.5
b2=1.5
b3=6.
```

```
y = np.linspace(0,30,100)
plt.plot(y,wtte.weibull.pmf(t=y,a=alpha,b=b1),':')
plt.vlines(x=wtte.weibull.mean(a=alpha,b=b1),ymin=0,ymax=.3,color='green',
           label = 'mean')
plt.vlines(x=wtte.weibull.mode(a=alpha,b=b1),ymin=0,ymax=.3,color='red',
           label = 'mode')
plt.vlines(x=wtte.weibull.quantiles(a=alpha,b=b1,p=0.25),ymin=0,ymax=.3,
           color='black',label = '25%%th')
plt.vlines(x=wtte.weibull.quantiles(a=alpha,b=b1,p=0.75),ymin=0,ymax=.3,
           color='grey',label = '75%%th')

plt.legend()
plt.xlim(0,y.max())
plt.show()
```

In [ ]:
```
plt.plot(y,wtte.weibull.pmf(t=y,a=alpha,b=b2),':')
plt.vlines(x=wtte.weibull.mean(a=alpha,b=b2),ymin=0,ymax=.3,color='green')
plt.vlines(x=wtte.weibull.mode(a=alpha,b=b2),ymin=0,ymax=.3,color='red')
plt.vlines(x=wtte.weibull.quantiles(a=alpha,b=b2,p=0.25),ymin=0,ymax=.3,
           color='black')
plt.vlines(x=wtte.weibull.quantiles(a=alpha,b=b2,p=0.75),ymin=0,ymax=.3,
           color='grey')
plt.legend()
plt.xlim(0,y.max())
plt.show()
```

In [ ]:
```
plt.plot(y,wtte.weibull.pmf(t=y,a=alpha,b=b3),':')
plt.vlines(x=wtte.weibull.mean(a=alpha,b=b3),ymin=0,ymax=.3,color='green')
plt.vlines(x=wtte.weibull.mode(a=alpha,b=b3),ymin=0,ymax=.3,color='red')
plt.vlines(x=wtte.weibull.quantiles(a=alpha,b=b3,p=0.25),ymin=0,ymax=.3,
           color='black')
plt.vlines(x=wtte.weibull.quantiles(a=alpha,b=b3,p=0.75),ymin=0,ymax=.3,
           color='grey')

plt.legend()
plt.xlim(0,y.max())
plt.show()
```

Proves Jet Engine

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from six.moves import xrange

import matplotlib.pyplot as plt
import keras
from keras import backend as K
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import GRU
from keras.layers import Activation
from keras.layers import Masking
from keras.layers import Lambda
from keras.layers import BatchNormalization
from keras.layers.wrappers import TimeDistributed

from keras.optimizers import RMSprop,adam
from keras.callbacks import History
from keras import callbacks

import wtte.weibull as weibull
import wtte.wtte as wtte
from wtte.wtte import WeightWatcher

from sklearn.preprocessing import normalize

np.random.seed(2)
pd.set_option("display.max_rows",1000)
```

In [ ]:
```
id_col = 'unit_number'
time_col = 'time'
```

```
feature_cols = [ 'op_setting_1', 'op_setting_2', 'op_setting_3'] +
['sensor_measurement_{}'.format(x) for x in range(1,22)]

column_names = [id_col, time_col] + feature_cols
np.set_printoptions(suppress=True, threshold=10000)

train_orig = pd.read_csv('path/train.csv',
                         header=None, names=column_names)
test_x_orig = pd.read_csv('path/test_x.csv',
                          header=None, names=column_names)
test_y_orig = pd.read_csv('path/test_y.csv',
                          header=None, names=['T'])
```

In [ ]:
```
from sklearn.pipeline import Pipeline
from sklearn.feature_selection import VarianceThreshold
from sklearn.preprocessing import StandardScaler, MinMaxScaler

scaler=StandardScaler().fit(train_orig[feature_cols])

train = np.concatenate([train_orig[['unit_number', 'time']],
                        scaler.transform(train_orig[feature_cols])], axis=1)
test = np.concatenate([test_x_orig[['unit_number', 'time']],
                       scaler.transform(test_x_orig[feature_cols])], axis=1)
```

In [ ]:
```
# Make engine numbers and days zero-indexed, for everybody's sanity
train[:, 0:2] -= 1
test[:, 0:2] -= 1
test_time = np.squeeze(test_y_orig.as_matrix()).astype(float)
```

In [ ]:
```
#construir el tensor (tte, u_t)
import tqdm
from tqdm import tqdm

# TODO: replace using wtte data pipeline routine
def build_data(engine, time, x, max_time, is_test):
    # y[0] will be days remaining, y[1] will be not-censored indicator,
    #always 1 for this data
    n_engines=np.unique(engine).shape[0]
    d=x.shape[1]

    out_y = np.full([n_engines, max_time, 2], np.nan)
    out_x=[]

    for i in tqdm(range(n_engines)):
        out_x_i = np.full([1,max_time, d], np.nan)

        engine_x = x[engine == i]
        seq_len = engine_x.shape[0]
```

```
            if seq_len > max_time: #delete the first ndrop rows that make the
                #sequence exceed length max_time
                ndrop = seq_len-max_time
                out_x_i[:] = np.delete(engine_x, np.s_[0:ndrop],0)
                final_len = max_time
            elif seq_len < max_time: #mask last values if seq is shorter than
                #max_time
                for k in range(0, seq_len):
                    out_x_i[:,k] = engine_x[k]
                for k in range(seq_len, max_time):
                    out_x_i[:,k] = np.full(d, np.nan)
                final_len = seq_len
            else:
                out_x_i[:] = engine_x
                final_len = max_time


            if is_test:
                max_engine_time = time[i]
            else:
                # When did the engine fail? (Last day + 1 for train data)
                max_engine_time = 1.

            v = max_engine_time
            for j in range(final_len-1, -1, -1):
                out_y[i,j,:]=np.array([v, 1.])
                v += 1.

            out_x.append(out_x_i)

        out_x = np.concatenate(out_x)
        return out_x, out_y

    max_time = 200
```

In [ ]:
```
    train_x, train_y = build_data(engine=train[:, 0], time=train[:, 1],
                                  x=train[:, 2:], max_time=max_time, is_test=False)
    test_x, test_y = build_data(engine=test[:, 0], time=test_time,
                                x=test[:, 2:], max_time=max_time, is_test=True)
```

In [ ]:
```
    def nanmask_to_keras_mask(x,y,mask_value, tte_mask):
        x[:,:,:][np.isnan(x)] = mask_value
        y[:,:,0][np.isnan(y[:,:,0])] = tte_mask
        y[:,:,1][np.isnan(y[:,:,1])] = 0.95
        return x,y

    tte_mean_train = np.nanmean(train_y[:,:,0])
    mask_value = -1.3371337
```

```
mean_u = np.nanmean(train_y[:,:,1])

x_test,y_test   = nanmask_to_keras_mask(test_x,test_y,mask_value,tte_mean_train)
x_train,y_train = nanmask_to_keras_mask(train_x,train_y,mask_value,tte_mean_train)

# Initialization value for alpha-bias
init_alpha = -1.0/np.log(1.0-1.0/(tte_mean_train+1.0) )
init_alpha = init_alpha/mean_u
print('init_alpha: ',init_alpha,'mean uncensored train: ',mean_u)
```

In [ ]:
```
reduce_lr = callbacks.ReduceLROnPlateau(monitor='loss',
                                        factor  =0.5,
                                        patience=50,
                                        verbose=0,
                                        mode='auto',
                                        epsilon=0.0001,
                                        cooldown=0,
                                        min_lr=1e-8)

nanterminator = callbacks.TerminateOnNaN()
history = callbacks.History()
weightwatcher = WeightWatcher(per_batch =False,per_epoch= True)
n_features = x_train.shape[-1]

def base_model():
    model = Sequential()
    model.add(Masking(mask_value=mask_value,input_shape=(None, n_features)))
    model.add(GRU(15,activation='tanh',return_sequences=True))
    return model
def wtte_rnn():
    model = base_model()

    model.add(TimeDistributed(Dense(2)))
    model.add(Lambda(wtte.output_lambda,
                     arguments={"init_alpha":init_alpha,
                                "max_beta_value":4.0,
                                "alpha_kernel_scalefactor":0.5}))

    loss = wtte.loss(kind='discrete',reduce_loss=False).loss_function
    model.compile(loss=loss, optimizer=RMSprop(lr=.02,rho=0.5))
    return model
```

In [ ]:
```
model = wtte_rnn()
model.summary()

K.set_value(model.optimizer.lr, 0.001)
model.fit(x_train, y_train,
          epochs=100,
```

```
                batch_size=150,
                verbose=1,
                validation_data=(x_test, y_test),
                callbacks=[nanterminator,history,weightwatcher,reduce_lr])

        plt.plot(history.history['loss'],    label='training')
        plt.plot(history.history['val_loss'],label='validation')
        plt.legend()
        plt.show()
        weightwatcher.plot()
```

In [ ]:
```
        mask = np.copy(x_test)
        mask[mask==mask_value] = np.nan
        mask = mask*0
        predicted = model.predict(x_test)+mask[:,:,:1]

        alpha_flat = predicted[:,:,0][~np.isnan(predicted[:,:,0])].flatten()
        beta_flat  = predicted[:,:,1][~np.isnan(predicted[:,:,0])].flatten()
```

In [ ]:
```
        # Optional: add a margin of zeros at the end of the sequence to see what the
        # model does after the first events (if it identifies death by
        # pushing prediction to inf)

        mask = np.copy(x_test)
        mask[mask==mask_value] = np.nan
        mask = mask*0
        predicted = model.predict(x_test)+mask[:,:,:1]

        alpha_flat = predicted[:,:,0][~np.isnan(predicted[:,:,0])].flatten()
        beta_flat  = predicted[:,:,1][~np.isnan(predicted[:,:,0])].flatten()

        ## log-alpha typically makes more sense.
        from matplotlib.colors import LogNorm
        counts, xedges, yedges, _ = plt.hist2d(alpha_flat, beta_flat, bins=50,norm=LogNorm())
        plt.title('Predicted params : density')
        plt.xlim([alpha_flat.min(),alpha_flat.max()])
        plt.ylim([beta_flat.min(),beta_flat.max()])
        plt.xlabel(r'$\alpha$')
        plt.ylabel(r'$\beta$')
        plt.show()

        # Pick out prediction for one sequence
        batch_indx = 500
        print('Bad Example?:')
        for batch_indx in [5,20]:
            seq_len = (~np.isnan(predicted[batch_indx,:,0])).sum()
            a = predicted[batch_indx,:seq_len,0]+mask[batch_indx,:seq_len,0]
            b = predicted[batch_indx,:seq_len,1]+mask[batch_indx,:seq_len,0]
```

```
        t = np.array(xrange(len(a)))+mask[batch_indx,:seq_len,0]

        tte_actual   = y_test[batch_indx,:seq_len,0]+mask[batch_indx,:seq_len,0]

        ##### Prediction (Using weibull-quantities like quantiles etc)
        drawstyle = 'steps-post'
        plt.plot(t,tte_actual,label='uncensored tte',color='black',
                 linestyle='solid',linewidth=2,drawstyle=drawstyle)

        plt.plot(weibull.quantiles(a,b,0.75),color='blue',label='pred <0.75',
                 drawstyle=drawstyle)
        plt.plot(weibull.mode(a, b), color='red',linewidth=1,
                 label='pred mode/peak prob',drawstyle=drawstyle)
        plt.plot(weibull.mean(a, b), color='green',linewidth=1,
                 label='pred mean',drawstyle='steps-post')
        plt.plot(weibull.quantiles(a,b,0.25),color='blue',
                 label='pred <0.25',drawstyle=drawstyle)

        #     plt.ylim(0, 2*np.nanmax(tte_actual))
        plt.xlabel('time')
        plt.ylabel('time to event')
        plt.title('prediction sequence '+str(batch_indx),)
        plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
        plt.show()
        print('Good Example?:')

    print('alpha : mean   ',alpha_flat.mean())
    print('alpha : median ',np.median(alpha_flat))
    print('alpha :    min ',alpha_flat.min())
    print('alpha :    max ',alpha_flat.max())
    print('beta  : mean   ',beta_flat.mean())
    print('beta  : median ',np.median(beta_flat))
    print('beta  :    min ',beta_flat.min())
    print('beta  :    max ',beta_flat.max())
```

In [ ]:
```
    batch_indx=20
    seq_len = (~np.isnan(predicted[batch_indx,:,0])).sum()
    a = predicted[batch_indx,:seq_len,0]+mask[batch_indx,:seq_len,0]
    b = predicted[batch_indx,:seq_len,1]+mask[batch_indx,:seq_len,0]
    t = np.array(xrange(len(a)))+mask[batch_indx,:seq_len,0]

    print('alpha', a)
    print('beta', b)
    print('actual TTE   ',y_test[batch_indx,:seq_len,0]+
          mask[batch_indx,:seq_len,0])
    print('quantile 75%   ',weibull.quantiles(a,b,0.75))
    print('quantile 25%   ',weibull.quantiles(a,b,0.25))
    print('mode   ',weibull.mode(a,b))
```

```
        print('mean   ',weibull.mean(a,b))
```

In [ ]:
```
alpha = a[100]
beta  = b[100]
# Weibull is a simple distribution.
y = np.linspace(0,200,100)
# pdf  : (b / a) * np.power(t / a, b - 1) * np.exp(-np.power(t / a, b))
plt.plot(y,weibull.pdf(t=y,a=alpha,b=beta),label = 'pdf')
# And all the nice
plt.vlines(x=weibull.mean(a=alpha,b=beta),ymin=0,ymax=.3,
           color='green',label = 'mean')
plt.vlines(x=weibull.mode(a=alpha,b=beta),ymin=0,ymax=.3,
           color='red',label = 'mode')

plt.vlines(x=weibull.quantiles(a=alpha,b=beta,p=0.25),ymin=0,
           ymax=.3,color='black',label = '25%%th')
plt.vlines(x=weibull.quantiles(a=alpha,b=beta,p=0.75),ymin=0,
           ymax=.3,color='grey',label = '75%%th')
plt.vlines(x=weibull.quantiles(a=alpha,b=beta,p=0.99),ymin=0,
           ymax=.3,color='orange',label = '99.9%th')
plt.legend()
plt.xlim(0,y.max())
plt.title('Continuous Weibull distribution')
plt.show()
```