"Embedding layer and LocalGLMnet feature selection in actuarial setting"

Warnauts, Aymeric

ABSTRACT

Inspired by the structure of generalized linear models, Wuthirch M. and Richman R. (2021) propose a new network architecture that shares similar features as generalized linear models, but provides a superior predictive power benefiting from the representation learning. This architecture allows for variable selection of tabular data and for interpretation of the calibrated deep learning model. The purpose of this master thesis is to develop and apply this network to predict the default probability of personnal loans but also to generalize its behaviour in an actuarial setting with response features from the exponential dispersion family. Extensions such as the integration of embedding layers to manage text features and SHAP surrogate model will be discussed and applied. We will also take the liberty of criticising certain components of the architecture and making modifications, in particular for the variable selection part which is one of the cornerstones of this modern prediction tool.

CITE THIS VERSION

Institute of Statistics, Biostatistics, and Actuarial Sciences

Embedding layer and LocalGLMnet feature selection in actuarial setting

—

Aymeric Warnauts

A thesis submitted to the
Université catholique de Louvain
in partial fulfillment of the
requirements for the degree of
Master in Data Science

Thesis committee:

Prof. **Hainaut Donatien**	*UCLouvain/ISBA*	Supervisor
Prof. **Hafner Christian**	*UCLouvain/ISBA*	Reader
Prof. **Hafner Christian**	*UCLouvain/ISBA*	President

2022-2023

Acknowledgments

Completing this master's thesis has been an incredible journey, and I could not have done it without the support and guidance of numerous individuals who have contributed to its completion in various ways.

First and foremost, I am deeply grateful to my supervisor, *Donatien Hainaut*, whose expertise and guidance have been invaluable in shaping the direction of my research and providing constructive feedbacks throughout the process.

I would also like to extend my sincere appreciation to the faculty members of the *Institute of Statistics, Biostatistics, and Actuarial Sciences*, whose mentorship, encouragement, and knowledge have been critical in helping me develop my research skills and expertise.

Furthermore, I would like to express my heartfelt thanks to my colleagues and friends, who have offered their unwavering support, encouragement, and feedback throughout this journey. I am particularly grateful to *Gabriel Bailly* and *Nathan Nepper*, who have been my pillars of strength during the most challenging moments of this research.

Last but not least, I would like to acknowledge my family for their unconditional love, encouragement, and support throughout my academic journey. Their unwavering belief in me has been a constant source of inspiration and motivation, and I owe them a debt of gratitude that can never be repaid.

*

This dissertation has been prepared in partial fulfillment of the requirements for the master degree in data science (statistics orientation) delivered by the Université Catholique de Louvain (Louvain-la-Neuve, Belgium).

Table of Contents

List of Figures

List of Tables

Notations

a . . . . . Scalar.

$\mathbf{A}$ . . . . . Matrix.

$\boldsymbol{a}$ . . . . . Vector.

$\boldsymbol{X}$ . . . . . Matrix of p features for all n data points ($n \times p$), where each row represents a data point and each column represents a feature.

$\boldsymbol{X}^\top$ . . . . . Transposed of the design matrix ($p \times n$).

n . . . . . Number of observations (or data points).

$\mathbf{I}[.]$ . . . . . Indicator function of an event, a function that takes the value 1 if the event is true and 0 otherwise.

$\mathbf{P}[.]$ . . . . . Probability of an event, a measure of the likelihood of the event occurring.

$\mathbf{E}[.]$ . . . . . Expectation of a random feature, the average value or mean that the feature is expected to take.

$\text{Var}[.]$ . . . . . Variance of a random feature, a measure of how much the values of the feature vary around their mean.

y_i . . . . . Realisation of the random feature $Y_i, i = 1, \ldots, n$, i^{th} observed value of the response.

$\widehat{y}_i$ . . . . . Fitted value, predicted response for the i^{th} individual or observation.

$\bar{y}$ . . . . . Sample mean of the n observed responses $y_1, \ldots, y_n$.

$\boldsymbol{y}$ . . . . . Column vector of all n response values where all vectors are column vectors by convention. Each element of the vector represents a response value.

p . . . . . Number of features, also called covariates or predictors when they influence the response

x_{ij} . . . . . Value of the j^{th} feature for the i^{th} data point, $i = 1, \ldots, n$, $j = 1, \ldots, p$ and that is the realisation of the random variable $\boldsymbol{X}_{ij}$.

$\boldsymbol{x}_i$ . . . . . Column vector of the p features for the i^{th} data point.

$\mathcal{L}$ . . . . . Likelihood or loss function, depends on the context.

L . . . . . Log-likelihood function.

φ . . . . . Exposure-to-risk.

$\widehat{\theta}$ . . . . . Estimator or estimate of an unknow parameter θ where parameters are denoted by Greek letters.

$cos(.)$ . . . . . Cosines metric for similarity computation between vectors.

ϕ, ϕ_j . . . . . Dispersion parameter in *glm* or Shapley value for the j^{th} feature.

V_j . . . . . Feature importance metrics for the j^{th} feature.

$\sim$ . . . . . means "is distributed as".

$\approx$ . . . . . means "is approximately equal to" or "is approximately distributed as".

$\mathcal{I}$ . . . . . Fisher information, a measure of the amount of information that a random variable carries about an unknown parameter in a statistical model.

Chapter 1

Introduction

For this thesis we will be focus on the application of the *Wuthrich* and *Richman* network architecture which is a new way for predictions based on the well known architecture of generalized linear models. The neural network proposed by them allows for variable selection of *tabular data* which is not as easy with deep learning models due to non transparency of the feature engineering. We will try to apply the theoretical concepts introduced in the paper Richman and Wüthrich (2021) on the *lending club dataset*[1] that has been retrieved from the *Kaggle* website. Moreover, benefiting from the representation learning this new method will provide a better accuracy for prediction, we will summarize the results and compare them with performances obtained with other prediction tools. As this thesis is meant for broad readership we will take the precaution of introducing key concepts, such as the different ways of preprocess data's or statistical modelling to gain a better understanding of insurance data analytics. Moreover, recent research has been focusing on interpreting machine learning predictions in retrospect and as the implementation of the *LocalGLMnet* aims to own that kind of interpretation power that can serve as surrogate model to shed more light into many other deep learning models we will try to extend, link and compare the network architecture to some of popular algorithms. In addition, as the predictive power of this architecture will be discussed we will take advantage of it for features selection, interaction detection procedure and explore a combination with the well known *embedding layer* to manage categorical features with many levels. So first let's have a short introduction of why data analysis can be a powerful tool in actuarial setting before diving deep in the statistical modeling part of this thesis.

[1] https://www.kaggle.com/imsparsh/lending-club-loan-dataset-2007-2011

1.1 Usefulness of data analysis

The particularity is that instances as insurers and bankers are dealing with promises business and more precisely, in non-live insurance policy the insurer promises to pay for the future losses of the clients in exchange of an upfront premium and bankers lend money to individuals in exchange for a promise to repay with interest. Hence, determining the best price for the contract the insurance company offers will be one of their key objectives since the organization must assess some indicators when the economic risk is passed from the policyholder to the insurer. Yet, they must refine their technical pricing even more to avoid any adverse selection effects brought on by rivalry with commercial insurers. In fact, in practice, the majority of portfolios are diversified and combine clients with various risk levels and thus on average, some policyholders tend to record more expensive or more frequent claims. The composition of the portfolio affects the insurance company's financial performance in a heterogeneous portfolio with a uniform price list. That's why *risk classification* - in order to keep track of the gains and losses - is of main concern for actuaries, as beyond mutualization, imposing the same premium on all insurers encourages subventional solidarity, mechanism that commercial insurers usually want to avoid.

Precisely, one of the main goal of the data science application to insurance is to evaluate the combined *operating cost* for each policy which is used as a technical price and a benchmark for internal risk assessment. Due to the conflicting interests of technical and commercial premiums, which are both sources of insurance profits, we note a semi-systematic difference between them. Moreover, the amount the insurance provider should charge in order to be able to pay out all claims without suffering any loss or making a profit is known as the *pure premium*.

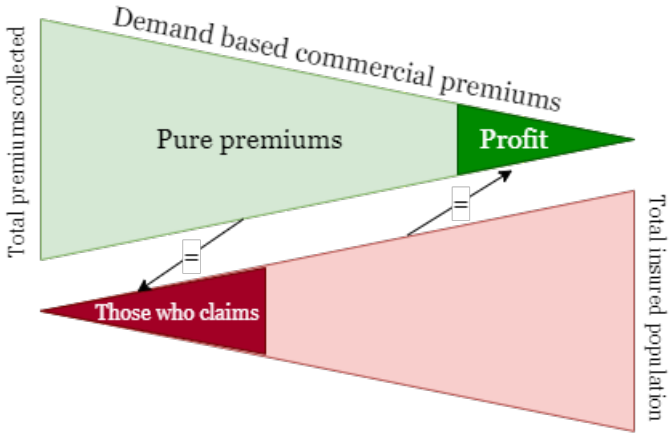


Figure 1.1: Premium transfers in insurance organisations

Additionally, the data analysis allows to extract useful information from large samples of customer represented by contracts with the company. Therefore, it seems convenient to recall

the law of large number that can help to understand how the averages in a portfolio are computed:

Theorem 1. *When $n \to \infty$ then:*

$$\bar{X}_n \xrightarrow{p} \mu$$

Such that $\mathbf{P}(lim_{n\to\infty}\bar{X}_n = \mu) = 1$

Illustration: Male drivers between the ages of 18 and 25 who purchase auto insurance may be considered to be "high risk" drivers. The Low of Large Numbers suggests that, when considered as a whole, that category of drivers is most likely to drive dangerously and inflict significant amounts of property and health damage. This is not to mean that all drivers in that category are linked to high risk level.

And thus, take Y_i reflecting the total benefits paid by the insurer in regard to individual policies during a period, as independent random variables with identical distribution and common moment generating function $Y_1, Y_2, \ldots, Y_n \sim ED(\theta, \phi/\varphi)$ and under mild technical conditions,

$$\bar{Y}_n = \frac{1}{n}\sum_{i=1}^{n} Y_i \sim ED(\theta, \phi/n\varphi)$$

can be seen as the average payment per policy in a portfolio of size n, where Y_i is also the total claim amount for policy i as conditionally to what has been said previously the pure premium match the expected claim amount.

The collection of large amount of data allows insurers to determine the risk profile of their clients through metrics as key ratio's. These ratio's are the result of response feature Y updated by an exposure φ which is a fundamental unit used by insurers to calculate how much risk is covered during a specific time period; as a result, it is often proportional to risk. In actuarial sciences the main focus will carry out such ratio's as the *loss ratio*, *claim severity*, *claim frequency* and for this thesis *loan default probability*. In fact, as regulators now impose insurance company which offer contracts to hold enough capital to fulfill the promises they sell, companies need to allocate premiums carefully. Precisely, expected losses may vary between policies and thus data analysis may help to prevent adverse selection, keep track of premium transfers made inside the portfolio as a result of using the commercial pricing list, and assess the value of the client by contrasting technical premiums with commercial ones. So, *individual analysis* - which leads to the identification of risky profiles - indicates how to allocate the total premium income among different policies. That's why it's very interesting to perform data analysis on individuals using

regression tools in order to estimate a well chosen response from a function of risk factors and parameters in order to determine what cause the claims. But we point out that the global insurance strategy is to perform *top-down pricing* which consists in predicting the portfolio premium in a first step before identifying risky profiles.

1.2 The need for supervised learning

As the total amount of information collected over policyholders is growing and that insurers owns more information than ever before to discriminate and build their tariff, richest and trustworthy databases become more and more available. We already mentioned the importance of risk classification based on *risk factors* but before going further it's fundamental to distinguish that kind of features with what's called the *rating factors*. While rating factors are the features insurers use to calculate the premium, risk factors refer to features that affect how intense the risk is. In a perfect world, all risk factors would serve as rating factors but in practice risk factors might be challenging for the insurer to quantify or not allowed to use.

Illustration: For instance, age is a significant risk factor in life insurance and is almost always taken into account when determining rating. Probably one of the biggest risk factors for auto insurance is driving speed. Yet, measuring this is challenging for insurers. As an alternative, driving speed is represented by rating factors. For instance, the insurance might anticipate that sports car owners or young mans will drive the fastest.

After the definitions of thess concepts, we introduce *supervised learning* that involves training a model using labeled data, meaning that the input variables and their corresponding outputs are already known. The goal of the model is to learn the relationship between the input variables and the output variables, so that it can make accurate predictions on new, unseen data. In the context of actuarial modeling, supervised learning techniques could be used to predict insurance claims or estimate the probability of a policyholder filing a claim, for example. The model would be trained on historical data with known outcomes, such as past claims, to make predictions on new data. Common supervised learning algorithms used in actuarial modeling include *regression analysis, decision trees, and neural networks.*

1.2.1 Generalized Linear Models

A Generalized Linear Model is defined by a specific distribution that is contained in the *Exponential family* together with a *link function* and a *linear predictor* as the link function $g(.)$ defines the

relationship between the linear predictor and the mean such that:

$$x \mapsto g(\mu) = g(\mu(x)) = \beta_0 + \sum_{j=1}^{p} \beta_j x_j \tag{1.1}$$

Furthermore, *GLMs* are an extension of linear regression models and are used when the distribution of the response variable is not normal, and/or the relationship between the response variable and the predictor variables is not linear. In fact, combining the systematic (linear score) and the random components (response distribution), the combination of dependent variable y was formalized using *GLMs* in such a way that the exponential family distribution has the following density function:

$$f_{Y_i}(y_i; \theta_i, \phi) = e^{\frac{y_i\theta_i - b(\theta_i)}{\phi/\varphi_i}} c(y_i, \phi, \varphi_i) \qquad \text{where } y_i \subseteq S \begin{cases} \mathbb{N} \\ \mathbb{R} \end{cases} \tag{1.2}$$

for the density function of y_i that represents the relative probability, or likelihood, to observe a certain value given the parameters of the distribution where ϕ is called the dispersion parameter and can take the form of the variance σ^2 if y is normally distributed. Moreover, by first normalizing with the link function and linearization with a transformation function $f(.)$ we obtain:

$$f\{g^{-1}(\mu)\} = \sum_{j=1}^{p} \beta_0 + \sum_{j=1}^{p} \beta_j x_j \tag{1.3}$$

with $(\beta_1, \ldots, \beta_p) \in \mathbb{R}^p$ and $\beta_0 \in R$.

Generalized Linear Models are usually estimated via the *Maximum Likelihood Estimation* as when the response feature has a normal distribution and the link function is the identity function, a *GLM* reduces to a linear regression model and it will give the same estimation than *OLS* back. In fact, with the data's considered as known and assuming a distribution, it allows us to estimate the population parameters by maximizing the likelihood of observing the data given the model parameters as with the first derivative of the log-likelihood with respect to our parameters we find the maximum of this function. The second derivative will help us to determine the shape of this function and thus confirm the previous assumption and as it's a measure of curve, we use it to compute the standard errors too.

Thus, as within the *Exponential distribution models* class, a family of probability distributions is uniquely characterized by its variance function, so if we want to use *GLM's* we only have to model the mean and the variance expressions.

First let's define the *EDM* moment generating function that is defined as the expected value of the exponential function raised to a multiple of the random variable:

$$\begin{aligned}\mathbf{E}(e^{tY}) &= \int e^{\frac{y(\theta+t\frac{\phi}{\varphi})-b(\theta)}{\frac{\phi}{\varphi}}} c(y,\phi,\varphi))\delta y \\ &= e^{\frac{b(\theta+t\frac{\phi}{\varphi}))-b(\theta))}{\frac{\phi}{\varphi}}} \cdot \int e^{\frac{y(\theta+t\frac{\phi}{\varphi})-b(\theta+t\frac{\phi}{\varphi})}{\frac{\phi}{\varphi}}} c(y,\phi,\varphi))\delta y\end{aligned} \tag{1.4}$$

and if the expectation is finite at least for $t \in R$ in a neighborhood of 0 and if the parameter space is open, $\theta + t\frac{\phi}{\varphi}$ is in the parameter space and the integrand is an *EDM*. Thus, as

$$\mathbf{E}(e^{tY}) = e^{\psi_i(t)} \tag{1.5}$$

where,

$$\psi_i(t) = \frac{b(\theta_i + t\frac{\phi}{\varphi_i}) - b(\theta_i)}{\frac{\phi}{\varphi_i}} \tag{1.6}$$

that is the cumulant generating function of an *EDM* Y_i, we can obtain the moments of Y_i by differentiating and setting $t = 0$:

$$\mathbf{E}(Y_i) = \psi_i^{'}(0) = b^{'}(\theta_i) \tag{1.7}$$

$$\mathrm{Var}(Y_i) = \psi_i^{''}(0) - (\psi_i^{'}(0))^2 = b^{''}(\theta_i)\frac{\phi}{\varphi_i} \tag{1.8}$$

Finally, we can show that we can express the canonical parameter θ_i in terms of the mean μ_i through the inverse link function:

$$\theta_i = b^{'-1}(\mu_i) \tag{1.9}$$

and the variance of Y_i is related to the mean through the variance function and the dispersion parameter:

$$\mathrm{Var}(\mu_i) := b^{''}(b^{'-1}(\mu_i)) \tag{1.10}$$

that is the variance function of the exponential family distribution and thus,

$$\mathrm{Var}(Y_i) = \mathrm{Var}(\mu_i)\frac{\phi}{\varphi_i} \tag{1.11}$$

As we want to test our model fit, once again we can use the likelihood and more precisely the

likelihood ratio such that

$$LR = -2(L(\beta_{H_0}) - L(\beta_{H_1})) \tag{1.12}$$

that compare the log-likelihood of the restricted model under $H_0 : \beta_i = 0\,, \forall i$. And so, the log-likelihood as a function of the parameter θ is:

$$L(\theta, \phi, y) = \frac{1}{\phi} \sum_i \varphi_i (y_i \theta_i - b(\theta_i)) + \sum_i c(y_i, \phi, \varphi_i) \tag{1.13}$$

and what's important to point out is that the ϕ doesn't influence the maximization of the expression with respect to θ. And with the demonstrated relationships $\mu_i = b^{'}(\theta_i)$ and $g(\mu_i) = \eta_i = \sum_j x_{i,j}\beta_j$, we can write the likelihood as a function of β , rather than θ. With the chain rule we obtain the following derivative expression:

$$\begin{aligned} \frac{\delta L}{\delta \beta_j} &= \sum_i \frac{\delta L}{\delta \theta_i} \frac{\delta \theta_i}{\delta \beta_j} = \frac{1}{\phi} \sum_i (\varphi_i y_i - \varphi_i b^{'}(\theta_i)) \frac{\delta \theta_i}{\delta \beta_j} \\ &= \frac{1}{\phi} \sum_i (\varphi_i y_i - \varphi_i b^{'}(\theta_i)) \frac{\delta \theta_i}{\delta \mu_i} \frac{\delta \mu_i}{\delta \eta_i} \frac{\delta \eta_i}{\delta \beta_j} \end{aligned} \tag{1.14}$$

so with $\mu_i = b^{'}(\theta_i)$ and$\frac{\delta \mu_i}{\delta \theta_i} = b^{''}(\theta_i) = \text{Var}(\mu_i)$ finally, $\frac{\delta \theta_i}{\delta \mu_i} = \frac{1}{\text{Var}(\mu_i)}$. Moreover, $\frac{\delta \mu_i}{\delta \eta_i} = (\frac{\delta \eta_i}{\delta \mu_i})^{-1} = \frac{1}{g^{'}(\mu_i)}$ and $\frac{\delta \eta_i}{\delta \beta_j} = x_{i,j}$ and so we finally get the **score function**:

$$\frac{\delta L}{\delta \beta_j} = \frac{1}{\phi} \sum_i \varphi_i \frac{(y_i - \mu_i)}{v(\mu_i) g^{'}(\mu_i)} x_{i,j} \tag{1.15}$$

for $j = 1, ..., p$ and where $\mu_i = g^{-1}(\eta_i) = g^{-1}(\sum_j x_{i,j}\beta_j)$.

Actuaries often use the deviance function along with other statistical measures of goodness of fit, such as the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC), to select the best model among a set of competing models. And the same logic applies to the deviance test as we compare the fit of a model with the saturated one that have one parameter for each observation such that:

$$D = -2(L(\beta_{H_0}) - L(\beta_{H_1}))$$

where the model is saturated under H_1. Thus as we reformulate, we compare the minimal model which contains the smallest set of terms that the problem allows, and the complete model in which all the Y's are different and match the data completely. Moreover, we can add two additional measures under the same assumption that are:

- The residual deviance = $L(\beta_{H_0})$

- The null-deviance = $L(\beta_{intercept}))$

that compared to each other indicates the increase in model fit from the null model.

Finally, if we let h denote the inverse of b', so that $\theta_i = h(\mu_i)$, with the definition of the *exponential distribution model* we get:

$$D = \frac{2}{\phi} \sum_{i=1}^{n} \varphi_i (y_i h(y_i) - b(h(y_i)) - y_i h(\widehat{\mu}_i) + b(h(\widehat{\mu}_i))) \tag{1.16}$$

The difficult problem of discussing how far transformations can produce both *linearity* and *normality* simultaneously now disappears because the models allow two different transformations to be used, one to induce the linearity of the systematic component and one to induce the desired distribution in the error component. Keeping in mind what we introduced with non-life insurance pricing/credit risk may suggests using linear regression but we seldom have a linear relation between the premium and rating factors. Even if it doesn't occur very often in practice, the assumption of *policy/loan independence* and *time independence* are fundamental for building such models.

Illustration: In fact, think to a collision between two cars insured by the same company and to a car driver that had an accident that may drive more carefully in the future.

So, let's denote Y_i the response for policy i or in time interval i respectively, then $Y_i, ..., Y_n$ are independent. In addition, the *homogeneity* of insurance policies groups is crucial, it's then reasonable to charge the same premium/default probability for all policies within the same cell, for the same duration and that's why we assume the independence of distributions.

1.2.2 Neural networks

In practice, neural networks are not used to approximate known functions, but they can be useful for building a model based on already collected data. Therefore, neural networks constitute a good way to train a regression model to determine an appropriate link between inputs and outputs of known data's. Moreover, one of the massive advantages is that neural networks are parsimonious, and thus with the same amount of data, they achieve a more accurate approximation of model parameters. During training, as the neural network holds differentiable functions, it performs gradient descent to find coefficients that match the data until it arrives at the optimal weights for the model. In fact, to expand on the topic of *retropropagation*, it is a widely used algorithm for training neural networks, which is based on *backpropagation*. The goal of retropropagation is to adjust the weights and biases in the neural network in order to minimize the difference

between the predicted output and the actual output. To achieve this goal, the algorithm first computes the output of the neural network for a given input and then, compares this output to the actual output and computes the error between the two. The error is then propagated backwards through the network, from the output layer to the input layer, and the weights and biases are adjusted accordingly.

Let's assume we have a neural network with d layers, and $w_{m,i,j}$ denotes the weight of the connection between neuron i in layer m and neuron j in layer $m+1$. Similarly, $b_{m,i}$ is the bias term of neuron i in layer m. The forward pass of the neural network takes an input x and computes the output $y^{(d:1)}(x)$ of the last layer. The loss function $\mathcal{L}(y^{(d:1)}(x), y)$ measures the error between the predicted output $y^{d:1}(x)$ and the actual output y.

During the backward pass, the algorithm calculates the gradients of the loss function with respect to the weights and biases of the neural network, by recursively applying the chain rule of differentiation. So, for each neuron i in the output layer d, we have:

$$\frac{\partial \mathcal{L}}{\partial y_{m,i}} = \frac{\partial \mathcal{L}(y^{d:1}(x), y)}{\partial y_{m,i}} \tag{1.17}$$

For each neuron i in layer m ($1 \leq m < d$), we can calculate the gradient of the loss function with respect to its output $y_{m,i}$ as follows:

$$\frac{\partial \mathcal{L}}{\partial y_{m,i}} = \sum_{j=1}^{q_{m+1}} \frac{\partial \mathcal{L}}{\partial y_{m+1,j}} w_{m+1,j,i} f'_{m,i}(y_{m,i}) \tag{1.18}$$

where q_m is the number of neurons in layer m, and $f'_{m,i}(y_{m,i})$ is the derivative of the activation function of neuron i in layer m evaluated at $y_{m,i}$.

Once we have calculated the gradients of the loss function with respect to the outputs of all neurons in the neural network, we can use them to calculate the gradients of the loss function with respect to the weights and biases of the network. For each weight $w_{m,i,j}$ and bias $b_{m,i}$, we can calculate their gradient as follows:

$$\frac{\partial \mathcal{L}}{\partial w_{m,i,j}} = z_{m-1,j} \frac{\partial \mathcal{L}}{\partial y_{m,i}} \tag{1.19}$$

$$\frac{\partial \mathcal{L}}{\partial b_{m,i}} = \frac{\partial \mathcal{L}}{\partial y_{m,i}} \tag{1.20}$$

where $z_{m-1,j}$ is the output of neuron j in layer $m-1$. Finally, we can update the weights and biases of the neural network using the calculated gradients and the learning rate α as follows:

$$w_{m,i,j} = w_{m,i,j} - \alpha \frac{\partial \mathcal{L}}{\partial w_{m,i,j}} \tag{1.21}$$

$$b_{m,i} = b_{m,i} - \alpha \frac{\partial \mathcal{L}}{\partial b_{m,i}} \tag{1.22}$$

Using the chain rule, we can compute $\frac{\partial \mathcal{L}}{\partial w_{m,i,j}}(W, X)$ where W is the weight matrix of the model and X the input matrix as:

$$\frac{\partial \mathcal{L}}{\partial w_{m,i,j}}(W, X) = \frac{\partial \mathcal{L}}{\partial y_{m,i}}(W, X) \frac{\partial y_{m,i}}{\partial w_{m,i,j}}(W, X) \tag{1.23}$$

where $\frac{\partial y_{m,i}}{\partial w_{m,i,j}}(W, X)$ is the derivative of the activation function with respect to the weight and the input matrix. And thus this term can be simplified as:

$$\frac{\partial y_{m,i}}{\partial w_{m,i,j}}(W, X) = z_{m-1,j} \tag{1.24}$$

since the output of neuron j in layer $m-1$ is the input to neuron i in layer m. Similarly, we can compute $\frac{\partial \mathcal{L}}{\partial b_{m,i}}(W, X)$ using the chain rule as:

$$\frac{\partial \mathcal{L}}{\partial b_{m,i}}(W, X) = \frac{\partial \mathcal{L}}{\partial y_{m,i}}(W, X) \frac{\partial y_{m,i}}{\partial b_{m,i}}(W, X) \tag{1.25}$$

Here, $\frac{\partial y_{m,i}}{\partial b_{m,i}}(W, X)$ is the derivative of the activation function with respect to the bias, which is 1. Therefore, we get:

$$\frac{\partial \mathcal{L}}{\partial b_{m,i}}(W, X) = \frac{\partial \mathcal{L}}{\partial y_{m,i}}(W, X) \tag{1.26}$$

Finally, to compute $\frac{\partial \mathcal{L}}{\partial y_{m,i}}$, we use the chain rule again. Specifically, we write:

$$\frac{\partial \mathcal{L}}{\partial y_{m,i}}(W, X) = \sum_{j=1}^{q_{m+1}} \frac{\partial \mathcal{L}}{\partial y_{m+1,j}}(W, X) \frac{\partial y_{m+1,j}}{\partial y_{m,i}}(W, X) \tag{1.27}$$

The term $\frac{\partial y_{m+1,j}}{\partial y_{m,i}}(W, X)$ is the derivative of the activation function in layer $m+1$ with respect to the output of neuron i in layer m. This term can be simplified as:

$$\frac{\partial y_{m+1,j}}{\partial y_{m,i}}(W, X) = w_{m+1,i,j} f'_{m,i}(y_{m,i}) \tag{1.28}$$

where once again $w_{m+1,i,j}$ is the weight connecting neuron i in layer m to neuron j in layer $m+1$, and $f'_{m,i}(y_{m,i})$ is the derivative of the activation function of neuron i in layer c evaluated at $y_{m,i}$. Combining these results, we can recursively compute the gradients of the loss function with respect to the weights and biases of the neural network using what we call *backpropagation*.

1.3 Growing concern about unsupervised learning

For actuarial modeling, unsupervised learning techniques could be used to identify segments of policyholders with similar characteristics, for example. The model would analyze data on policyholders, such as demographics and purchase history, to identify patterns and group them together. Common unsupervised learning algorithms used in actuarial modeling include *clustering, principal component analysis, anomaly detection*, we will try to apply these techniques to some parts of this project such as for our preprocesing. Actually, the insurance industry is undergoing a digital transformation as underwriters are collecting more data with the help of sophisticated machine learning algorithms for better risk management and customer-specific premium pricing. Moreover, *clustering* can also be used to identify emerging risks or trends within a portfolio of policies that is if a cluster of policyholders experiences a higher frequency of claims than expected, this could indicate the presence of a new risk factor that needs to be accounted for in future pricing and risk assessment.

Finally, as unlabelled data's are becoming common when extracting huge quantity of information from different distribution markets, unsupervised learning may be helpful because it can identify patterns within the data set without the need for pre-existing labels or categories. In what follows we will take advantage of powerful algorithms such as *k-means* that is a popular unsupervised machine learning algorithm used to group data points into k clusters based on their similarities and that will allow us to reduce the number of dimensions of a categorical feature for example. We will not redefine the step procedure of this method which consist ofiterative centroid assignments and then actualisation of the respective cluster means but we recall that the objective of *k-means* clustering is to minimize the within-cluster sum of squares, which is the sum of the squared distances between each data point and its assigned centroid:

$$\text{WCSS} = \sum_{j=1}^{k} \sum_{i=1}^{n} ||x_i - \mu_j||^2, \text{ if } i \in \text{cluster } j \tag{1.29}$$

Chapter 2

Data Description

For this thesis and in order to illustrate the *LocalGLMnet* architecture and performances we will construct models based on the *Lending Club Loan database*. This is one of the richest loan dataset that provides all loan information for all loans granted by the lending club between *2007* and *2011*, including current loan status that we will try to estimate through other available features. But first, let's introduce the lending club which is the United States largest loan marketplace that assists members in saving money, paying off high-interest debt, and taking charge of their financial futures. It's commonly described as a marketplace as it connects borrowers to investors allowing the former to access low interest rate. Thus, the data collection is fundamental for such an instance that needs to regulate this market though the determination of interest rates. In fact, using the information provided about the borrower, investors were able to search and browse the loan listings on the Lending Club website and choose loans that they wished to invest in. Then, the profit of the investors is interest rate based.

That's why data science can assist insurance firms in determining borrower risk as data scientists may develop models to forecast the likelihood of default or other risks by examining data on borrower demographics, credit ratings, and past behavior. These models can help insurance firms make informed judgments about interest rates, enabling them to establish rates that are suitable for the degree of risk associated with each borrower. That's why it is a difficult undertaking to determine interest rates in the insurance industry since it involves carefully taking into account a variety of variables, such as borrower risk, market circumstances, and investor expectations. Insurance businesses may use data science as a potent tool to evaluate and comprehend vast volumes of data to guide their interest rate choices. Moreover, data science may assist insurance providers in keeping an eye on market circumstances and adjusting interest rates but this will not be covered in this thesis.

Thus, in the Lending club peer-to-peer lending organisation determining interest rate is of main concern as borrowers are connected to investors and it needs data science to help providing

valuable insights into the risk and profitability of such transactions. By analyzing large amounts of data related to past loans, borrowers and other relevant factors, data science can identify patterns and trends that can help estimating the target likelihood of default. Precisely, in this context, the individual analysis can help insurance companies more accurately assess the risk associated with lending to specific borrowers, which can help them set interest rates that are appropriate for the level of risk involved. It can also help investors make more informed decisions about which loans to invest in, based on factors such as the borrower's creditworthiness or, as previously said, the probability of default. Thus, by leveraging the power of data analysis, insurance companies and investors ultimately improve the overall efficiency and effectiveness of the lending process.

2.1 Description of the objectives

The possibility that a borrower won't be able to fulfill their agreed-upon repayments on a certain loan during a given time frame, often one year, is known as the *default probability*. It may be used in a wide range of various credit analysis or risk management scenarios. The estimation of this rate for a company is essential because it has a direct impact on the credit rating as since there is a larger chance of default, lenders will normally charge higher interest rates. Along with this probability, insurance instances usually consider *loss given default*, that denotes the expected financial loss incurred by a bank or other financial institution as a result of a borrower defaulting on a loan and *exposure at default* which is defined as the a exposure of contracts at the time of the default. Finally, a *loan's estimated loss* is determined by multiplying the loss given default by the exposure at default as well as the chance of default.

More precisely, as for our dataset we use the formulation

$$D = \{Y_i, x_i\}_{i=1,\dots,n}$$

we denote the loan default probability:

$$\mathbf{P}[Y = 1|X = x] = p(x)$$

with $(Y|X = x) \sim Ber(p(x))$. As previously said, in actuarial setting we are more interested about the estimation of ratio's of a response feature Y and its exposure but as it appears that there is no row duplicates we will set an exposure $\varphi_i = 1$ for each individuals in the dataset as this information isn't known and as we do not identify any policy duplicates. Moreover, analyzing the lending club dataset, we have to deal with the *loan_status* target, that is a categorical feature, as response. This one can take three different levels that we will describe bellow:

- *Fully Paid*: The borrower has closed the account after the loan has been repaid.

- *Charged Off*: Creditor has written the account off as a loss, and the account is closed to future charges.
- *Current*: The loan is currently running.

	Frequency	Percentage	Cumulative percentage
Fully Paid	32 950	83 %	83 %
Charged Off	5 627	14.2 %	97.1 %
Current	1 140	2.9 %	100 %
Total	39 717	100 %	100 %

Table 2.1: *loan_status* levels frequencies (1)

Thus our target feature needs to be preprocessed, removing records with loan status as *Current*, as the loan is currently running and we can't infer any information regarding default from such loans. The previous step is useful as we go back to a binary response and dividing the number of *success* (in this case, the *Fully Paid* state) by the total number of contracts we otain the following final table:

	Frequency	Percentage	Cumulative percentage
Fully Paid	32 950	85.4 %	85.5 %
Charged Off	5 627	14.6 %	100 %
Total	38 577	100 %	100 %

Table 2.2: *loan_status* levels frequencies (2)

And thus, we can compute that $p(x)$ and advance that contracts that are assigned to default payment represent 14.59% of the total number of contracts in the portfolio. Moreover, as the probability mass function for the binomial exponential distribution is given by:

$$\mathcal{L}(p) = \binom{n}{k} p^k (1-p)^{n-k} \tag{2.1}$$

where $n \in \{1, 2, \dots\}$ is the number of trials, $p \in [0, 1]$ is the probability of success, and $k \in \{1, 2, \dots, n\}$ is the number of successes, in our case the number of defaulting contracts (5627). Then we can compute the expected *Fisher* information that measures the amount of information that the data provides about the parameter of interest p, and can be used to derive important properties of estimators such as their variance and asymptotic distribution and is given by:

$$\mathcal{I}(p) = \mathbf{E}[-H(\widehat{p})] = \mathbf{E}\left[-\frac{\partial^2}{\partial p^2} L(p)\Big|_{p=\widehat{p}}\right] = \frac{\mathbf{E}[Y]}{\widehat{p}^2} + \frac{n - \mathbf{E}[Y]}{(1-\widehat{p})^2} = \frac{n}{\widehat{p}(1-\widehat{p})} \tag{2.2}$$

Moreover, as the *Fisher* score $U(\theta) \approx Nor(0, \mathcal{I}(\theta))$ is defined as the first derivative of the

log-likelihood with respect to the canonical parameter (in the binomial case $\theta = ln(\frac{p}{1-p}) \longleftrightarrow p = \frac{e^\theta}{1+e^\theta}$) such that replacing p by its canonical expression in 2.1 and setting $U(\theta) = 0$ we get:

$$U(\theta) = \frac{\partial}{\partial\theta}L(\theta) = k - n(\frac{e^\theta}{1+e^\theta}) = 0 \tag{2.3}$$

We can find an estimation of the canonical parameter that is $\widehat{\theta} = ln(\frac{k}{n-k}) = -1.7674$ that gives the maximum-likelihood estimate of p that is $\widehat{p} = 0.14586$. Finally we can compute the observed information with respect to θ that is given by $\mathcal{I}(\widehat{\theta}) = -U^{'}(\widehat{\theta}) = n\frac{e^{\widehat{\theta}}}{1+e^{\widehat{\theta}}} = 4806.224$. And that leads to several important properties, as even if $\widehat{\theta}$ is the optimal value of the canonical parameter found by maximum likelihood estimation, other parameter points are plausible but we can compute a confidence interval for θ as $(\widehat{\theta} - \theta) \approx Nor(0, \frac{1}{\mathcal{I}(\theta)})$ that is the *large sample estimation*. Thus, we can find a confidence interval for our $\widehat{p}$ parameter as:

$$\mathbf{P}\left[\widehat{\theta} - 1.96\frac{1}{\sqrt{\mathcal{I}(\theta)}} \leq \theta \leq \widehat{\theta} + 1.96\frac{1}{\sqrt{\mathcal{I}(\theta)}}\right] \approx 0.95 \tag{2.4}$$

and as we know that under normally distribution assumption θ is contained in the interval $[\widehat{\theta} \pm 1.96/\sqrt{\mathcal{I}(\widehat{\theta})}] = [\widehat{\theta} \pm 0.02827] = [-1.795672; -1.739128]$ with approximate probability 95%, we can construct a confidence interval for the default probability estimate $\widehat{p}$ that is $[0.1424, 0.1494]$ under Normal approximation.

2.2 Preprocessing and features description

As the dataset we are working on seems to need cleaning we will detail the steps of this process in this section. In fact, the *loan* dataset consists of $n = 39717$ individual contracts with no repetition. Actually, detecting and addressing duplicate contract IDs is crucial for maintaining accurate policyholder records, preventing fraud, and ensuring regulatory compliance in the insurance industry. In fact, duplicate contract IDs can lead to inaccuracies in policyholder records, which can cause confusion and errors in the management of insurance policies. This can lead to problems such as incorrect billing, coverage issues, and claims processing delays. Duplicate contract IDs can also be used to commit fraud, as multiple policies with the same ID can be used to make multiple claims for the same event. This can result in financial losses for the insurer and higher premiums for policyholders. What the features concern, we denote a total amount of 110 explanatory features that mix *categorical*, *continuous* and *text* types. For the preprocessing part, the main goal will be to drop redundant and useless information.

Thus, before applying any cleaning algorithm we will list columns with no information that will not be used in our modelling process:

- Columns with same unique value across all rows: *pymnt_plan, initial_list_status, out_prncp, out_prncp_inv, next_pymnt_d, mths_since_last_major_derog, policy_code, application_type, annual_inc_joint, dti_joint, verification_status_joint, acc_now_delinq, tot_coll_amt, tot_cur_bal, open_acc_6m, open_il_6m, open_il_12m, open_il_24m, mths_since_rcnt_il, total_bal_il, il_util, open_rv_12m, open_rv_24m, max_bal_bc, all_util, total_rev_hi_lim, inq_fi, total_cu_tl, inq_last_12m, acc_open_past_24mths, avg_cur_bal, bc_open_to_buy, bc_util, delinq_amnt, mo_sin_old_il_acct, mo_sin_old_rev_tl_op, mo_sin_rcnt_rev_tl_op, mo_sin_rcnt_tl, mort_acc, mths_since_recent_bc, mths_since_recent_bc_dlq, mths_since_recent_inq, mths_since_recent_revol_delinq, num_accts_ever_120_pd, num_actv_bc_tl, num_actv_rev_tl, num_bc_sats, num_bc_tl, num_il_tl, num_op_rev_tl, num_rev_accts, num_rev_tl_bal_gt_0, num_sats, num_tl_120dpd_2m, num_tl_30dpd, num_tl_90g_dpd_24m, num_tl_op_past_12m, pct_tl_nvr_dlq, percent_bc_gt_75, tot_hi_cred_lim, total_bal_ex_mort, total_bc_limit, total_il_high_credit_limit*
- Columns with more than 50 % of missing values: *mths_since_last_delinq, mths_since_last_record*
- Columns with *NA* and 1 unique value: *collections_12_mths_ex_med, chargeoff_within_12_mths, tax_liens*

Finally this left us with **9** features to be cleaned, we list these with the number of respective missing values:

- Remaining columns with *NA*: *emp_title (2380), desc (12735), title (10), revol_util (50), pub_rec_bankruptcies (697), last_pymnt_d_year (71), last_pymnt_d_month (71), last_credit_pull_d_year (2), last_credit_pull_d_month (2)*

2.2.1 Cleaning by bootstrap

Therefore, we create a small function in R called *clean.data.by_bootstrap* that treats our lack of information for some attributes. Actually, we have to tackle both columns with no information (*NA* for all values) or with the same unique value for all individuals and columns with only a percentage of unavailable information. As we drop the former kind of features, we will use the previously cited and tuned function that takes the dataset and a threshold as parameters to perform a first filter on features with partial information. The threshold allows the user to make the filter more or less sensitive to missing values. What the treatment of partial uninformed features concerns, we choose to distinguish the used methodology for *numeric, factor* and *charachter* typed features. For the former types of columns we use this little bootstrap algorithm to replace missing values by sampling the non-missing values **with replacement** of

the concerned column:

Algorithm 1 Predict missing values by bootstrap

Require: ($\sum$ NA $\in$ column) / n $\leq$ Threshold
1: boot $\leftarrow$ UNIQUE(column) $\neq$ NA
2: **for** $i < n$ **do**
3: **if** column[i] = NA **then**
4: column[i] $\leftarrow$ Take 1 random value in boot with replacement
5: **end if**
6: **end for**

2.2.2 Cleaning by interpolation

Bootstrap can be a useful tool for data cleaning in some cases, but it may not be appropriate for loan contract datasets where the replacement of missing values in columns must be done appropriately to keep the intrinsic characteristics of each contract. In fact, the replacement of missing values in loan contracts often requires domain-specific knowledge and expertise to ensure that the replaced values are meaningful and accurate. For example, if a loan contract dataset has a missing value in the "interest rate" column, it is important to replace it with a value that is consistent with the prevailing interest rates in the market and the specific terms of the loan contract.

Thus, if we use bootstrap to clean the data in this scenario, we may end up replacing missing values with values that are not representative of the underlying population. In other words, the bootstrap samples may not reflect the true distribution of the loan contract dataset, and as a result, the replaced values may not accurately represent the characteristics of each contract.

Furthermore, loan contract datasets often have a complex structure and interdependencies between columns, such as the loan amount, loan term, and interest rate. The replacement of missing values in one column can affect the values in other columns, and this requires careful consideration and expertise to ensure that the resulting dataset accurately reflects the characteristics of each loan contract.

The strategy will be to fit linear regression to impute missing values in the loan dataset by fitting a model for each column with missing values and predicting the missing values using the predictor columns. Concretely, at each iteration (i.e for each feature to be cleaned) we will cut the dataset in two; the part with predictor attributes and non missing values of the column to be cleaned and the part with all predictors and the missing values of the column of interest. Then using the first created part, we will train a regression model to find a representation of the

missing columns with respect to all predictors such that:

$$missing\_col = \widehat{\beta_0} + \widehat{\beta_1} x_1 + \widehat{\beta_2} x_2 + \cdots + \widehat{\beta_p} x_p + \epsilon \tag{2.5}$$

where *missing_col* is the column with missing values, $x_1, x_2, ..., x_p$ are the predictor columns, $\widehat{\beta_0}, \widehat{\beta_1}, \widehat{\beta_2}, ..., \widehat{\beta_n}$ are the coefficients to be estimated, and ϵ is the error term. Then we use the trained regression model to predict the missing values in the second created dataset. As explained above, for each missing column we select the rows that have missing values in that column and predicts the missing values using the *predict*() function in R.

Algorithm 2 Predict Missing values with Regression

Require: A dataset *loan_complete* with complete data, a dataset *loan_missing* with missing data, a list *missing_cols* of columns with missing values in *loan_missing*, a list *predictor_cols* of predictor columns to use in the regression model

1: **for** *col* in *missing_cols* **do**
2: $y_{train} \leftarrow$ get values of *col* from *loan_complete*
3: $x_{train} \leftarrow$ get all columns from *loan_complete* except those in *missing_cols*
4: *model* $\leftarrow$ train linear regression model using x_{train} as predictors and y_{train} as response variable
5: *na_rows* $\leftarrow$ identify rows with missing values for *col* in *loan_missing*
6: $x_{test} \leftarrow$ get all columns from *loan_missing* except those in *missing_cols* for the rows identified in *na_rows*
7: $y_{pred} \leftarrow$ predict missing values for *col* using the trained *model* and x_{test}
8: replace missing values for *col* in *loan_missing* with y_{pred}
9: **end for**

At the opposite, for *character* typed attributes, we simply choose to replace missing values by the **"no_information"** character. This will be helpful for our embedding clustering and representations as we will try to identify if missing values in this feature types are highly linked in the input space, as if it's the case contracts with no information given regarding to the description of the loan purpose will be close to each other in this high-dimensional space.

2.2.3 Filters: correlation analysis

Correlation analysis is a statistical technique that measures the strength and direction of the relationship between two or more variables and as it's of main concern for this thesis, in the context of machine learning and data analysis, as it is often used as a starting point for identifying which feature may be most relevant to our default probability prediction. It's really important to point out that *filter analysis* involves selecting features based on some statistical criteria, such as correlation coefficients for instance or variance thresholds, mutual information scores. In contrast, *wrapper analysis* that will be performed in the next sections involves using a machine

learning algorithm to select features based on their performance in predicting the target variable such as step-wise procedure. The main advantage of *filter analysis*, such as correlation analysis, is that it is computationally efficient and can quickly narrow down the set of variables that are most likely to be relevant to the default probability prediction for example. However, it is also important to note that correlation analysis has some limitations as it can only identify linear relationships between features and may not capture more complex relationships. Additionally, correlation analysis cannot account for interactions between variables or identify which variables may be most informative in combination with others.

Thus, as the redundancy can not be detected directly by statistical, numerical understanding of the way features are calculated for every feature we analyze the correlation matrix of continuous features and we only keep one feature by highly correlated groups.

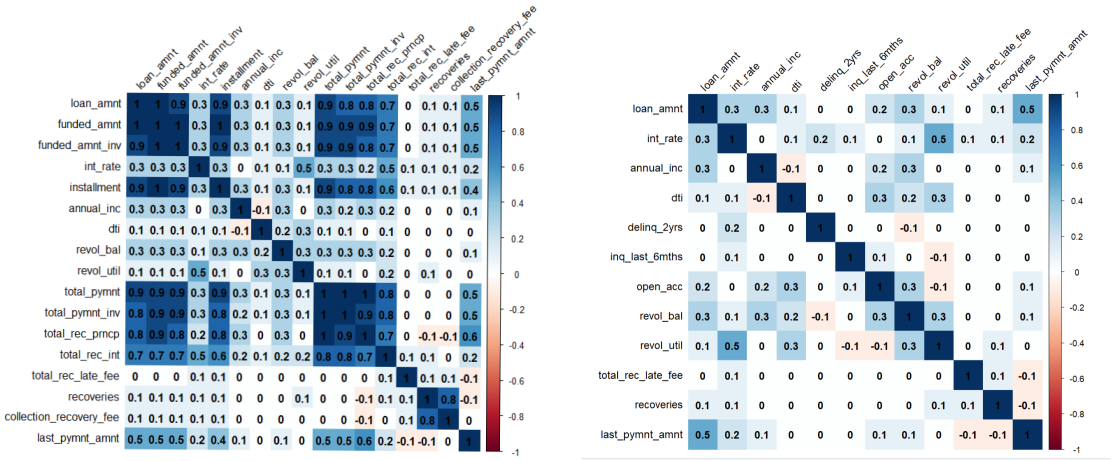


Figure 2.1: Correlation matrix cleaning for numeric input features

Then, we remove the following list of features {*funded_amnt, funded_amnt_inv, total_pymnt, total_pymnt_inv, total_rec_prncp, total_rec_int, installment, collection_recovery_fee, total_acc, pub_rec*} to attenuate grouped correlated attributes. Moreover, we will decide to drop the *grades* feature as it presents a correlation value of **1** with the *int_rate* when analyzing the correlation between categorical and continuous features. In fact, we can represent the link between interest rate and loan grades/subgrades using the following formula:

$$r = R_0 + R_g * g \tag{2.6}$$

where r is the interest rate, R_0 is the base interest rate, which represents the interest rate charged for loans with the lowest risk profile, K_g is a constant that represents the risk premium for loans of a certain grade or subgrade and g is the loan grade or subgrade assigned to a loan profile. Thus, it shows that the interest rate charged for a loan profile increases as the loan grade

or subgrade becomes riskier, as represented by the constant K_g.

2.2.4 Date features transformation

Before describing and trying to understand these, we have to treat the dates features *earliest_cr_line, last_pymnt_d, last_credit_pull_d, issue_d*. The description of the 4 attributes will be helpful:

- *earliest_cr_line*: The month the borrower's earliest reported credit line was opened.
- *last_pymnt_d*: Last month payment was received.
- *last_credit_pull_d*: The most recent month credit has been pulled for this loan.
- *issue_d*: The period of time when the loan was funded.

Looking at the format of these dates we denote that it takes the same form for all observations of the 4 attributes (**month - year**). We can use the ***lubridate*** package in R, precisely the ***my*** function which allows us to transform the *Character* attributes to *Date* format. Then two strategies are available; we can use the *as.numeric* function which provides us the difference of time between our date and the 01/01/1970 or we can choose to use the ***year*** and ***month*** functions to split the dates in two new attributes in order to keep the potential periodicity of the observations. As we know that in actuarial setting the economic situation of a period can have a heavy impact on the values taken by variables linked to a contract, we will choose to perform the latest variable transformation. Thus we create for each time feature x_j, two alternative ones *x_year* and *x_month*. But when dealing with time-related features in actuarial data, it's important to handle them appropriately in order to build a predictive model that can handle future information beyond the time range of the existing ones. There are several ways to deal with this information as time-related variables can be seen as cyclical variables for month features such that we can treat the month making the assumption that it has a cyclical behaviour and encode it as sine and cosine waves. This approach can help capture the seasonality of the data.

2.2.5 Features description

Finally this part of our preprocessing left us with **30** explanatory features that we will describe, but let's point out that when training our models, we will not include time features for instance, as we will generalize as much as possible so that it can handle with several different actuarial tabular data. So we will present the 12 remaining continuous features that will be subject to standardization in what follows.

- *loan_amnt*: The loan's specified amount, as requested by the borrower. This number will be affected if the credit department ever decides to lower the loan amount.
- *int_rate*: Interest rate on the loan.
- *annual_inc*: The yearly income the borrower submitted when registering.
- *dti*: A ratio established by dividing the borrower's self-reported monthly income by the sum of all monthly payments on all debt commitments, except the mortgage and the proposed LC loan.
- *open_acc*: How many credit lines a borrower has open in their credit history.
- *revol_util*: Revolving line usage rate, or the proportion of available credit that the borrower is using,
- *revol_bal*: Total credit revolving balance.
- *last_pymnt_amnt*: Most recent total payment received.
- *total_rec_late_fee*: Late fees received to date.
- *recoveries*: Post charge off gross recovery.
- *delinq_2yrs*: The number of 30+ days past-due incidences of delinquency in the borrower's credit file for the past 2 years.
- *inq_last_6mths*: The quantity of inquiries in the last six months (excluding auto and mortgage inquiries).

Let's notice that the feature *recoveries* is supposed to be highly correlated with the default probability as post charge off gross recovery is a term used in the banking and credit industry to refer to the amount of money that a lender is able to recover from a borrower after the borrower has defaulted on a loan and the lender has charged off the debt. In fact, when a borrower fails to make payments on a loan for a certain period of time (here denoted by the levels of the categorical feature *term* thus 36 or 60 months), then the lender may charge off the debt, which means they remove it from their books as a loss. However, the lender may still attempt to recover some or all of the outstanding balance from the borrower or a third-party debt collector and thus, the amount that the lender is able to recover from these efforts is referred to as the value of the *recovery* feature. That's why this feature will be removed from our analysis. Let's have a look over our categorical features and especially on the number of levels and the values that they can take for each one. In fact, it's of main concern to keep an eye over this as a high number of levels can lead to several issues in a machine learning model, that is why it often needs to be reduced.

- *emp_length*: Employment length in years. → 11 levels + NA: where 0 means less than one year and 10 means ten or more years.
- *verified_status*: If the source of the revenue was confirmed, whether the income was validated, or whether it was not. → 3 levels: *Not Verified, Source Verified, Verified.*
- *home_ownership*: The home ownership status provided by the borrower during registration → 5 levels: *rent, own, mortgage, other, none.*
- *term*: How many loan payments there are. → 2 levels: *36 months, 60 months.*
- *purpose*: A grouping the borrower gave for the loan application. → 14 levels.
- *addr_state*: State indicated in the loan application by the borrower. → 50 levels.
- *pub_rec_bankruptcies*: Indicator of the number of public record bankruptcies. → 2 levels: *0, 1 if 0 <.*
- *earliest_cr_line_year*: The year the borrower's earliest reported credit line was opened. → 53 levels: from *1969* to *2068.*
- *earliest_cr_line_month*: The month the borrower's earliest reported credit line was opened. → 12 levels.
- *last_pymnt_d_year*: Last year payment was received. → 9 levels: from *2008* to *2016.*
- *last_pymnt_d_month*: Last month payment was received. → 12 levels.
- *last_credit_pull_d_year*: The most recent year credit has been pulled for this loan. → 10 levels: from *2007* to *2016.*
- *last_credit_pull_d_month*: The most recent month credit has been pulled for this loan. → 12 levels.
- *issue_d_year*: The year when the loan was funded. → 5 levels: from *2007* to *2011.*
- *issue_d_month*: The month when the loan was funded. → 12 levels.

Text features:

- *emp_title*: The occupation listed by the borrower on the loan application.
- *desc*: The borrower's description of the loan.
- *title*: The borrower-provided loan title.

2.2.6 Levels reduction

Finally we have to deal with a last issue before going further in our analysis because some categorical variables have too many levels. In order to reduce non-arbitrarily the number of levels of the *emp_length, purpose, addr_state* and *earliest_cr_line_year* features we will perform clustering based on the odds of default for each level. For example, looking at this statistic for all the levels of the *purpose* feature we get the following partition:

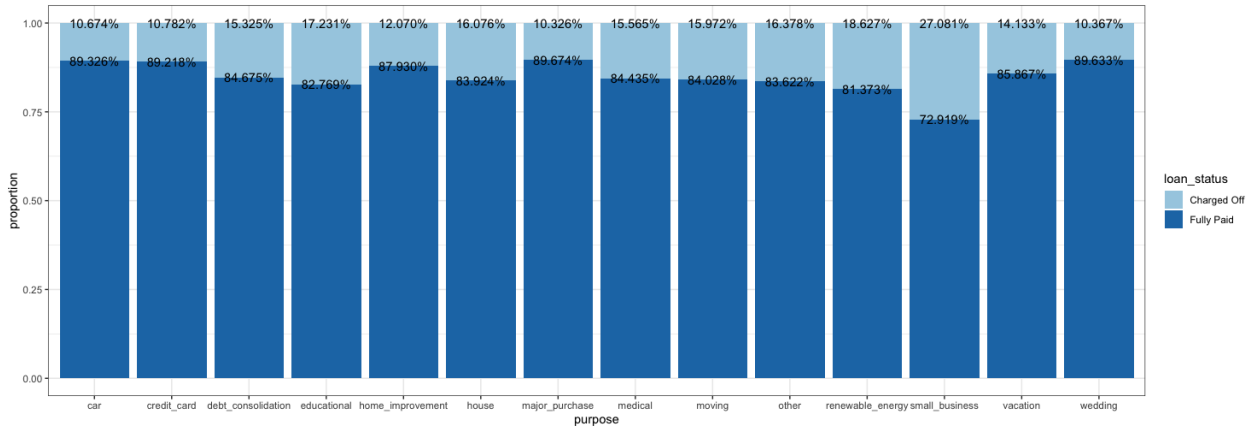


Figure 2.2: Odds of default for the *purpose* feature

Thus, as we want to perform the well known *k-means* clustering on this space representation of the levels of the 4 previously cited features, we will briefly explain the way we proceed. In fact, the little algorithm that follows is performed on each one to determine the appropriate clustering and thus to allow us to re-code the features levels:

Algorithm 3 levels reduction for 1 feature

contin ← contingency(feature ∼ loan_status)
d ← length(levels(feature))
for $i < d$ **do**
    $n_{p,i}$ ← contin[i, "Charged Off"] / contin[i, "Sum"]
    $n_{q,i}$ ← contin[i, "Fully Paid"] / contin[i, "Sum"]
    o_i ← $n_{p,i}$ / $n_{q,i}$
end for
k-means(o)

Therefore, to perform this levels reduction we will use the default *kmeans* function in R on the odds contingency tables computed with the little algorithm explained before. For each feature we compute the *within sum of squares* that is a measurement of the separation between each centroid and the occurrences of its corresponding class as said previoulsy and such that the

cluster values are farther apart from the centroid the bigger the *WSS*. The *fviz_cluster* function from the *factorextra* package in R allows to compute this value for different number of clusters. Thus looking at the elbow, we find the optimal number of clusters as it appears to be the bend in the knee formed in the *WSS* by number of clusters plot. Then we fit the *kmeans* clustering with the appropriate number of clusters and trying a total of 25 random sets of initial centers for the optimization.

Finally, what *addr_state* concern we compute 3 clusters. Once again we have a cluster lonely determined by the *NE*-linked observations. Another one consists of observations attributed to the [*DC, IA, IN, ME, WY*] levels. The remaining observations are part of a third group.

Thus it clusters the levels of our 4 categorical features and we can already interpret the results:

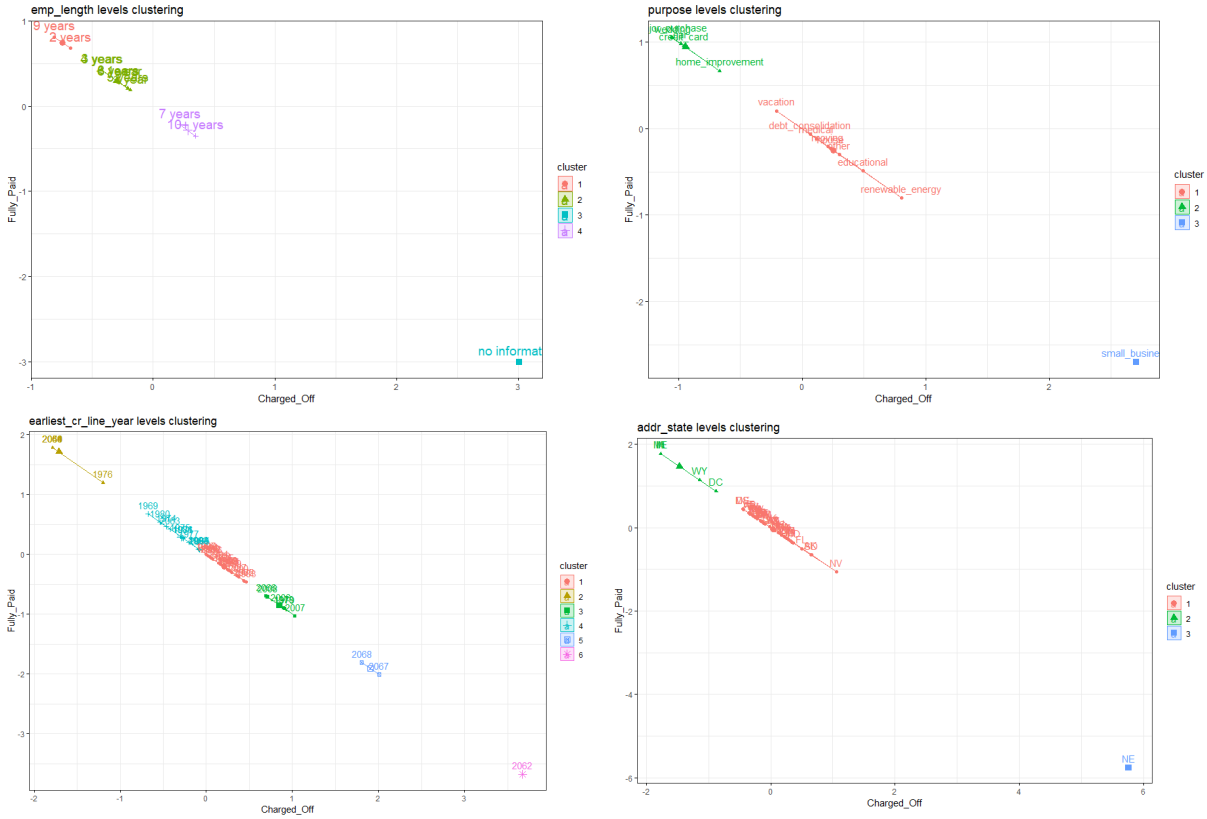


Figure 2.3: k-means clustering for levels reduction

In fact, what *emp_length* feature concerns we clearly see that the *na*, recoded with the *no information* label, observations stay in their own cluster. Moreover, we denote that we get a cluster with [*9 years, 2 years*], a cluster with [*7 years, 10+ years*] observations and a last one with the rest.

For the *purpose* levels the clustering works really well as we cut the observations in 3 distinct

ones. Once again we denote that a cluster only owns observations linked to *small_business*. A second one is attribuated to observations that correspond to levels [*car, credit_card, home_improvement, major_purchase*]. The remaining are included in a last one.

The *earliest_cr_line_year* levels were distributed among 6 different clusters. We observe that weird values and underrepresented levels belong to clusters of their own such as *2062* and [*2068,2067*].

2.3 Descriptive statistics and plots

Descriptive analysis of actuarial data is useful and important for several reasons. In fact, it allows us to gain insights into the underlying characteristics of the data, such as its distribution, mean, median, and variance. This helps us to understand the behavior of the data, identify potential outliers or anomalies, and gain insights into the drivers of risk or loss. Moreover, descriptive analysis can help us to identify patterns or trends in the data. This is particularly important in actuarial science, where we are often interested in identifying patterns of risk or loss over time, across different demographic groups, or in different regions. By identifying these patterns, we can develop more accurate models of risk and improve our ability to predict future outcomes.

In addition, descriptive analysis can help us to identify potential data quality issues or missing data. This is important because accurate data is critical for the development of reliable models of risk or loss. By identifying and addressing these issues, we can improve the quality of our data and ensure that our models are more accurate. This has already been done in the previous section with our preprocessing.

In terms of tuning parameters in machine learning applications, descriptive analysis of actuarial data can be useful for several reasons. Firstly, it can help us to identify the most important features that are predictive of risk or loss. By focusing on these variables, we can develop more efficient models that are better able to capture the underlying patterns of risk or loss. Secondly, descriptive analysis can help us to identify potential issues with our models, such as *overfitting* or *underfitting*. Overfitting occurs when a model is too complex and fits the training data too closely, resulting in poor generalization to new data. Underfitting occurs when a model is too simple and fails to capture the underlying patterns of risk or loss. By analyzing the descriptive statistics of the data, we can identify potential issues with our models and adjust the tuning parameters accordingly. Finally, descriptive analysis can help us to evaluate the performance of our models and by comparing the predicted outcomes of our models to the actual outcomes, we can identify areas of improvement and fine-tune the parameters of our models to achieve better accuracy. We display some statistics for out continuous features with the following figure to get

more insights about what will follow.

	vars	n	mean	sd	median	trimmed	mad	min	max	range	skew	kurtosis	se
loan_amnt	1	38577	11047.03	7348.44	9600.00	10152.18	6819.96	500.00	35000.00	34500.00	1.08	0.84	37.41
int_rate	2	38577	11.93	3.69	11.71	11.79	4.08	5.42	24.40	18.98	0.29	-0.45	0.02
annual_inc	3	38577	68777.97	64218.68	58868.00	61545.17	29456.30	4000.00	6000000.00	5996000.00	31.20	2308.36	326.96
dti	4	38577	13.27	6.67	13.37	13.31	7.72	0.00	29.99	29.99	-0.03	-0.86	0.03
delinq_2yrs	5	38577	0.15	0.49	0.00	0.01	0.00	0.00	11.00	11.00	5.03	39.72	0.00
inq_last_6mths	6	38577	0.87	1.07	1.00	0.69	1.48	0.00	8.00	8.00	1.38	2.52	0.01
open_acc	7	38577	9.28	4.40	9.00	8.88	4.45	2.00	44.00	42.00	1.01	1.69	0.02
revol_bal	8	38577	13289.49	15866.49	8762.00	10313.04	8854.09	0.00	149588.00	149588.00	3.21	15.05	80.78
revol_util	9	38577	48.70	28.36	49.10	48.79	34.84	0.00	99.90	99.90	-0.03	-1.11	0.14
total_rec_late_fee	10	38577	1.37	7.32	0.00	0.00	0.00	0.00	180.20	180.20	8.43	100.93	0.04
recoveries	11	38577	98.04	698.65	0.00	0.14	0.00	0.00	29623.35	29623.35	16.28	368.49	3.56
last_pymnt_amnt	12	38577	2746.24	4494.65	568.26	1687.12	712.09	0.00	36115.20	36115.20	2.66	8.54	22.88

Figure 2.4: Descriptive statistics for continuous features

2.3.1 Odds of default analysis

In actuarial context, the odds of default refer to the probability of a borrower defaulting on their loan payments and this probability can be calculated using the *loan_status* feature, which indicates whether a loan has been fully paid off or has been charged off. The odds of default can be expressed using the following formula:

$$\text{Odds of Default} = \frac{\text{Probability of Default}}{\text{Probability of Not Defaulting}}$$

To calculate the probability of default, we need to use the *Charged Off* category of the *loan_status* feature. This category indicates that the borrower has failed to repay their loan and the lender has charged off the debt and we can calculate the probability of default as follows:

$$\text{Probability of Default} = \frac{\text{Number of Charged Off Loans}}{\text{Total Number of Loans}}$$

To calculate the probability of not defaulting, we need to use the *Fully Paid* category of the *loan_status* feature. This category indicates that the borrower has successfully repaid their loan in full. We can calculate the probability of not defaulting as follows:

$$\text{Probability of Not Defaulting} = \frac{\text{Number of Fully Paid Loans}}{\text{Total Number of Loans}}$$

We already explore this concept in the Description of the objectives section but going further by exploring odds of default within categorical feature levels is important in an actuarial setting because it can provide insights into the risk associated with certain groups of borrowers. In

particular, understanding the relationship between *loan status* and categorical features such as employment status, loan purpose, or credit score can help lenders and insurers assess the likelihood of a borrower defaulting on a loan or insurance policy. And thus, exploring odds of default within categorical feature levels can help actuaries better understand the risk associated with different borrower groups and develop more accurate risk models for lending and insurance. Moreover, it will give indications about how our machine learning models have to attribute marginal weights without taking interactions in consideration.

Odds in addr_state

What the odds of default analysis in the state indicated in the loan application by the borrower concerns, we can verify that our previous clustering was efficient.

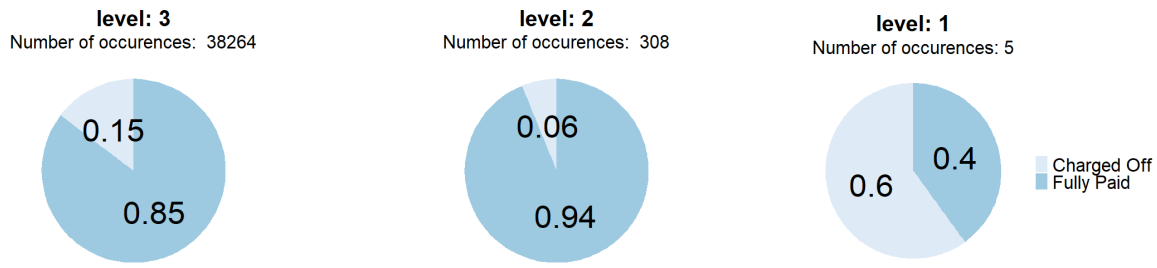


Figure 2.5: odds of default in levels

In fact, even if the third level of this feature seems to contain more contracts than the other, we can assess that the cutting has been well performed. The most interesting is that we denote more risky profiles in the level **3** of the new recoded *addr_state* feature than for the level **2**.

Odds in pub_rec_bankruptcies

As we analyze the distribution of the *loan_status* feature levels in the indicator of the number of public record bankruptcies levels, once again we denote that this analysis is helpful. In fact, a public bankruptcy is a complex process that can have significant consequences for the entity involved, as well as for its creditors and the global economy as it may involve negotiations with creditors to restructure the debt, reducing the amount owed or extending the repayment period, or it may result in a default on the debt and a total loss for the creditors.

Figure 2.6: odds of default in levels

Thus, even if the increase is not impressive, we can denote a significant higher level of risk for contracts which has been linked to at least one previous public bankruptcy.

Odds in term

We recall that the *term* features can take two levels which are [*36 months, 60 months*]. We know that long term loan payments are more subject to defaulting than short-term ones due to several factors, including changes in the borrower's financial situation, economic conditions, and interest rate fluctuations. However, it's important to note that the risk of default can be reduced through proper financial planning, including setting aside savings for unexpected expenses and regularly reviewing and adjusting the loan repayment plan.

Figure 2.7: odds of default in levels

Thus, we notice that our portfolio of contracts has a clear distinction in terms of risk between profiles linked to long and short term loans. In fact, over the 9481 long term loans incurred by borrowers, **25 %** are defaulting.

2.3.2 Interest rate analysis

The interest rate is a critical component to consider when analyzing actuarial data because it affects the value of future cash flows and the expected cost of an insurance policy or investment product. In the case of insurance products, the insurer collects premiums from policyholders and invests those funds to generate a return. Thus, the interest rate is the return that the insurer expects to earn on those investments, and it affects the insurer's ability to pay claims and maintain financial solvency.

Moreover, in actuarial analysis, the interest rate is used to calculate the present value of future cash flows, such as premiums, benefits, and expenses. The present value is the current worth of a future cash flow, and it reflects the time value of money, or the idea that money today is worth more than the same amount of money in the future due to the potential for investment returns. Thus, we can represent the link between interest rate and the probability of default using the following formula:

$$r = R_0 + K_{p(x)} * p(x) \tag{2.7}$$

where $K_{p(x)}$ is a constant that represents the increase in interest rate due to an increase in the probability of default and $p(x)$ is still the probability of default. Even if we drop this feature for our modeling process we can represent the way the interest rate is computed as a combination of grades/subgrades and the probability of default:

$$r = R_0 + K_g * g + K_{p(x)} * p(x) \tag{2.8}$$

The descriptive analysis of our interest rate will be done in several steps as it's important to get a global and marginal overview of the behaviour of this feature to understand properly how the attribution of interest rate values are discriminated in the case of the *lending club* instance. We display the conditional densities plot and the boxplots of the interest rate in both the levels of the *loan_status* and *term* factors. What the second plot concerns, we clearly notice that the *anova* theory will be helpful, but normality and homogeneity of variances should still be checked for each group separately. The application of this concept goes beyond the limits of this thesis so we stop at the interpretation of the results but let's point out that the *F-statistic* can still be computed as the ratio of the between-group mean square weighted by the size of each group, to the within-group mean square.

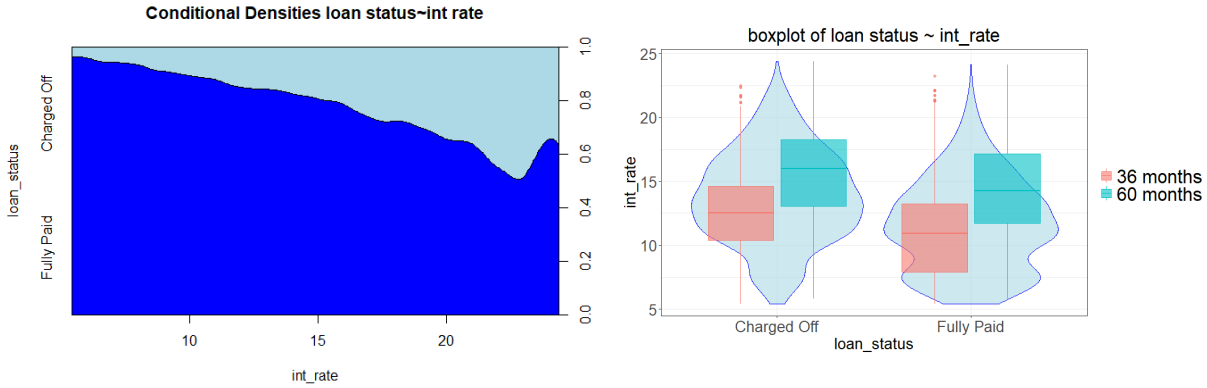


Figure 2.8: Interest rates distribution

What the first plot concerns, let's define what we mean by *conditional density* as this statistics of *int_rate* given *loan_status* is the probability density function of *int_rate* for a fixed value of *loan_status*. We can compute the conditional density using a technique called *kernel density estimation* as it starts by choosing a kernel function K, which is typically a probability density function with zero mean and finite variance. Then, for each value of *loan_status*, we compute the kernel density estimate of the conditional density of *int_rate* given *loan_status* by summing up the kernel functions centered at each observation of *int_rate*, weighted by their proximity to *loan_status*. Mathematically, this *KDE* estimate is given by:

$$\widehat{f}_{y|x}(y|x) = \frac{1}{nh}\sum_{i=1}^{n} K(\frac{y_i - y}{h})\mathbf{I}(x_i = x) \tag{2.9}$$

where h is the bandwidth, y_i and x_i are the ith observations of *int_rate* and *loan_status*, respectively, $\mathbf{I}(x_i = x)$ is an indicator function that takes on the value 1 if $x_i = x$ and 0 otherwise, and K is the kernel function. Thus, looking at our conditional densities of the *interest rate* accross the *loan status*, we denote that as the interest rate increase we are more likely to count defaulting loans. In fact, around 22% of interest rate we can approximate a rate of defaulting contracts to be equal to 50%.

2.4 Outliers analysis

In this section we will try to identify observations/contracts that deviate significantly from the majority of the data points in our dataset. In fact, outlier analysis is of main concern in actuarial settings, particularly for loan contract datasets as outliers can have a significant impact on the accuracy and reliability of statistical models and risk assessments by distorting distributions of data's.

Moreover, in this context, outliers can arise due to a variety of factors. For example, outliers may be the result of data entry errors, fraudulent activity, or extreme values that are legitimate but rare and can also be indicative of high-risk borrowers, which can have significant implications for the financial stability of a lending institution. Additionally, outlier analysis can help to identify potential areas for improvement in the data collection and management process. By identifying and addressing the root causes of outliers, actuarial professionals can improve the accuracy and reliability of loan contract datasets, which can lead to better lending decisions and improved financial performances for lending institutions.

In practice, for continuous features, we can use the *interquartile range* to identify potential outliers as it is the range between the 25th and 75th percentiles of the data. Any data point that falls below the first quartile (Q1) minus 1.5 times the IQR, or above the third quartile (Q3) plus 1.5 times the IQR, is considered as a potential outlier. Formally, we can write this as:

$$\text{Outlier} = \begin{cases} \text{data point} & \text{if; data point} < Q1 - 1.5 \times \text{IQR} \\ \text{data point} & \text{if; data point} > Q3 + 1.5 \times \text{IQR} \\ \text{not an outlier} & \text{otherwise} \end{cases} \tag{2.10}$$

But we will try some other less common way to identify anomalies in the data with a technique that is widely used in actuarial data analysis that is the *isolation forest* algorithm. In fact this unsupervised outlier detection machine learning algorithm aims to identify outliers by splitting values of a selected feature between the maximum and minimum values of this feature and dividing the data into two parts based on this split and repeating this recursively for each sub-group until all data points are isolated in their own tree nodes. Finally, anomalies are expected to have shorter average path lengths to reach them from the root node than normal data points such as explained by the following binary assignation:

$$h(x, \theta_m) = \begin{cases} 1 & \text{if } h_m(x) \leqslant \theta_m \\ -1 & \text{otherwise} \end{cases} \tag{2.11}$$

where x is the data point, $h_m(x)$ is the path length of the data point in the m-th tree, and θ_m is the threshold value for the m-th tree. The isolation factor of a data point x that measures how difficult it is to isolate that data point from the others is defined as:

$$s(x, t) = 2^{-\frac{\sum_{m=1}^{t} h_m(x)}{t}} \tag{2.12}$$

where t is the total number of trees used in the algorithm. Finally, the *isolation forest* algorithm can be used to identify anomalies in the data by comparing the isolation factor of each

data point with a predefined threshold value. A data point x is considered as an anomaly if its isolation factor $s(x,t)$ is less than the threshold value τ:

$$s(x,t) < \tau \tag{2.13}$$

Thus we will adapt the algorithm to plot the impact of anomalies on the target *loan status* for each continuous explanatory feature as we first create an instance of the Isolation Forest algorithm with a contamination parameter of 0.05, which specifies the approximate percentage of outliers in the data and creates a contingency table to calculate the percentage of anomalies for each *loan_status* level for the current feature. But the contamination parameter can not be initialized randomly as if the contamination parameter is set to 0.1, then the threshold value τ is set to the isolation factor value at the position that corresponds to the 10% largest values of the isolation factor. Thus, we use the *interquartile range* to compute the average outlier rate accross the continuous features that is 4.61%.

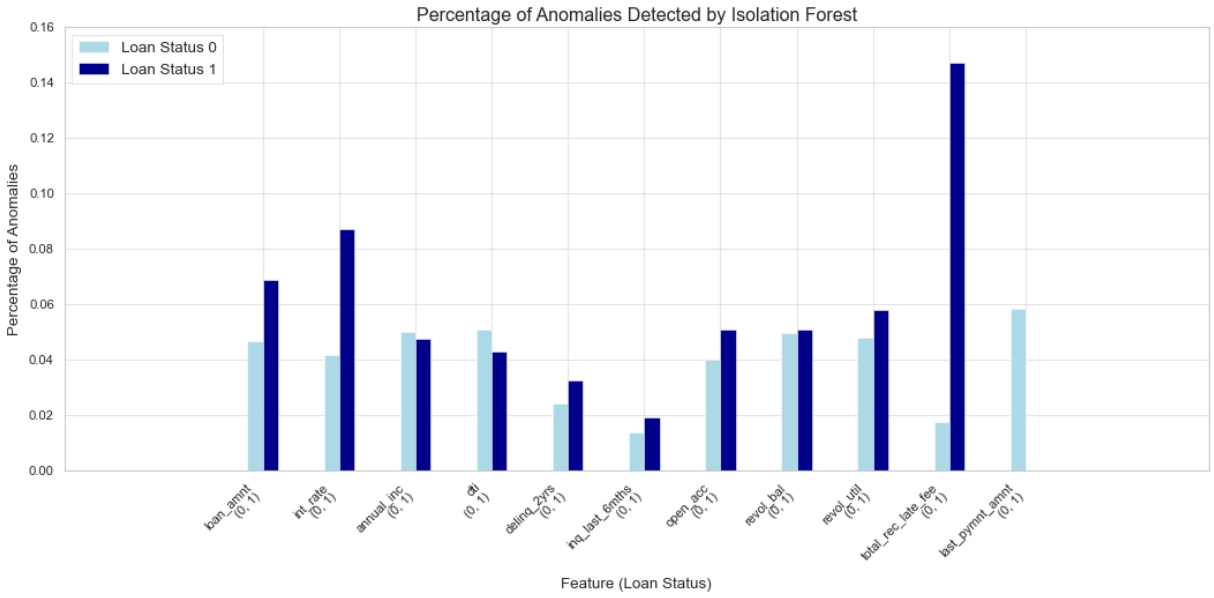


Figure 2.9: percentages of anomalies for each level of loan status

Chapter 3

Introduction to statistical modeling

3.1 Logistic regression

The logistic regression model works by estimating the probability of the binary outcome using a logistic function, an *S-shaped* curve that maps any input value to a probability value between 0 and 1. The input features can be categorical or continuous, and they are typically combined into a linear combination using weights and biases. The output of this linear combination is then passed through the *logistic* function to obtain the predicted probability.

Thus, as logistic regression is the binomial case of the generalized linear model, it's a statistical method used to model the probability of a binary response variable y given one or more predictor variables x and the probability of y being 1 (e.g., success) is modeled as a function of x using the following function:

$$\mathbf{P}(y = 1|x) = \sigma(x) = \frac{1}{1 + e^{(-z(x))}} \tag{3.1}$$

where z is a linear combination of the predictor variables:

$$z(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_p x_p \tag{3.2}$$

Here, β_0 is the intercept term and $\beta_1, \beta_2, ..., \beta_p$ are the coefficients associated with each predictor variable $x_1, x_2, ..., x_p$, respectively.

The coefficients in the logistic regression model are estimated using maximum likelihood estimation and the likelihood function for logistic regression is:

$$\mathcal{L}(\beta_0, \beta_1, ..., \beta_p) = \prod_{i=1}^{n} \left(\frac{e^{(z_i)}}{1 + e^{(z_i)}} \right)^{y_i} \left(\frac{1}{1 + e^{(z_i)}} \right)^{1-y_i} \tag{3.3}$$

where n is the sample size, y_i is the binary response variable for the i-th observation, and z_i is the corresponding linear combination of predictor variables for that observation. The likelihood function measures the goodness of fit of the model to the observed data. Then, the maximum likelihood estimates of the coefficients $\beta_0, \beta_1, ..., \beta_p$ are obtained by maximizing the log-likelihood function:

$$L(\beta_0, \beta_1, ..., \beta_p) = \sum_{i=1}^{n} (y_i z_i - \log(1 + \exp(z_i))) \tag{3.4}$$

This is typically done using numerical optimization algorithms, such as gradient descent or Newton-Raphson. We will come back to the definition of these concepts in the section Gradient descent of the next chapter as it takes a huge part of this one.

Once the coefficients are estimated, the logistic regression model can be used to predict the probability of y being 1 for new values of x. Once again, the predicted probability is given by the logistic function with the estimated coefficients:

$$\widehat{\mathbf{P}}(y = 1|x) = \frac{1}{1 + e(-\widehat{z})} \tag{3.5}$$

where $\widehat{z}$ is the linear combination of predictor variables using the estimated coefficients. Moreover as we go back to the expressions found in the previous section Generalized Linear Models we can write that in case of the binomial exponential distribution:

$$\begin{aligned} \mathbf{P}(y|\theta, \phi) &= \binom{n}{k} p^k (1-p)^{n-k} \\ &= e^{(k \log(\frac{p}{1-p}) + n \log(1-p))} \binom{n}{k} \end{aligned} \tag{3.6}$$

And from the different components of the exponential distribution formula explained in the previous section, we can find the canonical parameter expression of this distribution:

$$\theta = b^{'-1}(\mu) = \log(\frac{p}{1-p}) \tag{3.7}$$

$$b(\theta) = -n \log(1-p) = n \log(1 + e^{\theta}) \tag{3.8}$$

And thus as we know that it specifies the relationship between the mean of the response variable we can find the expectation and variance expressions for the response variable Y:

$$\mathbf{E}(Y) = \psi^{'}(0) = b^{'}(\theta) = np \tag{3.9}$$

$$\phi = 1 \tag{3.10}$$

$$\text{Var}(Y) = \text{Var}(\mu)\frac{\phi}{\varphi} = \mu(1-\mu) = p(1-p) \tag{3.11}$$

Additionally, we can show how we get these assumptions as the cumulant function of a random variable X is defined as the logarithm of its moment generating function, that is:

$$\kappa_X(t) = \log(M_X(t)) = \log(\mathbf{E}[e^{tX}]) \tag{3.12}$$

then for our binomial distribution with parameters n and p, the moment generating function is given by:

$$M_X(t) = \mathbf{E}[e^{tX}] = \sum_{k=0}^{n} e^{tk}\binom{n}{k}p^k(1-p)^{n-k} \tag{3.13}$$

and taking the logarithm of this expression, we obtain:

$$\begin{aligned}
\kappa_X(t) &= \log\left(\sum_{k=0}^{n} e^{tk}\binom{n}{k}p^k(1-p)^{n-k}\right) \\
&= \log\left(\sum_{k=0}^{n} \binom{n}{k}(pe^t)^k(1-p)^{n-k}\right) \\
&= \log\left((pe^t+1-p)^n\right) \\
&= n\log(pe^t+1-p)
\end{aligned} \tag{3.14}$$

This allows us to get the mean and variance expressions of the distribution back, as we can take the first and second derivatives of the cumulant generating function, respectively:

$$\mu = \frac{\partial}{\partial t}\kappa_X(t)\bigg|_{t=0} = np \tag{3.15}$$

$$\sigma^2 = \frac{\partial^2}{\partial t^2}\kappa_X(t)\bigg|_{t=0} = np(1-p) \tag{3.16}$$

And finally, to find the link function expression for the binomial distribution, we can use the assumption that the canonical link function for the binomial distribution is the *logit* function:

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p \tag{3.17}$$

we can demonstrate this by solving for p in terms of the mean and variance expressions we just found. Setting $p = \mu/n$, we have:

$$\sigma^2 = np(1-p) \rightarrow \frac{\sigma^2}{\mu(1-\mu/n)} = 1 - \frac{\mu}{n} \tag{3.18}$$

that gives after the application of the natural logarithm of both sides and rearrangement:

$$\log\left(\frac{\mu}{n-\mu}\right) = \log\left(\frac{\sigma^2}{\mu(1-\mu/n)}\right) \tag{3.19}$$

and thus this suggests that the *logit* function is a suitable choice for the link function of the binomial exponential distribution, since taking the *logit* of μ/n gives $\log(\mu/(n-\mu))$.

To summarize the *logit* function and the *logistic* function are related but different functions as the first mentioned formula allows to maps real numbers to probabilities as it can takes any values between $-\infty$ and ∞, while the second one maps probabilities to real numbers.

3.1.1 Processing and results

We will fit the logistic regression model to our binary outcome *loan_status* with the predictors previously chosen after the preprocessing part. In a first step we will avoid introducing seasonality in our estimations but this will be explored later with the introduction of the *date* features.
As we want to capture non-linear affects as usual in actuarial settings, we will cut our *numeric* typed features. The main strategy will be to choose the quartiles of these features as breaking points but we will adapt our cutting manually for features that seems to be very skewed such as *delinq_2years* and *total_late_rec_fee*. This approach, also known as binning or discretization, can be useful in situations where there is a non-linear relationship between the continuous feature and the response variable, and when we do not want to use polynomial terms or splines. In fact, simply adding higher-order polynomial terms to the model can lead to overfitting when the model fits the noise in the data rather than the underlying signal introduced.

Moreover, we will use *stepwise selection* to select the subset of predictors that yields the best model fit and that combines forward selection and backward elimination. Precisely, the idea of stepwise selection is to add or remove predictors from the model in a step-by-step fashion based on some statistical criterion, such as *AIC* or *BIC*. In fact, the algorithm starts with an empty model and iteratively adds and removes variables based on their statistical significance, until no additional variables improve the model's fit beyond a certain threshold or until only a predetermined number of variables remain in the model. Here is the pseudo code for this algorithm as it will more clear for the reader:

Algorithm 4 Stepwise selection procedure for logistic regression with AIC

Require: A dataset *data* with outcome variable y and predictor variables $x_1, \ldots, x_p$, $\widehat{p}$ is $\widehat{p(x)} = \mathbf{P}[\widehat{Y} = 1 | X = x]$

1: Initialize an empty model M_0 that only includes the intercept: $M_0 : \log(\frac{\widehat{p}}{1-\widehat{p}}) = \widehat{\beta_0}$
2: Set the current model to $M = M_0$ and the current set of predictor variables to $S = \emptyset$
3: **for** $k = 1$ to p **do**
4: Find the predictor variable x_k that, when added to S, produces the smallest AIC for the logistic regression model with $S \cup x_k$ as predictors:

$$\min_{x_k \notin S} \text{AIC}(M_k), \quad M_k : \log(\frac{\widehat{p}}{1-\widehat{p}}) = \widehat{\beta_0} + \sum_{x_i \in S} \widehat{\beta_i} x_i + \widehat{\beta_k} x_k$$

5: Add x_k to S and update the current model to $M = M_k$
6: **if** no additional variable can be added to S **then**
7: Terminate the forward selection stage and proceed to the backward elimination stage
8: **end if**
9: **end for**
10: **while** at least one predictor variable can be removed from S **do**
11: Find the predictor variable x_j that, when removed from S, produces the smallest AIC for the logistic regression model with $S \setminus x_j$ as predictors:

$$\min_{x_j \in S} \text{AIC}(M_j), \quad M_j : \log(\frac{\widehat{p}}{1-\widehat{p}}) = \widehat{\beta_0} + \sum_{x_i \in S \setminus \{x_j\}} \widehat{\beta_i} x_i$$

12: Remove x_j from S and update the current model to $M = M_j$
13: **end while**
14: **return** The final model M and the set of predictor variables S

To this end we will perform a cross validation procedure to validate the choice of features for each step of the stepwise selection using *AIC* as criterion. You can fin the description of the code in the section R code for logistic regression and stepwise selection of the appendices.

Thus we will stop and retain the model chosen after the optimization of this algorithm and compare the results looking at the performances of the full model and the stepwise model. In fact, as we want to prevent our model from overfitting, we will try to reduce its complexity while insuring accuracy. Moreover, the full logistic regression model may be difficult to interpret when it includes a large number of features, especially with our large insurance databases, and by reducing the complexity of the model, the actuary can focus on the most important variables and better understand how they affect the outcome. In the final step of the application of the previously explained procedure we are left with the following predictors for the *loan_status* target:

$$
\begin{aligned}
loan\_amnt \sim\ & loan\_amnt\_first\_q + loan\_amnt\_second\_q + loan\_amnt\_third\_q \\
& + loan\_amnt\_third\_q + term36.months + int\_rate\_first\_q \\
& + int\_rate\_second\_q + int\_rate\_third\_q + emp\_length1 + emp\_length2 \\
& + emp\_length3 + annual\_inc\_first\_q + annual\_inc\_second\_q \\
& + annual\_inc\_third\_q + verification\_statusNot.Verified \\
& + verification\_statusSource.Verified \\
& + purpose1 + purpose2 + addr\_state2 + open_acc\_first\_q + revol\_util\_first\_q \\
& + revol\_util\_second\_q + revol\_util\_third\_q + last\_pymnt\_amnt\_first\_q \\
& + last\_pymnt\_amnt\_second\_q + last\_pymnt\_amnt\_third\_q \\
& + pub\_rec\_bankruptcies0
\end{aligned}
\tag{3.20}
$$

However, *stepwise selection* can be unstable, meaning that the set of selected features can change if the data is slightly altered or if a different starting point is chosen. Thus, as said previously a sensitivity analysis has been performed to assess for the validity of this features selection, as we cut the dataset in 5 folds for which the procedure was repeated 10 times to check for inconsistency of the *stepwise features selection*. Moreover, as we retain the *binomial deviance* for the train and test set, by comparing the performances of the full logistic regression model and the stepwise selection procedure model, we can determine which model is more appropriate for the given data and default probability prediction. We report in the following table the results for the full model that contains all features from our preprocessing and for the stewpise reduced model that includes features described above:

Models	$D_{\mathcal{D}}$	$AIC_{\mathcal{D}}$	$D_{\mathcal{T}}$
Full model	17 772.86	17 868.86	4 468.25
stepw. model	17 781.44	17 841.44	4 469.62

Table 3.1: Results for the Logistic regression

Another argument for our appropriate choice of features by *stepwise selection* is to check that it effectively prevents for overfitting and as the final values for the deviance statistics in the train and test are comparable we can go further with this subset of features, as we reduced the number of variables in the model without sacrificing too much predictive power. In addition, we will introduce other more robust feature selection techniques that will be helpful to understand the way this is done in neural networks with regularization methods such as *Lasso* or *Ridge*.

3.2 Regularization

Lasso (Least Absolute Selection and Shrinkage Operator) and *Ridge* regularization are two commonly used techniques to prevent overfitting in linear regression models as the *Ridge* one adds a penalty term to the sum of squares of the model coefficients, which forces the model to choose smaller coefficient values. But it doesn't perform feature selection such as *lasso* implicitly and thus can't shrink coefficients to zero, which makes it less effective in scenarios where there are many irrelevant predictors. The objective function for this regularization can be denoted as:

$$\text{minimize} \left(\sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda_2 \sum_{j=1}^{p} \beta_j^2 \right) \tag{3.21}$$

where β_j are the coefficients for the p predictors, x_{ij} are the values of the jth predictor for the ith observation, y_i is the response variable for the ith observation, β_0 is the intercept term, and λ_2 is a tuning parameter that controls the strength of the ℓ_2-penalty.

Lasso regularization, on the other hand, adds a penalty term to the sum of absolute values of the model coefficients, which can result in some coefficients being exactly null. But this ℓ_1-penalty can struggle with correlated predictors, as it tends to select one of the correlated predictors randomly and ignore the others. The objective function for *Lasso* regularization can be written as:

$$\text{minimize} \left(\sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right) \tag{3.22}$$

where the notations have the same meaning as in the *Ridge* regularization equation. Thus, if we want to build logistic regression with *Lasso* and *Ridge* penalty also known as the *elastic net* that overcome the drawbacks and limitations of both regularisation's we get this loss function:

$$\mathcal{L}(\widehat{\beta}; y_i, x_i) = \sum_{i=1}^{n} \left(y_i \log \left(\frac{\widehat{p}_i}{1-\widehat{p}_i} \right) + (1-y_i) \log \left(\frac{1-\widehat{p}_i}{\widehat{p}_i} \right) \right) + \lambda_1 \sum_{j=1}^{p} |\widehat{\beta}_j| + \lambda_2 \sum_{j=1}^{p} \widehat{\beta}_j^2 \tag{3.23}$$

where,

$$\lambda_1 \geq 0, \lambda_2 \geq 0$$

And thus the logistic regression model is extended by adding *Lasso* and *Ridge* penalties and

the hyperparameters λ_1 and λ_2 control the strength of the penalties, as the main objective is still to reduce the model variance to increase its accuracy. We will compare the results with the 3 regularization's techniques explained previously but as we already filter our input features in term of correlations we don't expect to encounter improper behavior of the *Lasso* one.

To fit the model, we minimize the loss function $L(\beta)$ using gradient descent or a similar optimization algorithm. The gradient of the loss function with respect to the coefficients β_j is given by:

$$\forall \beta_j : \frac{\partial \mathcal{L}(\beta)}{\partial \beta_j} = \sum_{i=1}^{n} (p_i - y_i) x_{ij} + \lambda_1 \text{sign}(\beta_j) + 2\lambda_2 \beta_j \tag{3.24}$$

where,

$$\text{sign}(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x = 0 \\ -1, & \text{if } x < 0 \end{cases}$$

We can then use this gradient to update the coefficients in an iterative algorithm such as gradient descent. Moreover, it can be shown that the logistic regression model with *Lasso* and *Ridge* penalties is equivalent to solving the following constrained optimization problem:

Minimize

$$-\sum_{i=1}^{n} \left(y_i \log \left(\frac{p_i}{1 - p_i} \right) + (1 - y_i) \log \left(\frac{1 - p_i}{p_i} \right) \right) \tag{3.25}$$

subject to

$$\begin{aligned} \sum_{j=1}^{p} |\beta_j| \leq t_1, \\ \sum_{j=1}^{p} \beta_j^2 \leq t_2 \end{aligned} \tag{3.26}$$

where t_1 and t_2 are chosen to control the strength of the penalties. This optimization problem can be solved using techniques such as coordinate descent or quadratic programming.

3.2.1 Processing and results

We go back to the expression of the full model build in the previous Logistic regression section and as we have created matrices for the predictor variables and the response we will compare our performances and coefficients convergence to *0* to choose the appropriate regularization.

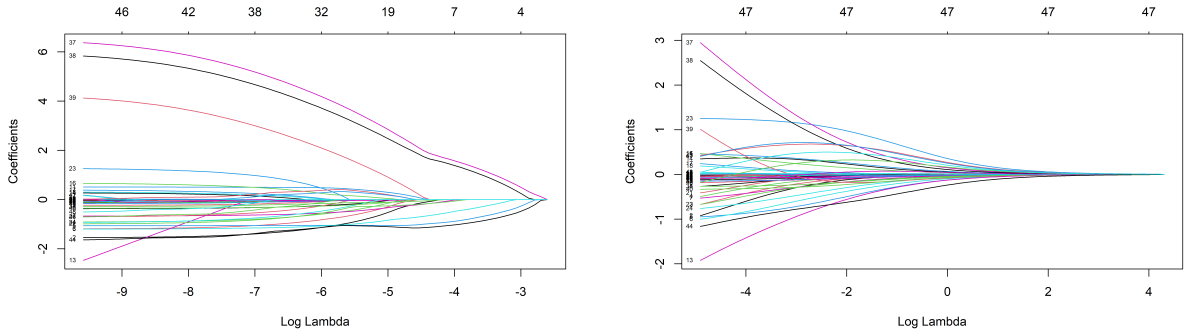


Figure 3.1: Coefficients conv. for Lasso and Ridge with different λ

Once again we let the reader try to understand the implementation with the provided code but we only join the R code for elastic net with CV, based on grid-search for the hyperparameters λ and α, in the appendices for this section. As expected in terms of convergence of regression coefficients, ℓ_2 regularization typically results in smoother convergence, as the penalty term is quadratic and therefore provides a smooth optimization surface. On the other hand, ℓ_1 regularization can result in more erratic convergence, as the penalty term is non-differentiable at 0 and therefore results in sharp corners in this optimization surface. But the coefficient convergences are not the only metrics to keep an eye on as the performances in term of *Bernoulli deviance* is of main concern. We summarize the results in what follows:

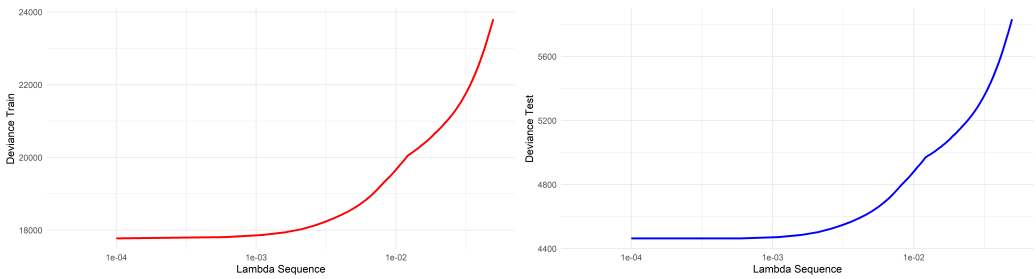


Figure 3.2: Lasso performances with different λ

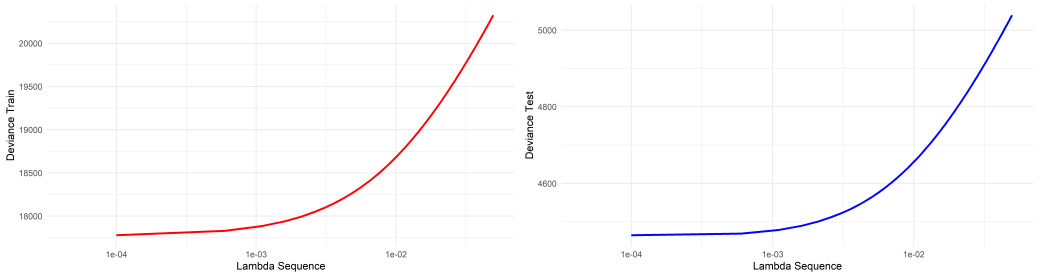


Figure 3.3: Ridge performances with different λ

λ	α	$D_{\mathcal{D}}$	$D_{\mathcal{T}}$
1e-01	0.1	23014.20	4380.655
1e-01	0.2	23599.61	4506.229
1e-01	0.3	24167.68	4628.799
1e-01	0.4	24845.70	4775.148
1e-01	0.5	25659.48	4950.942
1e-01	0.6	26420.18	5114.783
1e-01	0.7	26824.50	5197.148
1e-01	0.8	26852.23	5202.951
1e-01	0.9	26852.23	5202.951
1e-01	1.0	26852.23	5202.951
5e-02	0.1	21775.70	4106.186
5e-02	0.2	22127.66	4180.046
5e-02	0.3	22429.58	4244.845
5e-02	0.4	22728.82	4309.565
5e-02	0.5	23013.66	4371.620
5e-02	0.6	23323.06	4439.654
5e-02	0.7	23685.05	4519.025
5e-02	0.8	24109.87	4611.898
5e-02	0.9	24606.83	4720.262
5e-02	1.0	25136.66	4836.594
1e-02	0.1	19784.79	3670.601
1e-02	0.2	19875.17	3687.128
1e-02	0.3	19977.82	3706.180
1e-02	0.4	20082.98	3725.503
1e-02	0.5	20178.96	3744.020
1e-02	0.6	20288.26	3765.329
1e-02	0.7	20411.27	3789.642
1e-02	0.8	20540.98	3815.188
1e-02	0.9	20646.98	3836.312
1e-02	1.0	20755.90	3858.270
5e-03	0.1	19317.72	3569.041
5e-03	0.2	19343.26	3572.923
5e-03	0.3	19374.59	3578.143
5e-03	0.4	19410.17	3584.495
5e-03	0.5	19451.23	3592.021
5e-03	0.6	19494.43	3599.544
5e-03	0.7	19538.15	3607.043
5e-03	0.8	19579.59	3614.006
5e-03	0.9	19613.24	3619.828
5e-03	1.0	19647.60	3626.046
1e-03	0.1	18883.28	3472.167
1e-03	0.2	18881.00	3471.207
1e-03	0.3	18879.24	3470.418
0.001	0.4	18877.42	3469.689
0.001	0.5	18876.01	3469.070
0.001	0.6	18874.82	3468.458
0.001	0.7	18873.63	3467.845
0.001	0.8	18872.76	3467.317
0.001	0.9	18872.11	3466.838
0.001	1.0	18870.75	3466.164
0.0005	0.1	18829.13	3459.801
0.0005	0.2	18827.42	3459.132
0.0005	0.3	18825.46	3458.471
0.0005	0.4	18823.66	3457.852
0.0005	0.5	18822.12	3457.316
0.0005	0.6	18820.69	3456.814
0.0005	0.7	18819.20	3456.295
0.0005	0.8	18817.87	3455.824
5e-04	0.9	18816.61	3455.386
5e-04	1.0	18815.35	3454.946
1e-04	0.1	18791.95	3451.296
1e-04	0.2	18791.67	3451.170
1e-04	0.3	18791.41	3451.035
1e-04	0.4	18791.13	3450.903
1e-04	0.5	18790.93	3450.785
1e-04	0.6	18790.75	3450.674
1e-04	0.7	18790.56	3450.568
1e-04	0.8	18790.35	3450.460
1e-04	0.9	18790.17	3450.362
1e-04	1.0	18790.00	3450.273

Table 3.2: Results for Logistic regression with elastic net regularization for different λ, α

3.3 Neural Networks

A neural network with one hidden layer is a type of machine learning model used for classification and regression tasks. More precisely, it consists of an input layer, a hidden layer, and an output layer where the input layer takes the predictor variables, and the output layer produces the predicted probability of default. The hidden layer contains a set of hidden units, each of which computes a weighted sum of the input variables and applies a nonlinear activation function to the result.

Since the probability of default is modeled using the *binomial distribution*, which is a discrete distribution that describes the number of successes in a fixed number of independent *Bernoulli* trials, it has two parameters: the number of trials n and the probability of success p. In our case, the number of trials is fixed at 1, since we are only interested in whether or not a default occurs and the probability of success p is the probability of default, which is what we want to predict.

Then, the neural network can be trained by minimizing the *binomial deviance* function, which measures the difference between the predicted probabilities and the observed outcomes. The binomial deviance function is defined as:

$$D(y_i, \widehat{y_i}) = \begin{cases} 2\varphi_i(y_i \log(\frac{y_i}{\widehat{y_i}}) + (1-y_i)\log(\frac{1-y_i}{1-\widehat{y_i}})), & \text{if } y_i > 0 \\ -2\varphi_i \log(1-\widehat{y_i}), & \text{if } y_i = 0 \\ -2\varphi_i \log(\widehat{y_i}), & \text{if } y_i = 1 \end{cases}$$

where y_i is the observed outcomes (0 or 1), $\widehat{y_i}$ is the predicted probability, and n is the sample size. We will demonstrate how we go back to this expression from the likelihood function of a logistic regression model in the section Loss Function.

To train the neural network, we start by initializing the weights and biases randomly. Then, we feed the input variables into the network and calculate the predicted probability of default using the weighted sum and activation function in the hidden layer and a logistic activation function in the output layer. Next, we calculate the binomial deviance function using the predicted probabilities and the observed outcomes. We use backpropagation to update the weights and biases in the network to minimize the deviance function. As explained previously, backpropagation involves calculating the gradient of the deviance function with respect to the weights and biases and using this gradient to update the weights and biases in the opposite direction of the gradient.

We repeat this process for a number of epochs (i.e., iterations) until the binomial deviance function reaches a minimum or convergence is achieved. At this point, the neural network has been trained to predict the probability of default using the input variables. For the introduction of this kind of prediction models, we display the chosen architecture as it has been advanced by the universal approximation theorem that a single-hidden layer architecture can approximate any function arbitrarily well (we cite Y. Lu and Lu (2020) that helps for understanding the way it can approximate probability distributions):

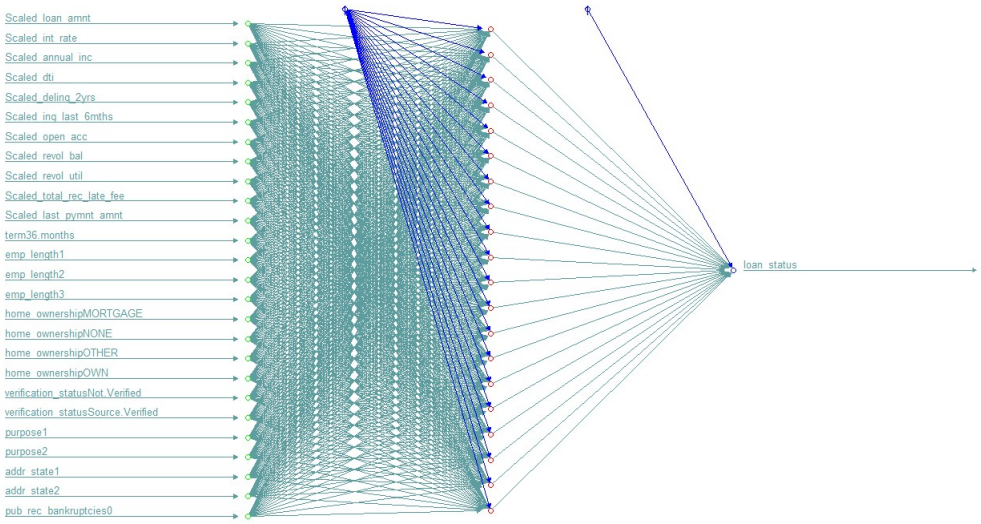


Figure 3.4: Single-hidden layer architecture (universal approximation theorem)

It should be noted that, as you know, a neural network only takes numerical variables as input, and that is why we have transformed the categorical features into binary ones for each level of this feature while removing one level from our analysis to avoid any problem linked to multicollinearity (dummy encoding). Moreover, the treatment of continuous features only needs standardization instead of categorization by cutting as neural networks allows to catch non-linear relationships between inputs and outputs. So, suppose we have a variable x_j with n observations, then we can standardize X to the range $[0, 1]$ by subtracting the minimum value of x_j and dividing by the range of x_j:

$$x_{ij}^* = \frac{x_{i,j} - \min_{i=1,\dots,n}(x_{i,j})}{\max_{i=1,\dots,n}(x_{i,j}) - \min_{i=1,\dots,n}(x_{i,j})} \tag{3.27}$$

3.3.1 Dropout and early stopping

We will introduce two additional regularization techniques used when training a neural network as the way the weights are trained allows us to perform *dropout* and *early stopping*. The idea behind *dropout* is to introduce noise in the network during training, making it more robust and

preventing it from relying too heavily on any input feature or neuron and thus to overfit our data's. In fact, *dropout* can be seen as a form of ensemble learning, where multiple models are trained on different subsets of the data and where the predictions are combined to improve the overall performance of the model. As said, by introducing noise in the network during training, dropout forces the network to learn more robust weights that are not specific to a particular subset of the data. So, during the training of our neural network, each neuron in a layer where *dropout* has been introduced is retained with a probability of p and dropped out with a probability of $1 - p$, This means that the output of each neuron is multiplied by a *Bernoulli* random variable which is either 0 or 1 with the parameter p and then the output of a layer with a *dropout* function introduced can be expressed mathematically as:

$$\mathbf{h}^{(l)} = f(\mathbf{h}^{(l-1)}W^{(l)} + \mathbf{b}^{(l)}) \otimes \mathbf{m}^{(l)} \tag{3.28}$$

where $\mathbf{h}^{(l)}$ is the output of layer l, f is the activation function, $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the weight matrix and bias vector for layer l, $\otimes$ is the element-wise product (Hadamard product), and $\mathbf{m}^{(l)}$ is a binary mask vector of the same size as $\mathbf{h}^{(l)}$ that is sampled afresh for each input sample and dropout iteration. During inference, the dropout layers are turned off, and the activations are scaled by the probability p to account for the fact that more neurons are active during testing than during training and the dropout rate p is a hyperparameter that can be tuned using cross-validation to find the best value for the given dataset and network architecture. Finally, *dropout* can be applied to different layers of the network, including the input layer, hidden layers, and output layer but it is typically more effective to apply *dropout* to the hidden layers, as the input and output layers are usually less prone to overfitting. Actually, we invite the reader who wants to learn more about the subject to have a look at the survey of dropout methods for deep neural networks that has been performed by Labach, Salehinejad, and Valaee (2019).

Early stopping is a cross-validation technique used to prevent overfitting in machine learning models and involves dividing the training data into training and validation sets, and monitoring the performances of the model on the validation set during training. The training process is then stopped earlier when the validation loss stops improving for a specified number of epochs, indicating that the model has started to overfit the training data. The number of epochs to wait before stopping the training process also called the *patience* is a hyperparameter that can be tuned using once again cross-validation as if the number of epochs is set too low, the model may not have enough time to converge to a good solution and at the opposite if the number of epochs is set too high, the model may overfit the training data. Early stopping can also be combined with other regularization techniques, such as weight decay, to further improve the generalization of the model as using a combination of regularization techniques, it is possible to achieve better performances on new data while avoiding overfitting the training data. This will be explored in

what follows as we will provide the pseudo-code of this procedure but we refer to what has been reviewed in the bias regularization in neural network models for general insurance pricing paper Wüthrich (2022).

Algorithm 5 Early Stopping Algorithm

Input: Training data $\mathcal{D}_{train}$, validation data $\mathcal{D}_{val}$, number of epochs n, early stopping patience k, initial parameter values θ_0

Output: Optimized parameter values θ

Initialize $\theta \leftarrow \theta_0$; $best_{val\_loss} \leftarrow \infty$; $patience \leftarrow k$;

for $i \leftarrow 1$ *to* n **do**
  Train the model on $\mathcal{D}_{train}$ using backpropagation to update the parameters θ; Compute the training loss $\mathcal{L}(\mathcal{D}_{train}, \theta)$ and validation loss $L(\mathcal{D}_{val}, \theta)$;
  if $L(\mathcal{D}_{val}, \theta) < best_{val\_loss}$ **then**
    $best_{val\_loss} \leftarrow \mathcal{L}(\mathcal{D}_{val}, \theta)$; $patience \leftarrow k$; Save the current best model parameters as θ_{best};
  end
  else
    $patience \leftarrow patience - 1$;
    if $patience = 0$ **then**
      Stop training and return θ_{best};
    end
  end
end
Return the final parameters θ;

Note that in this algorithm, we introduce two additional variables: *best_val_loss* and as introduced previously the *patience* that will keep track of the lowest validation loss achieved so far, and the latter counts the number of consecutive epochs where the validation loss does not improve. When the patience runs out (i.e., when the validation loss fails to improve for k consecutive epochs), we stop training and return the parameters that achieved the lowest validation loss.

3.3.2 Batch, epochs and learning rate

As the *batch* size, the number of *epochs* and the *learning rate* are hyperparameters to be tuned in neural networks we will redefine the batch size as the number of epochs and the learning rate are known and inherent concepts. Actually, the *batch* size refers to the number of training samples that are processed in one iteration of the neural network before updating other model parameters. Mathematically, given a dataset of n training samples $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$, the dataset is divided into batches of size B as follows:

$$\begin{aligned}&\{(x_1, y_1), (x_2, y_2), \ldots, (x_B, y_B)\},\\ &\{(x_{B+1}, y_{B+1}), (x_{B+2}, y_{B+2}), \ldots, (x_{2B}, y_{2B})\},\\ &\{(x_{k\times B+1}, y_{k\times B+1}), (x_{k\times B+2}, y_{k\times B+2}), \ldots, (x_n, y_n)\}\end{aligned} \tag{3.29}$$

where $k = \frac{n}{B}$ and thus, for each iteration of the *backpropagation* algorithm, the model is trained on one batch of samples. This can be expressed mathematically as:

$$\widehat{\theta}_{t+1} = \widehat{\theta}_t - \eta \left[\frac{1}{B}\sum_{i=1}^{B} \nabla_{\widehat{\theta}_t} \mathcal{L}(f_{\widehat{\theta}_t}(x_i), y_i)\right] \tag{3.30}$$

where $\widehat{\theta}_t$ is the parameter values of the neural network at time step t including weights and biases and $f_{\widehat{\theta}_t}(x_i)$ represents the output or prediction of the neural network for the $i-th$ sample, given the current parameter values $\widehat{\theta}_t$.

For The LocalGLMnet architecture we will choose to implement a *full-batch* training that can be a useful approach in certain situations, particularly when working with small datasets or complex models as with full-batch training, the model can converge to a good solution in fewer iterations, as it is updating the weights based on the gradients computed on the entire training dataset. Moreover, it can provide more accurate estimates of the true gradient, as it reduces the noise caused by the stochastic nature of mini-batch gradient descent and thus can lead to a better estimate of the true distribution of the data, thus highest generalization performances on unseen data. We refer to the Geiping, Goldblum, Pope, Moeller, and Goldstein (2022) paper that advances that non-stochastic full-batch training can achieve comparably strong performance.

3.3.3 Processing and results

One of the main issues with neural network models is selecting the appropriate hyperparameters of the architecture. This includes the number of hidden neurons, the activation function, and other architectural choices as the selection of these hyperparameters can have a significant impact on the performances of the model. Thus recent research has suggested using ℓ_2*-penalized loss* with $ReLU$ activation function, which can lead to a sparse number of active neurons in the network as it penalizes large weights in the model, and can help prevent overfitting (Neyshabur, Tomioka, and Srebro (2015)). The $ReLU$ activation function has also been found to have useful properties for this purpose, including the ability to induce sparsity in the model and is defined as:

$$f(x) = max(0, x) \tag{3.31}$$

Furthermore, these studies have shown that the number of active neurons can be optimized by using a sufficiently large number of epochs during model training. This is because the $ReLU$

activation function encourages the network to be sparse, and this sparsity can be optimized over time with the right number of training epochs. Then, we will find our hyperparameters such as the number of neurons q on the hidden layer, the number of epochs and the ℓ_2-*regularization* λ parameter by grid-search and cross-validation that is a procedure that can be used to estimate the generalization error of a model. Actually, by splitting the data into training, validation, and test sets and training the model on different folds of the data, we can get a better estimate of the model's performance on new data. For the case of the single hidden layer neural network also known as the *plain vanilla neural network*, we will use the built-in R package *nnet*, which means that it does not require any external installation or configuration and is also relatively easy to use and has a simple interface but later we will need the *keras* and *tensorflow* that are more flexible libraries. Moreover, for our hyperparameter tuning, we will need more efficient algorithms that converge more quickly to the optimal combination of hyperparameters in what follows.

K	λ	$D_{\mathcal{D}}$	$D_{\mathcal{T}}$
1	1e-09	30354.376	7380.0961
10	1e-09	20367.816	5614.442
25	1e-09	18511.12	5378.152
35	1e-09	18209.761	5523.64
50	1e-09	17824.220	5508.117
75	1e-09	17521.015	5628.511
100	1e-09	16854.435	5740.945
1	1e-08	23497.145	5875.080
10	1e-08	21253.695	5645.8105
25	1e-08	18681.285	5613.36230
35	1e-08	18395.855	5536.585
50	1e-08	17484.855	5797.810
75	1e-08	17351.390	5779.548
100	1e-08	17059.0429	5924.340
1	1e-07	30319.492	7373.480
10	1e-07	20821.234	5516.072
25	1e-07	18779.732	5471.682
35	1e-07	17903.484	5608.8144
50	1e-07	18173.395	5619.255
75	1e-07	17156.515	5785.283
100	1e-07	16838.917	5933.79195
1	1e-06	30043.347	7293.982
10	1e-06	19987.214	5512.273
25	1e-06	18561.357	5472.217
35	1e-06	18133.722	5549.092
50	1e-06	17973.0996	5781.1953
75	1e-06	17434.6406	5764.6265
100	1e-06	17132.5938	5769.2822

K	λ	$D_{\mathcal{D}}$	$D_{\mathcal{T}}$
1	1e-05	30003.6094	7283.7388
10	1e-05	21071.6973	5459.9966
25	1e-05	18601.7637	5697.3218
35	1e-05	18236.6992	5617.2124
50	1e-05	17750.3203	5549.0771
75	1e-05	17225.5898	5709.3975
100	1e-05	17281.6699	5695.5767
1	1e-04	22345.6270	5721.4556
10	1e-04	19902.7832	5576.3037
25	1e-04	18306.4219	5471.3232
35	1e-04	18236.1621	5569.2593
50	1e-04	17770.3848	5669.9736
75	1e-04	17329.8770	5776.6162
100	1e-04	17054.2305	5709.6162
1	1e-03	29984.6523	7278.5215
10	1e-03	20825.3457	5515.0996
25	1e-03	18394.9336	5636.4150
35	1e-03	18296.0352	5525.5381
50	1e-03	18379.2129	5555.6836
75	1e-03	17400.1777	5684.1733
100	1e-03	16959.9766	5706.0639
1	1e-02	30100.1426	7391.9111
10	1e-02	19557.8574	5522.1572
25	1e-02	17992.9375	5526.4629
35	1e-02	17974.0977	5408.6924
50	1e-02	17509.5996	5475.152
75	1e-02	17399.645	5483.367
100	1e-02	17074.025	5464.736

Table 3.3: Results for single-layer NN with different and K, λ

3.4 Calibration

Calibration of a model loss in relevant subgroups is an important step in evaluating the performance of a predictive model. The *Hosmer-Lemeshow* test is one method that can be used to assess calibration in the case of predicting the default probability of loans and we will apply this calibration test in the spirit of what has been done by Dimitriadis, Dümbgen, Henzi, Puke, and Ziegel (2022).

The Hosmer-Lemeshow test is a goodness-of-fit test that compares the observed frequencies of a binary outcome variable with the expected frequencies based on the predicted probabilities from a logistic regression model. The test is performed by dividing the data into a number of equally sized subgroups based on the predicted probabilities, and then comparing the observed and expected frequencies within each subgroup. In fact, let n be the number of observations, p_i be the predicted probability of the binary outcome variable for the ith observation, o_i be the observed outcome for the ith observation (0 or 1), and g_i be the group membership of the ith observation, where $g_i \in 1, 2, ..., G$ and G is the number of subgroups. Let n_g be the number of observations in subgroup g, O_g be the total number of observed outcomes in subgroup g, and E_g be the total number of expected outcomes in subgroup g based on the predicted probabilities.

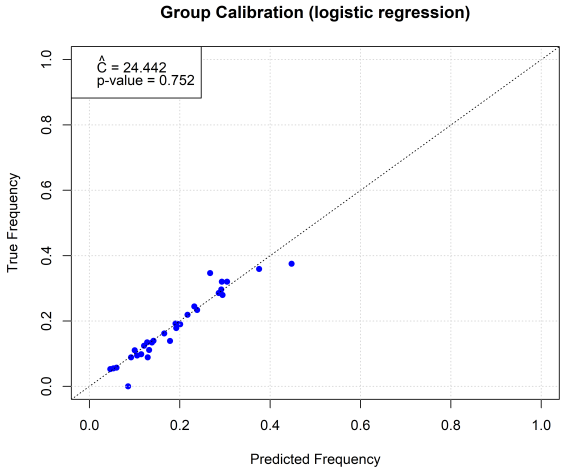
The expected outcomes E_g are calculated as follows:

$$E_g = \sum_{i=1}^{n} p_i \mathbf{I}(g_i = g) \tag{3.32}$$

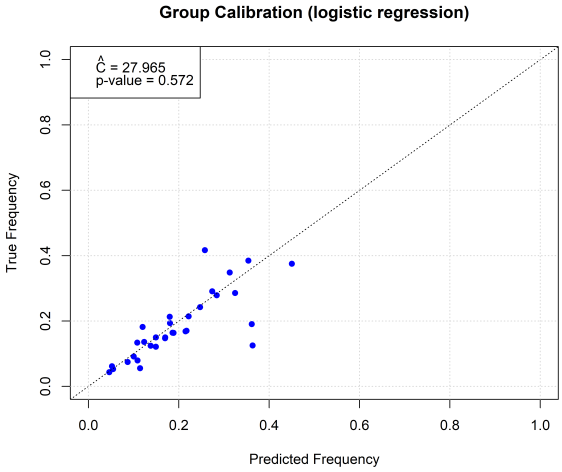
where $\mathbf{I}(g_i = g)$ is an indicator function that equals 1 if $g_i = g$ and 0 otherwise. Thus, the Hosmer-Lemeshow test statistic is calculated as follows:

$$\widehat{C} = \sum_{g=1}^{G} \frac{(O_g - E_g)^2}{E_g(1 - E_g/n_g)} \tag{3.33}$$

Under the null hypothesis of perfect calibration, the test statistic follows a chi-squared distribution with $G - 2$ degrees of freedom. The $p - value$ for the test is calculated based on this distribution. By assessing calibration in this way, we can identify areas where the model may be under- or over-predicting outcomes and take steps to improve its performance. We will use the resulting $p - value$ of the test and an observed/predicted plot to analyze the quality of our calibration. We plot the calibration in subgroups:

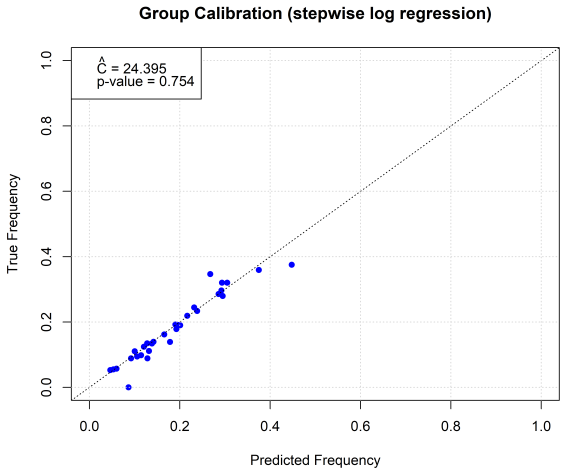


(a) Calibration in training set $\mathcal{D}$

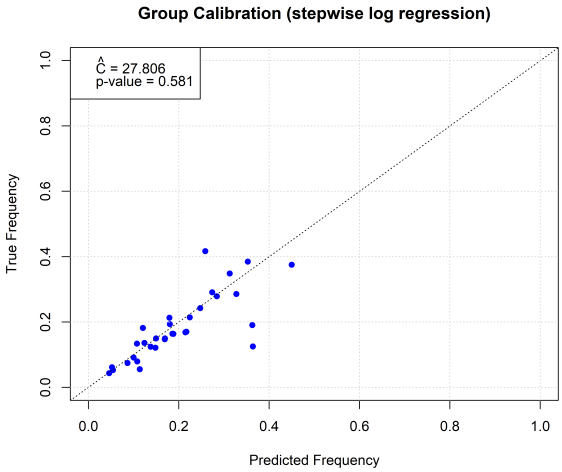


(b) Calibration in testing set $\mathcal{T}$

Figure 3.5: Calibration for logistic regression



(a) Calibration in training set $\mathcal{D}$



(b) Calibration in testing set $\mathcal{T}$

Figure 3.6: Calibration for stepwise logistic regression

Thus, we can assess for the appropriate calibration of our models because for both we can't reject the null hypothesis with a confidence level of 5% that advances that the observed and predicted frequencies are not the same in subgroups. We recall that the division in subgroups with same characteristics has been performed as in the context of predicting default probabilities for groups of policies with similar individual behaviour, calibration is a useful tool for ensuring that the model's predictions are accurate and reliable because policies with similar feature values are likely to have similar levels of risk. By calibrating the model's predictions to match observed

default rates within these groups, actuaries can better estimate the likelihood of default for each policy group, which can in turn help insurers more accurately price their policies and manage their risk. Actually, policyholders with a higher probability of defaulting pose a greater financial risk to the insurer, and as a result, they may be charged higher premiums. Moreover, by examining policyholders with similar characteristics, actuaries can determine the appropriate pricing structure and underwriting guidelines to ensure the financial stability of the insurance company. As said previously, insurance companies are subject to various regulations and requirements, which may include maintaining a certain level of solvency and capital adequacy and predicting default probabilities allows actuaries to evaluate the financial health and stability of the company, ensuring compliance with regulatory standards. Accurate estimations of default probabilities in subgroups also assist in stress testing scenarios to assess the impact on the insurer's financial position under adverse conditions. Finally, calibration enable insurance companies to develop strategic plans and make informed business decisions. By understanding the potential default risks associated with different policyholder groups, insurers can tailor their marketing efforts, product offerings, and risk management strategies to maximize profitability and mitigate potential losses.

For instance, it's common to prefer *two-step pricing* models that allows insurers to balance the need for efficiency in pricing with the desire to account for individual risk differences among policyholders. By initially grouping policyholders with similar characteristics and then refining the pricing on an individual basis, the model helps ensure fairness and adequacy in premium calculations while also managing administrative complexity. We refer to the following paper Grari, Charpentier, and Detyniecki (2022), which develops an efficient prediction method following this logic, especially thanks to the *Autoencoder*

Chapter 4

The LocalGLMnet architecture

Machine learning techniques have revolutionized the field of modeling by offering the potential for highly accurate predictions. However, these approaches have been criticized for being difficult to interpret, particularly when used in deep learning models such as neural networks, which are often compared to black boxes due to the internal weight learning from input features. As a result, the effect of each variable on the final rating decision can be hard to interpret as in contrast, *generalized linear models* are easier to interpret than deep learning models as they typically model the relationship between input variables and output predictions through a linear function, and the coefficients of this function provide a direct interpretation of the relationship between each input variable and the output that makes the understanding of input features relevancy analyis straightforward. Furthermore, while *generalized linear models* are easier to interpret, they may not always provide the same level of accuracy compared to deep learning models, in particular for deep learning models such as neural networks that can be more accurate in predicting the likelihood of default than logistic regression for example. To address this, we will introduce the modern architecture of the *localGLMnet*, which combines the interpretability of GLMs with the accuracy of deep learning models, allowing for interpretation such as feature selection and interaction analysis. We will compare the results obtained with logistic regression and single hidden layer neural network in the previous sections to the performances of the implementation of this architecture, to assess for the growing concern around it.

The main objective of this thesis will be to demonstrate that the architecture of this neural network makes it fully interpretable. We will show that its internal construction allows for feature selection and interaction analysis, making it easier to identify which input variables are relevant and how they affect the output. In particular, we will demonstrate how we can use the architecture of the network to perform; as said, feature selection and interaction analysis, enabling us to gain a deeper understanding of the underlying data. Finally, we will explore the possibility of extending this architecture by performing embedding mapping to categorical and

text features, further increasing its interpretability. By doing so, we aim to offer a new way of understanding and interpreting deep learning models, which will enable these models to be used more widely in areas where the combination of interpretability and high trainable performances are essential.

4.1 Architecture

As we have briefly introduced some concepts in the Neural Networks section, we will explain the modern architecture of the *localGLMnet*. Actually, this architecture is a machine learning model proposed by Richman and Wüthrich (2021) for spatially correlated data analysis. It is an extension of the popular generalized linear models that allows for the incorporation of spatial information into the modeling process for huge input matrix from tabular data's. It uses a two-stage approach to modeling the data as first, a global model is fitted to the data, which captures the overall trends and patterns in the data and in the second stage, a local model is fitted to each spatially adjacent group of observations, which captures the local variations in the data that are not captured by the global model. Thus, the implementation follows the *attention* pattern what as been introduced by Bahdanau, Cho, and Bengio (2016) that is a mechanism that allows the network to focus on certain parts of the input data, depending on the relevance of each part to the task at hand. In the proposal, this is done internally, constructing a fully connected neural network that form the basis of the regression attention weight construction. Then, a skip connection is used to combine the learned regression attention of last layer and the untransformed input features from the input layer that should be of the same sizes in order to give a linear modeling part around which the network model is built. Finally, the result enter à last neuron that contains a link activation function comparable to the link function g of the *generalized linear model* that provides the prediction for the mean $\mu(x)$. And then, the before-last layer that compute the dot product between the regression attention coefficients and the inputs features may be used after training as an intermediary component that allows for feature selection and interaction analysis that will be subject to gradient and spline fitting.

Thus, it differs with what has been done with methods like *PDP*, *LIME*, *ALE* or *SHAP* recently as here the internal structure allows for interpreting, explaining, general interactions and benefiting from the transparent β interpretation of *generalized linear models*.

4.1.1 FFN extension of GLM

A *fully connected feedforward* neural network is a type of neural network in which the information flow is unidirectional, from the input layer to the output layer, every neuron in each layer being connected to every neuron in the next layer. Overall, such a neural network can be represented

as a union of affine transformations followed by nonlinear activation functions, as shown below:

$$f(x) = \phi_{W_L,b_L} \circ \sigma \circ \phi_{W_{L-1},b_{L-1}} \circ \sigma \cdots \circ \phi_{W_1,b_1} \tag{4.1}$$

where $\phi_{W_l,b_l}(x) = W_l x + b_l$ is the affine transformation of layer l with $W^{(l)} \in \mathbb{R}^{m_l \times m_{l-1}}$ and $b^{(l)} \in \mathbb{R}^{m_l}$ are the weight matrix and bias vector for layer l, respectively, and σ is the nonlinear activation function. Thus, we can generalize this expression for the $m-th$ layer of the fully-connected feed forward neural network, where raw features are first non-linearly transformed before performing the scalar product of the GLM such that

$$z_j^{(m)}(x) = \phi_m(w_{0,j}^{(m)} + \sum_{l=1}^{q_{m-1}} w_{l,j}^{(m)} x_l) \tag{4.2}$$

for neurons $z_j^{(m)}(x)$, $1 \leq j \leq q_m$, q_m being the number of neurons on layer m. Let's point out that we replace the bias notation $b^{(l)}$ by $w_0^{(l)}$ as it's more clear for what follows. Thus with the formulation 4.1, for a *fully connected feedforward* neural network with d layers, the last layer output is given by

$$x \mapsto z^{(d:1)}(x) = (z^{(d)} \circ ... \circ z^{(1)})(x) \tag{4.3}$$

that provides a deep representation of the set of inputs x_i, $1 \leq i \leq q_0$, q_0 being the length of the input vector.

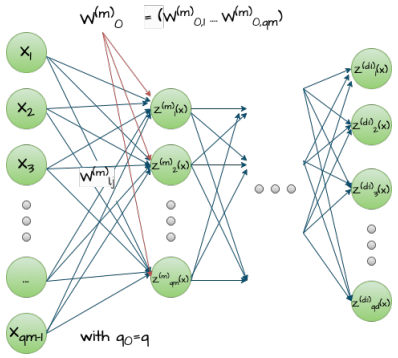


Figure 4.1: Fully-connected feed forward neural network architecture

What becomes really interesting is the way we will retain the learned representation of the input features in the neural network architecture. Actually, retaining the values of learned representation of inputs in a specific layer in neural network refers to the process of preserving the outputs of one layer in a neural network during training or inference, rather than discarding them. These activations are often referred to as the *hidden representations* of the inputs, as they

capture the internal structure and features of the input data that are relevant to the task at hand. Thus, it will be more clear looking at the following graph representation but as we go further in our explanations, the architecture will retain this input representation in the before last layer of the neural network. We still maintain our notation $z^{(d:1)}(x) \in \mathbb{R}^{q_d}$ for this layer as the last one is only applying the link function at each iteration to link the *generalized linear models* theory. Thus, this is the way the last neuron provides our outputs while training:

$$x \mapsto g(\mu(x)) = \beta_0 + \langle \beta, z^{(d:1)}(x) \rangle \tag{4.4}$$

Now that the way the training is performed is more clear, we still face a lack of interpretation power as feature's importance is of main concern in actuaries. Furthermore, as we face the black-box interpretation issue of deep networks architectures, the x_j influence on the response $\mu(x)$ becomes not clear for local individual and global sample interpretations. The *LocalGLMnet* architecture will solve this issue by switching from *regression parameter* β_j into network learned *regression attention* $\beta_j(x)$ that is features dependent for the *generalized linear models* expression such that:

$$x \mapsto g(\mu(x)) = \beta_0 + \langle \beta(x), x \rangle \tag{4.5}$$

where $\beta(x) = z^{(d:1)}(x)$ that in a small environment $\mathbb{B}(x)$ allows us to approximate $\beta(x')$, $x' \in \mathbb{B}(x)$ by a constant regression parameter leading to local GLM interpretation. But when the coefficients $\beta(x)$ are features dependent, it means that the relationship between the predictor variables and the response variable is not linear any more, then the effect of a change in one predictor variable on the response variable depends on the values of the other predictor variables.

This breaks the assumption of linearity in a GLM, which assumes that the relationship between the predictor variables and the response variable is constant across all values of the predictor variables. When this assumption is violated, the GLM is no longer an appropriate model. However, if the dependence of the coefficients on the predictor variables is smooth, meaning that the coefficients vary smoothly across the predictor space, we can still use a sort of *local GLM* and estimate a separate *GLM* for each region of the predictor space where the coefficients are approximately constant. In fact, this approach is applied in the *generalized additive model* and can be a useful way of modeling non-linear relationships between predictor variables and response variables.

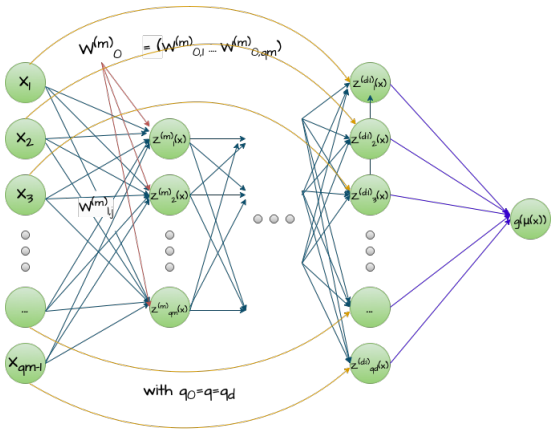


Figure 4.2: Deep LocalGLMnet architecture

Moreover, $\beta_j(x)$ may be interpreted as attention given to x_j and thus to maintain this relationship it's essential that the number of neurons on the before last layer $z^{(d:1)}(x)$ is such that $q_0 = q_d = q$. Finally, a *LocalGLMnet layer* can be defined by replacing the scalar product by a component-wise product such that:

$$x \mapsto \beta^{(1)}(x) \otimes x = (\beta_1^{(1)}(x)x_1, ..., \beta_q^{(1)}(x)x_q)^\mathsf{T} \tag{4.6}$$

and similarly the definition of the *deep LocalGLMnet* is obtained by combining such layers, an example with two layers:

$$x \mapsto g(\mu) = g(\mu(x)) = \beta_0^{(2)} + \langle \beta^{(2)}(\beta^{(1)}(x) \otimes x), x \rangle \tag{4.7}$$

is the total number of features (including both numeric and categorical features). By taking the Hadamard product of the outputs of two separate linear transformations of the input x, we effectively create an interaction term between the input features. This can be seen as a way of allowing the neural network to model more complex relationships between the input features. For example, if we consider the case where x has two features, x_1 and x_2, the Hadamard product operation between the outputs of two separate linear transformations can be written as:

$$\begin{aligned} &\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \otimes \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \odot \begin{bmatrix} w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} \otimes \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_3 \\ b_4 \end{bmatrix} \\ &= \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + b_1 \\ w_{21}x_1 + w_{22}x_2 + b_2 \end{bmatrix} \odot \begin{bmatrix} w_{31}x_1 + w_{32}x_2 + b_3 \\ w_{41}x_1 + w_{42}x_2 + b_4 \end{bmatrix} \end{aligned} \tag{4.8}$$

This will display the equation in two lines, with the first line containing the matrix multiplication terms and the second line containing the element-wise multiplication terms..

4.1.2 Interpretability & feature selection procedure

In addition, the *LocalGLMnet* solves the explanation problem of the *Shapley values* extension for deep learning models as it directly postulates an additive decomposition for $\mu(.)$ after applying the link function. Thus, the gradients $\nabla\widehat{\beta}_j(x)$ for $1 < j < q$ allows to study the derivative of regression attention with respect to x_k for fixed j with $\nabla\widehat{\beta}_j(x_i)$, $1 < i < n$ performing *spline fits* to the sensitivities by regressing :

$$\partial x_k \widehat{\beta}_j(x_i) \sim x_{i,j} \tag{4.9}$$

$\beta_j(x) \equiv 0$	$\beta_j(x) \equiv \beta_j$	$\beta_j(x) \equiv \beta_j(x_j)$	$\beta_j(x) \equiv x_{j'}/x_j$
Drop x_j	GLM term in x_j, not feature dependent	No interactions of x_j with $x_{j'}$ $j \neq j'$	Not full interpretability in model identifiability

Table 4.1: Interpretation of $\beta_j(x)x_j$

Moreover, as it's clear that if $\beta_j(x) \equiv \beta_j$ we come back to *GLM* theory as the regression attention is not features dependent, for the case where $\beta_j(x) \equiv 0$ we will need statistical testing to validate our hypotheses. In fact, if the values of regression attention $\beta_j(x)$ fluctuate around 0 for different values of x_j we can use an empirical test for determining the rejection region of the regression attentions. This will be done by introducing a completely random additional input feature x_{q+1} that does not have any influence on the output $g(\mu(x))$ and independent of x, instead of the traditionally *likelihood ratio test* or *Wald test*. We will recall the statistical background of these two tests and why there are not suitable for our architecture.

Likelihood ratio test

The likelihood ratio test is a statistical test used to compare two nested models, with one that is a restricted version of the other and thus test the null hypothesis that the restricted model is a good fit for the data, while the alternative hypothesis suggests that the unrestricted model leads to a better fit. If we denote L_{Full} as the log-likelihood of the full model and L_{Rest} the log-likelihood of the restricted one we can prove that:

$$2(L_{Full} - L_{Rest}) \sim \chi^2_d \tag{4.10}$$

where d is the difference in the number of parameters between the two models. As the two models are nested we can write the likelihood ratio statistic as:

$$LR(x) = \frac{\max_{\theta \in \Omega_0} \mathcal{L}(\theta|x)}{\max_{\theta \in \Omega} \mathcal{L}(\theta|x)} \tag{4.11}$$

where the numerator only maximize the likelihood function by selecting parameters in the restricted subset Ω_0. Thus it is clear that if the set of parameter chosen at optimum are in the bounded interval Ω_0 then $LR = 1$, since maxed likelihoods will be equal. Thus what's interesting is that with the following expression we can link the likelihood ratio with the chi-squared statistic as:

$$-2log(LR(x)) \xrightarrow{D} \chi^2_d \tag{4.12}$$

To perform this test, we first estimate the parameters of both the restricted and unrestricted models using maximum likelihood estimation and then, we compute the likelihood ratio test statistic and compare it to the critical value from the chi-squared distribution at the significative level we choose (usually 5% or 1%). We refer to Lehmann (2006) which explores the superiority of integrated over maximum likelihood if you want to understand why it's not always appropriate.

But as it is obvious that the *maximum likelihood estimation* is the key of this ratio estimation it will not suitable for the *LocalGLMnet* due to the early stopping of this architecture. Moreover the *MLE* is very sensitive to the choice of the initialized parameters and this may affect the volatility of regression attentions $\widehat{\beta}_j(x)$ as it may exhibit different degrees of sensitivity to changes in the input features or model parameters.

Wald test

The *Wald test* is a statistical method used to evaluate constraints on statistical parameters by measuring the weighted distance between an unrestricted estimate and a hypothesized value under the null hypothesis, where the weight represents the precision of the estimate where a larger weighted distance indicates a lower likelihood that the constraint is true. However, this test has a significant drawback in that it is not invariant to changes in the representation of the null hypothesis, leading to nontrivial differences in the corresponding Taylor coefficients.

If we define $\widehat{\theta}$ the parameter found by *MLE* and θ_0 the value of this parameter under the null hypothesis H_0 then we formulate the following expression for the *Wald* statistic:

$$W = \frac{(\widehat{\theta} - \theta_0)^2}{\text{Var}(\widehat{\theta})} \tag{4.13}$$

and taking the square root of this static gives us a pseudo *t-ratio*. The previously displayed

Wald statistic is following a chi-squared distribution χ^2_1 with one degree of freedom.

However, the *Wald test* assumes that the predictor features in the model are not highly correlated with each other and thus doesn't allow for multicollinearity. When this assumption is not verified then the standard errors of the estimates may be inflated, leading to incorrect conclusions.

4.1.3 Spline fitting

Spline fitting is a method used to approximate a curve or a function by a piecewise polynomial function, where each polynomial is defined on a subinterval of the original domain. The subintervals are called *knots* and the polynomial function is chosen to be smooth and continuous at each knot, the main objective being to find a function that accurately represents the data, while also being simple and easy to interpret.

More formally, let $x_1, x_2, \ldots, x_n$ be a set of n data points, and let $y_1, y_2, \ldots, y_n$ be the corresponding values of a function $f(x)$. To achieve the goal of spline fitting we try to find a function $S(x)$ that approximates $f(x)$ using a piecewise polynomial function of degree d on each knot interval $[x_{i-1}, x_i]$, $i = 1, 2, \ldots, n$. The function $S(x)$ is defined by:

$$S(x) = \begin{cases} S_1(x) & x_1 \leq x < x_2 \\ S_2(x) & x_2 \leq x < x_3 \\ \ldots & \\ S_{n-1}(x) & x_{n-1} \leq x \leq x_n \end{cases} \tag{4.14}$$

As said previously, the polynomial functions are chosen to be smooth and continuous at each knot, which means that the function and its derivatives up to order $d-1$ are continuous at each knot. Moreover, the choice of polynomial degree d and the number of knots depend on the data and the desired level of accuracy as higher degrees and more knots can lead to a more accurate representation of the data, but can also result in overfitting and a more complex model.

To find the polynomial functions $S_i(x)$, spline fitting typically involves solving a system of linear equations that ensure the smoothness and continuity of $S(x)$ at each knot. This system of equations is known as the *spline interpolation* problem, and its solution gives the coefficients of the polynomial functions.

In the context of the *LocalGLMnet*, spline fits are used to regress the sensitivities $\partial x_k \widehat{\beta}_j(x_i)$

with respect to $x_{i,j}$ for fixed j. This means that a spline function is used to approximate the relationship between the sensitivities and the input variables, allowing for the analysis of the derivative of regression attention with respect to each input variable.

4.2 Loss Function

As it has been already determined, in the case of the **lending club** dataset, we get only one loan per *id*, so that we have set the exposure $w_i = 1$. When we have to deal with a binary target such as the *loan_status* the naive approach in our case would be to choose implementing a classifier with the following cross-entropy loss function:

$$\mathcal{L}(y_i, p_i) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{q} y_{i,k} \cdot \log(\widehat{y_{i,k}})$$

where $\widehat{y_{i,k}} = \mathbf{P}[y_{i,k} = 1]$, thus that observation i is in class k.

To derive the formula for the binomial deviance, we start with the likelihood function of a logistic regression model with binary response variables:

$$\mathcal{L}(\beta; y_i, x_i) = \prod_{i=1}^{n} p_i(y_i; \beta, x_i)^{y_i}(1 - p_i(y_i; \beta, x_i))^{1-y_i} \tag{4.15}$$

where $p_i(y_i; \beta)$ is the probability of success for the ith observation, which depends on the values of the predictors x_i and the coefficients β. The logarithm of the likelihood function is given by:

$$L(\beta; y_i, x_i) = \sum_{i=1}^{n} y_i \log p_i(y_i; \beta, x_i) + (1 - y_i)\log(1 - p_i(y_i; \beta, x_i)) \tag{4.16}$$

We can use the predicted probabilities $\widehat{p}_i$ from the model to estimate the probability function $p_i(y_i; \beta)$:

$$p_i(y_i; \beta, x_i) = \begin{cases} \widehat{p}_i & \text{if } y_i = 1 \\ 1 - \widehat{p}_i & \text{if } y_i = 0 \end{cases} \tag{4.17}$$

Now starting from what we advanced in the section we get the following formula for the deviance formula:

$$D = 2(L(\widehat{\beta}_{H_1}; y) - L(\widehat{\beta}_{H_0}; y)) \tag{4.18}$$

and we can replace in this expression the two formula for the likelihood function of the restricted and saturated model such that:

$$L(\widehat{\beta}_{H_1}; y) = \sum_{i=1}^{n} \left\{ y_i \log(\frac{y_i}{1-y_i}) + \log(1-y_i) \right\} \tag{4.19}$$

$$L(\widehat{\beta}_{H_0}; y) = \sum_{i=1}^{n} \left\{ y_i \log(\frac{\widehat{p}_i}{1-\widehat{p}_i}) + \log(1-\widehat{p}_i) \right\} \tag{4.20}$$

and finally by replacing this two expressions in the deviance formula we get:

$$D = 2 \sum_{i=1}^{n} \left\{ y_i \log \left(\frac{y_i}{\sigma(x_i^\top \widehat{\beta})} \right) + (1-y_i) \log \left(\frac{1-y_i}{1-\sigma(x_i^\top \widehat{\beta})} \right) \right\} \tag{4.21}$$

$$D = 2 \sum_{i=1}^{n} \left\{ y_i \log(y_i) - y_i \log(\sigma(x_i^\top \widehat{\beta})) + (1-y_i) \log(1-y_i) - (1-y_i) \log(1-\sigma(x_i^\top \widehat{\beta})) \right\} \tag{4.22}$$

where the expected probabilities are estimated by regression and passed trough the sigmoid function σ in our output layer. This is exactly the same loss function as in Neural Networks.

4.3 Gradient descent

We start with a basic feedforward neural network that takes an input vector $\boldsymbol{x}$ and produces an output $\widehat{y}$, with weights and biases represented by the parameter vector $\boldsymbol{\theta}$. As we have seen previously, the network applies a series of linear transformations and nonlinear activations to the input to generate its output. In order to train the neural network to make accurate predictions, we need to adjust the values of the weights and biases so that the output is as close as possible to the true target values y. We can measure the error of the network's predictions using a loss function $\mathcal{L}(\widehat{y}, y)$, in our case the *binomial deviance*. Our goal is to minimize this loss function with respect to $\boldsymbol{\theta}$.

One approach for the minimization of the loss function is gradient descent. The idea behind gradient descent is to iteratively adjust the parameters in the direction of steepest descent of the loss function. In other words, we want to move the parameters in a way that reduces the loss function as quickly as possible. To do this, we need to compute the gradient of the loss function with respect to the parameters that tells us the direction in which the parameters should be adjusted to reduce the loss function. We already explain this concept previously but in the case of a neural network, the gradient is computed using the *backpropagation* algorithm.

Remember that the *backpropagation* algorithm starts with the output of the neural network and works backwards through the layers of the network, computing the gradient of the loss function with respect to the inputs to each layer by the way of the *chain rule*. The gradient

of the loss function with respect to the output of the network is computed first, and then this gradient is propagated backwards through the layers of the network to compute the gradients of the loss function with respect to the inputs to each layer. Once we have computed the gradient of the loss function with respect to the parameters of the network, we can update the parameters using the following update rule:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\widehat{y}, y) \tag{4.23}$$

Here, $\boldsymbol{\theta}^{(t)}$ represents the parameter values at iteration t, η is the learning rate (a hyperparameter that controls the step size of the update), and $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\widehat{y}, y)$ is the gradient of the loss function with respect to the parameters. We repeat this process of computing the gradient and updating the parameters for a fixed number of iterations or until convergence is achieved (i.e., the change in the loss function between iterations is below some threshold). Here's the algortihm in pseudo code:

Algorithm 6 Gradient Descent Algorithm

Require: Learning rate η, numb. of iterations N, cost function $\mathcal{L}(\theta)$, init. parameter values θ_0

1: Initialize θ_0
2: **for** $i \leftarrow 1$ to N **do**
3: Compute gradient of cost function: $\nabla \mathcal{L}(\theta_i)$; Update parameters: $\theta_{i+1} \leftarrow \theta_i - \eta \nabla \mathcal{L}(\theta_i)$
4: **end for**
5: **return** θ_N

And thus in this algorithm, the input variables are the learning rate η, the number of iterations N, the loss function $\mathcal{L}(\theta)$, and the initial parameter values θ_0 that provides the output that is the optimized parameter values θ_N. The algorithm first initializes the parameters with the initial values θ_0 and then, for each iteration from 1 to N, the gradient of the cost function is computed using the current parameter values, and the parameters are updated by subtracting the learning rate times the gradient. We refer to the paper of J. Lu (2022) that explains this process.

Overall, gradient descent is a powerful and widely-used optimization algorithm that is essential for training neural networks. By iteratively adjusting the parameters in the direction of steepest descent of the loss function, we can gradually improve the accuracy of the network's predictions.

4.4 Preprocessing, processing and results

4.4.1 Treatment of categorical feature

For this preprocessing part we will not use the *dummy* encoding for features with more than two levels as it's usually done for modelling *generalized linear models*. In fact, if we assume that

the feature x_j is categorical with respectively k levels, dummy coding selects a reference level and maps the levels that are different from the reference level to the $k-1$ unit vectors of the Euclidean space $\mathbb{R}^{k-1}$. Thus, *dummy* encoding produces design matrices of full rank, which is a crucial requirement for *generalized Linear Models*, but we point out that this is not a property needed for the *localGLMnet* implementation. On the other hand, one-hot encoding does not result in full rank design matrices as there is one redundancy in this encoding. However, this redundancy is actually necessary for certain applications of the *localGLMnet*.

4.4.2 Treatment of continuous feature

For the *localGLMnet* construction we will rely on two different scaling as we want our model to be comparable to what we have done before we will consider *min-max scaling* but we will have to build a model that is related to what Richman and Wüthrich (2021) have advanced and obtained, especially for the feature selection part of this thesis. Both scaling methods will be implemented and the comparison of results will be done.

4.4.3 In-sample and out-of-sample

In order to conduct a reliable analysis of how well the model generalizes to new data, it's essential to split our data into two sets: a learning set $\mathcal{D}$ (*in-sample*) and a test set $\mathcal{T}$ (*out-of-sample*). For consistency and comparability across different depth in our architecture, we will maintain the same partitioning throughout the analysis. The learning set will be used to train our model as usual (including implementing early stopping rules in gradient descent fitting), while the test set will only be used to assess for how well the model performs on new, previously unseen data.

We make the assumption that all observations/contracts are identically and independently distributed and after the partition of respectively 80/20% of our dataset for the *in-sample* and *out-of-sample* partition we can compute the average *default frequency* to assess if stratification is needed for the sampling process.

- *in-sample* frequency: 14.64% of defaulting contracts
- *out-of-sample* frequency: 14.36% of defaulting contracts

Moreover, our model will be trained using a validation split $\mathcal{V}$ during the model learning such that $\mathcal{D} = \mathcal{U} \cup \mathcal{V}$ where $\mathcal{U}$ is the effective part of our dataset that will be subject to training as $\mathcal{V}$ will be used to track early stopping using callbacks. Actually, as our validation split is disjoint from the test data $\mathcal{T}$, it will be helpful to assess for the performances of our model on unseen data while allowing for hyperparameter tuning. Additionally, callbacks will allows us to stop training the model when the performance on the validation set stops improving and thus this

helps prevent overfitting and improves the generalization ability of the model.

4.4.4 Hyperband hyperparameter tuner

As we want to optimize the implementation of the *localGLMnet* we will choose to use the state of art of hyperparameter tuning implementation that is a combination of a random search and a successive halving algorithm which aims to find the best set of hyperparameters for a given model in the shortest possible time and with the fewest resources. The *hyperband* implemented algorithm intoduced by L. Li, Jamieson, DeSalvo, Rostamizadeh, and Talwalkar (2018) works by first randomly sampling a set of hyperparameters for the model and training it for a fixed number of epochs. Thus, in this first step the performance of each model is evaluated trough the validation set and the algorithm selects the top-performing models and discards the poorly performing ones. The selected models are then trained for a different number of epochs, and the process of selection and training continues until only one model remains, that is what we called the *successive halving* that allows to speed up the process while staying in the best hyperparameters space. Before explaining parameters in details, we display the pseudo code algorithm:

Algorithm 7 Hyperband algorithm for minimizing binomial deviance with early stopping

Require: A machine learning model M, hyperparameter space H, maximum resource budget R, number of configurations to evaluate at each iteration n, binomial deviance D to be minimized, and early stopping callback with patience p

Initialize an array of configurations C with n random hyperparameter sets from H
For each configuration c in C, train M for a fixed fraction of the maximum budget R, using early stopping with a validation set and a patience of p, and record its Deviance score on the validation set
Determine the fraction of configurations f to keep for the next iteration based on a formula that takes into account the maximum budget R and the number of configurations n: $f = \lfloor n(R+1) \rfloor^{-1}$
while budget $r > 0$ **do**
    Select the top f configurations from C based on D
    Calculate the maximum budget for the selected configurations: $r_{max} = \frac{R}{f}$
    Train the selected configurations for a fraction of r_{max} using early stopping with a validation set and a patience of p, and record D
    Discard the bottom $1 - f$ fraction of configurations from C
    Replace the discarded configurations with new random configurations from H
    Decrement the budget: $r \leftarrow r - r_{max}$
end while
return The hyperparameter set with the lowest D on the validation set

Thus this algorithm aims to find the best set of hyperparameters and optimal number of epochs for a fixed batch size, combined with a *callback* parameter that allows for early stopping when convergence isn't reached and that provides an efficient tool to speed up the algorithm

even more. For the implementation we will use the *Hyperband* class of the *KerasTuner* API that allows for a wide range of customization parameter that we will describe below:

- *max_epochs*: This parameter corresponds to the maximum resource budget R in the algorithm and specifies the maximum number of epochs that the model will be trained for.
- *factor*: This parameter determines the fraction of configurations to keep for the next iteration f in the algorithm when pruning is performed.
- *objective*: This parameter specifies the objective to be optimized that will be the deviance loss, as our implementation extend the *keras.loss* the algorithm knows that it has to be minimized.
- *hyperband_iterations*: This parameter controls the number of times the it has to iterate over the full *Hyperband* algorithm such that it will reach $max\_epochs \cdot (\log_{factor}(max\_epochs))^2$ cumulative epochs across all trials for one hyperband_iterations.

We choose this approach for our parameter tuning after comparing performances for different implementation of hyperparameter space search such as *Random search* or *Bayesian search* for example and we conclude that the *Hyperband* one outperforms others in term of performances, convergence rate and efficiency. In fact, *Bayesian search* is a probabilistic approach that uses a surrogate model to predict the performances of different hyper-parameters and chooses the next hyper-parameters to evaluate based on their predicted performance. It also maintains a distribution over the hyper-parameters and updates this distribution as it receives new data to learn from. One main advantage of Bayesian optimization is that it can handle noisy or incomplete evaluations, which can occur in machine learning due to factors such as random initialization or small sample sizes. However, Hyperband has been shown to outperform Bayesian optimization in some cases, especially when the number of hyper-parameters is large and the evaluation of each configuration is relatively cheap. In addition, Hyperband has been shown to be more computationally efficient and scalable, which makes it a popular choice for hyper-parameter tuning in deep learning. A comparison of hyper-parameter tuning algorithms has been done by Y. Li et al. (2022) and advances that *Hyper-Tune* outperforms other hyper-parameter tuning systems on a wide range of scenarios.

4.4.5 Architecture implementation and results

As we have introduced the basic concepts of the *localGLMnet* and as we have explained our pre-processing, we will go deeper in the implementation of the architecture, the way we choose the hyper-parameters of the neural network by the previously introduced *Hyperband* tuning algorithm and we will try to justify as much as possible the choices we made for training of

our model. More precisely, in the section Neural Networks we already mentioned the fact that tuning hyper-parameters is an essential step in building a neural network using *Keras*. The performances and convergence rates of a neural network are highly dependent on the choice of hyper-parameters, such as the learning rate, batch size, number of layers, number of neurons in each layer, regularization parameters, and others.

Since the objective of our model is still to predict probability of default for contracts we don't change the sigmoid function defined by the equation 3.1 for the activation function $\phi_d = g$ of the the last layer d. Moreover, we will use a linear activation function in the attention layer that is related to the fact that the attention mechanism is essentially a weighted sum operation, where each input feature is multiplied by a weight value learned by the model. In this case, the linear activation function allows the weights to take on any real value, positive or negative, without being constrained by a non-linear activation function such as *ReLU* or *sigmoid*. For the hidden layers we still choose the *ReLu-$\ell_2$penalty* as it has been demonstrated in the Processing and results section of the Neural Networks part that sparsity induced by the ReLU activation function and the regularization provided by the the penalty can help to prevent overfitting and improve generalization performances.

Thus, in order to find the optimal parameters that best fit our data's we will be inspired by the multi-stage optimization procedure for the tuning of our deep neural network as it has been introduced by Tahmassebi (2018). In fact, the grid search will be performed in a two-stage by first focusing on the results for neural networks with different depths and number of neurons for a fixed learning rate (0.001), batch gradient descent ($nrow(\mathcal{D})$) and ℓ_2 weight regularization (0.01). Actually, as it's just a visual and experimental step to analyze the behaviour of our performances with respect to the depth and sizes of layers we will choose to set the length of the batch size equal to the number of rows of $\mathcal{D}$. This may lead to faster model evaluation as when the batch size is equal to the length of the training set, the model is evaluated once per epoch. This can speed up the evaluation process, allowing us to quickly test different hyperparameter combinations and identify which ones are worth exploring further. Thus, we will start by training a shallow network with just a few layers and gradually increase the number of layers trying for each depth a huge set of different number of neurons for all layers. We display the ranges for this two parameters:

- *depth*: [3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20]
- *number of neurons*: [5, 10, 15, 20, 30, 45, 50, 100, 128]

We display some results for the deviance convergences of some combinations for the train $\mathcal{U}$ and validation sets $\mathcal{V}$, with 200 epochs and *ReLu* activation function on the hidden layers:

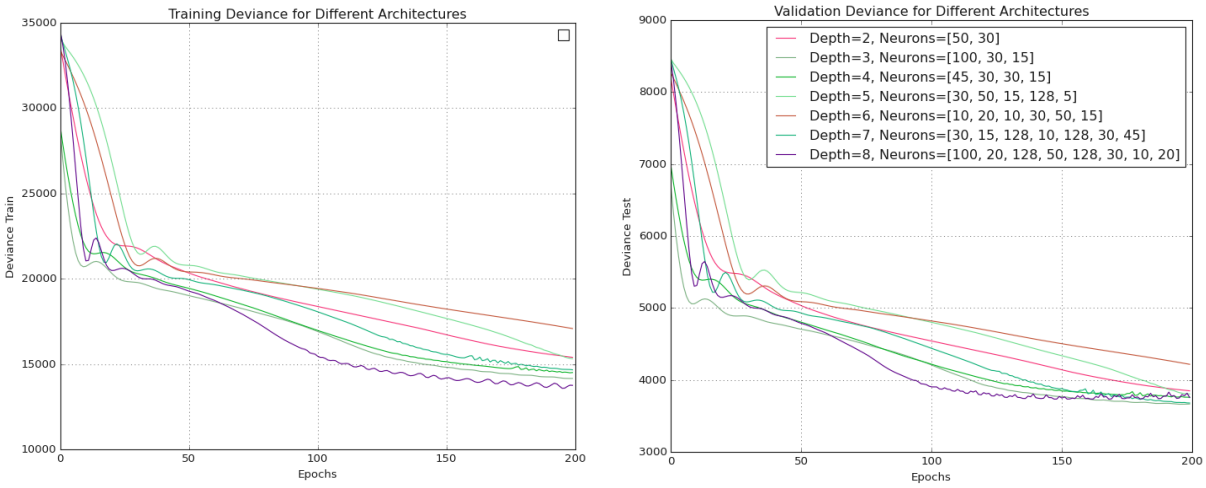


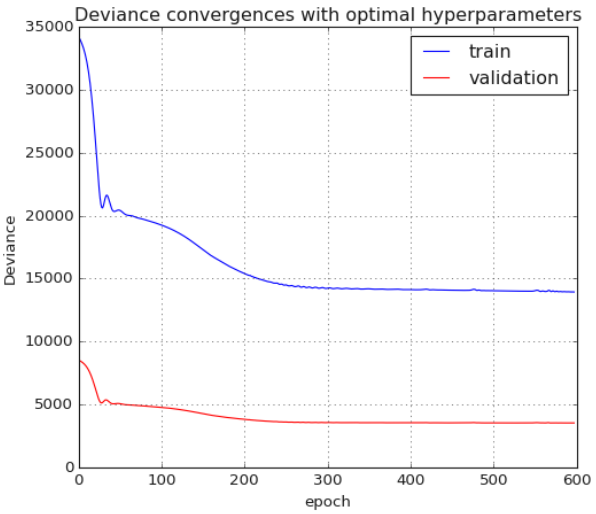
Figure 4.3: Depth convergences

The first statement that can be made is that as we look for signs of overfitting in the plots, such as a divergence between the train and validation loss curves that arises when the training loss continues to decrease while the validation loss starts to increase and that indicates that the model is memorizing the training data instead of generalizing well to new data, we do not notice this behaviour for all trials displayed on the plots. The same observation can be made for underfitting as a high train loss and a high validation loss that do not decrease significantly over time indicates that the model is not complex enough to capture the underlying patterns in the data and may benefit from additional layers, units, or other architectural changes. But as we observe that until the epoch 100 and 150, performances of neural networks with higher depths begin to fluctuate a lot that may indicate that the model has reached a point of saturation or instability in the training process, we will choose to tune other hyperparameters with a depth range between 2 and 6.

And now it's time to implement the *Hyperband hyperparameter tuner* as we reduce the layer loop to be able to reach a maximum depth of 6 layers and that is determined dynamically while tuning other hyperparameters. As said previously, this implementation is efficient to find the optimal values for a wide range of hyperparameters values for a fixed full-batch that we will thus set to $nrow(\mathcal{D})$ to allow each parameter update to be based on the entire dataset. We still conserve the *localGLMnet* architecture for the input, attention with linear activation function, dot and output with sigmoid activation function layers while setting a maximal number of epochs/budget $R = 1000$ with a factor $f = 3$. These have been set so that we can determine the maximum number of models n_M that can be trained in a single *Hyperband* iteration that is given by:

$$n_M = R(\log_f(R))^2 = 39535 \tag{4.24}$$

and that's why we implemented a callback with a patience $p = 20$ for preventing the validation loss to fluctuate too much. Finally, as the *Hyperband* hyperparameter tuner stops when either the budget of resources R is fully used or only one configuration remains, we display the results after convergence with the optimal configuration that has been retained because it leads to the lowest validation *binomial deviance* loss. We warn whoever wants to run the code that it can last for a long time depending on your performances, as we perform the search over a wide range of hyperparameter values.



(a) convergence plot

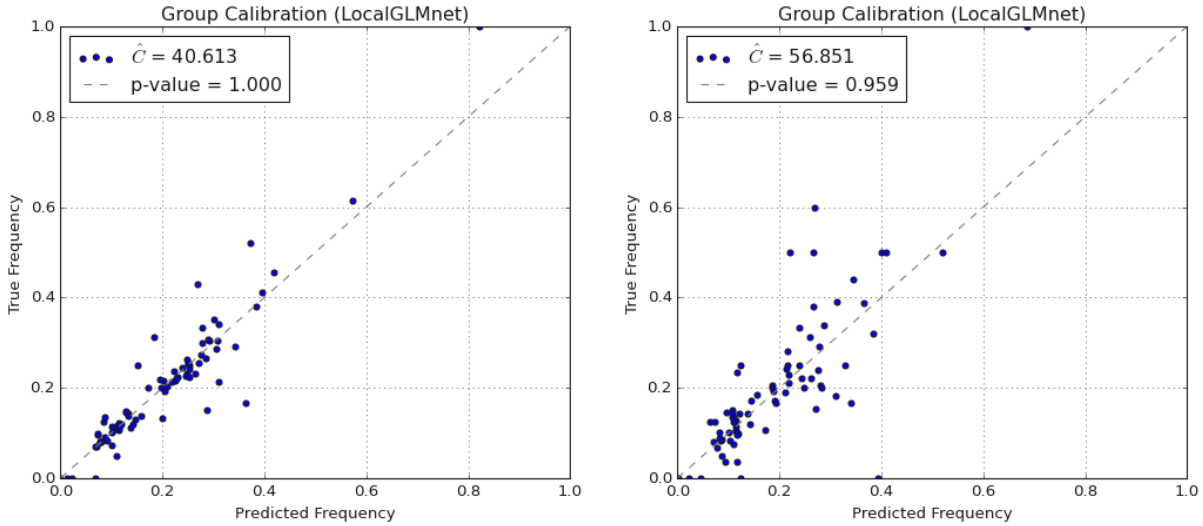
Hyperparameter	Value
num_layers	6
units_0	28
activation	relu
l2_reg_0	0.1
learning_rate	0.001
units_1	23
l2_reg_1	0.08
units_2	59
l2_reg_2	0.0
units_3	19
l2_reg_3	0.04
units_4	39
l2_reg_4	0.0
units_5	51
l2_reg_5	0.01
tuner/epochs	1000

(b) Optimal hyperparameters

Figure 4.4: LocalGLMnet hyperparameters optimisation

What must be emphasized in the analysis of the results is that the architecture chosen for the deep neural network is of 6 hidden layers which contain different numbers of neurons $[28, 23, 59, 19, 39, 51]$. Moreover, even if the number of epochs chosen by the *Hyperband* algorithm is 1000, we realize that the callback takes effect from epoch 600, and in the convergence plot, nothing indicates an undesirable fluctuation before this stopping point. Finally, the optimal ℓ_2-penalties have been chosen for each layer after optimization that provides non-zero regularization for 4 layers and the optimal learning rate has been set to $0,001$ with the *Adam* (Adaptive Moment Estimation) optimization algorithm that is a variant of stochastic gradient descent that uses both gradient and momentum information to update the model parameters. As we see that the curve of convergence is quite smooth and that we can consider the optimal convergence, using the optimal parameters, it is time to investigate the calibration of our neural network in the

different subgroups. In the same way of what we have done in the Calibration section, we will compute the *Hosmer-Lemeshow* test statistics and the corresponding p-values. We recall that if the p-value associated with the Hosmer-Lemeshow test statistic is less than a predetermined significance level (usually 0.05), then the model is considered to have poor fit and may need to be re-evaluated or adjusted.



(a) Calibration in training set $\mathcal{D}$

(b) Calibration in testing set $\mathcal{T}$

Figure 4.5: Calibration for LocalGLMnet

Thus as a $p-value$ closer to 1 indicates better calibration as non-significant $p-value$ suggests that there is no significant difference between the observed and expected frequencies, we are satisfied with the results we obtain, that are outperforming results from our previous models.

4.4.6 Variable selection

As we have discussed about the performances of the neural network that takes all features preprocessed as inputs, it's time to go deeper in the way the architecture allows for variable selection. For achieving this we will have to introduce a way for coefficient nullity testing as we have said that common parametrical statistical tests such as the *wald test* and *likelihood ratio test* one are not relevant for the implementation of the *localGLMnet*. In order to identify which fluctuation around 0 is acceptable for our regression attentions $\widehat{\beta}_j(x)$ we will choose to add two supplementary features x_{q+1}, x_{q+2} that are not relevant for our prediction model as we know that the true regression function $\mu(x)$ does not contain these features. In fact, adding two random features to our input features with different distributions is a technique known as *noise injection* as this can help in understanding the importance of individual features and their influence on the output of a machine learning model by analyzing fluctuation of these

features regression attention $\widehat{\beta}_{q+1}(x), \widehat{\beta}_{q+1}(x)$. We will choose the construct two identically and independently distributed features with mean $\mu(x_{q+1}) = \mu(x_{q+2}) = 0$ and unit variance. Thus we first try to build a model with centered and normalized continuous features and two control features, the first one following a uniform distribution and the second one following a standard normal distribution. We can compute the mean values of the regression attentions attribuated to thes features and their respective standard deviations such that:

$$\bar{b}_{q+1} = \frac{1}{n}\sum_{i=1}^{n}\widehat{\beta}_{q+1}(x_i^+) \qquad (4.25)$$

$$\widehat{s}_{q+1} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\widehat{\beta}_{q+1}(x_i^+) - \bar{b}_{q+1})^2} \qquad (4.26)$$

where (x^+) is the extended input matrix that contains 33 features, after *noise* introduction and we obtain $\bar{b}_{unif} = -0.07103$, $\bar{b}_{gauss} = -0.0083$, $\widehat{s}_{unif} = 0.0798$ and $\widehat{s}_{gauss} = 0.0115$. We can see that the choice of the distribution of our noise feature slightly change the distribution of regression attentions, we plot them in what follows:

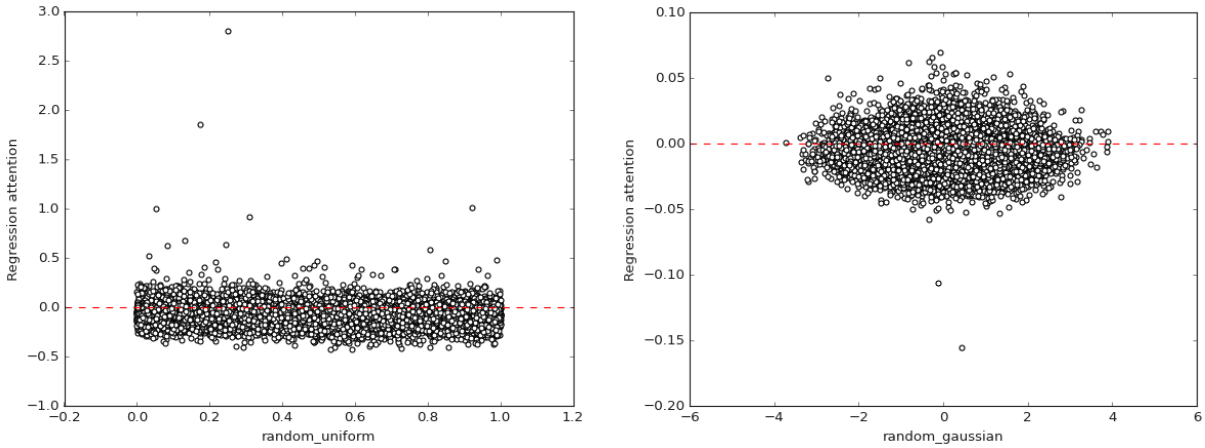
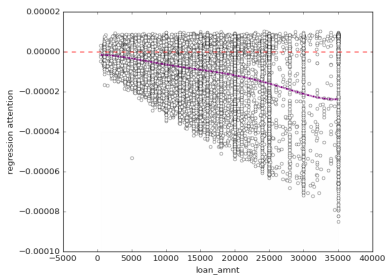
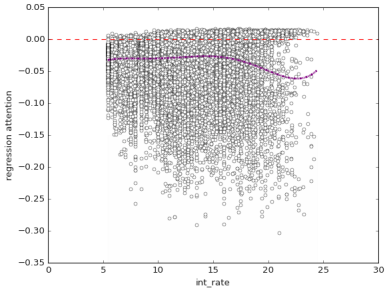


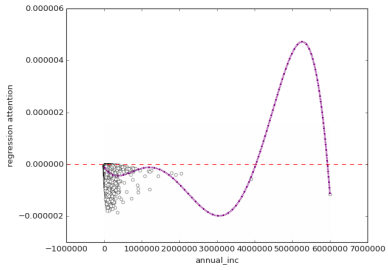
Figure 4.6: Beta attribution for noise introduced features

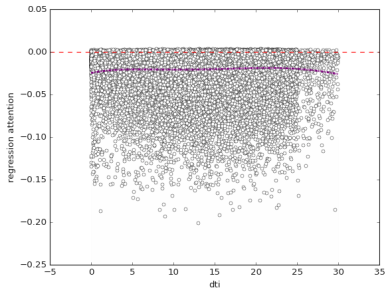
Unfortunately, the introduction of these noise features leads to poor performances and interpretations as it makes it harder for the network to learn useful patterns in the data. This is because the noise can introduce additional uncertainty into the input data, which can make it harder for the network to make accurate predictions and drown out the input features signal that the network needs to learn. Thus, as we consider features with a huge ranges that have been preprocessed we will prefer to perform deviance testing methods based on our *min-max* scaled features that leads to optimal results in term of calibration. We first display the values taken by our regression attentions for every values of our input features as we recall that in the *localGLMnet* we will try to detect the different feature behaviours cited in the Interpretability &

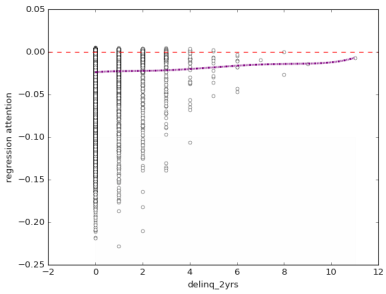
feature selection procedure table:

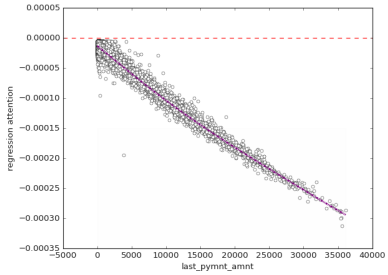


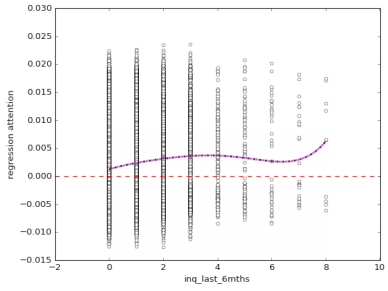


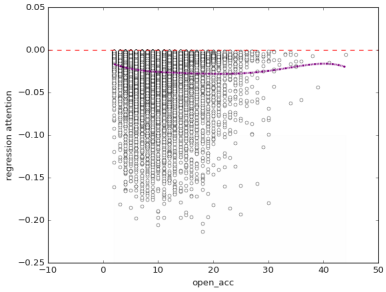


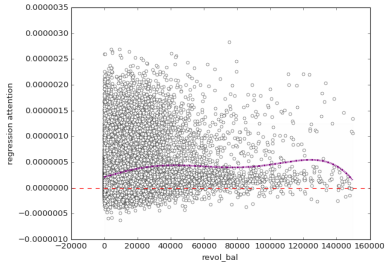


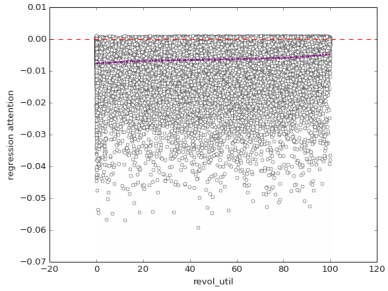


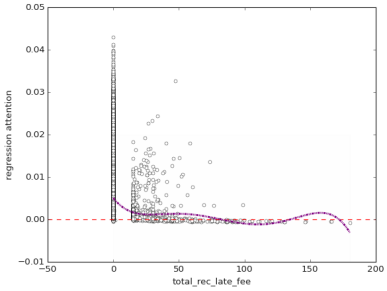


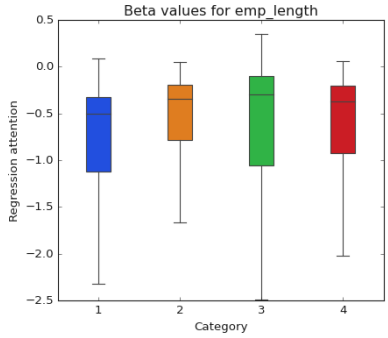












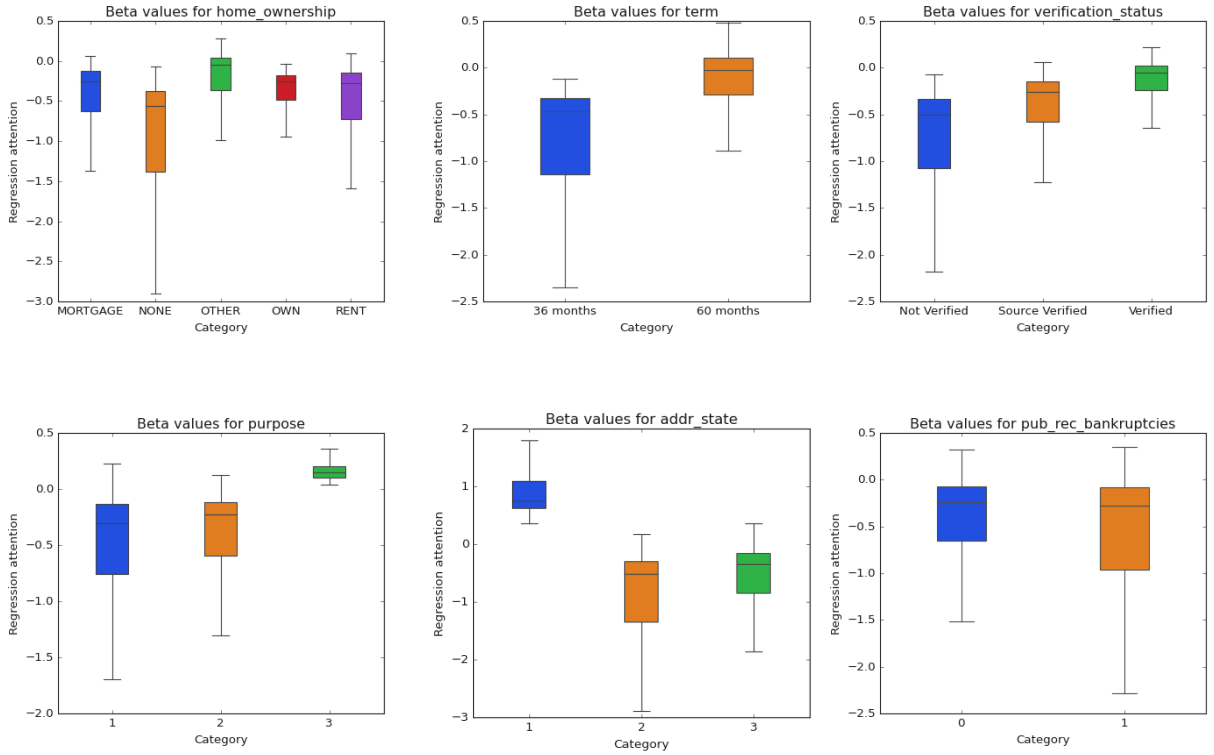


Figure 4.12: Regression attentions for saturated *localGLMnet*

What interpretation of these coefficients concerns, for continuous ones we do not see any regression attention concentrated around 0 (*red* line), meaning that all features are relevant to describe the true regression function μ. We clearly see the impact of outliers over the values taken by the regression attention in the *annual_inc* and points in the *last_pymnt_amnt* seem to be concentrated around a line of constant negative slope. For binary features we can already interpret the regression attentions $e^{\widehat{\beta}(x)}$ that shows us by how much the probability of defaulting contracts decreases ($e^{\widehat{\beta}(x)} < 1$) or increase ($e^{\widehat{\beta}(x)} > 1$) that is associated to the level 1 with respect to the level 0 of the binary feature. In fact, we retrieve the relative relationship between the odds ratios in the levels of *emp_length, term* and *purpose* but for others it's a bit tedious. What we have to say about continuous ones is that $e^{\widehat{\beta}(x)}$ tells us by how much the odds of being *charged off* will change for each 1 unit change in the predictor as it multiplies the odds of defaulting for each unit increase. Thus we can see that the highest the *last payment amount* and *loan amount* values, the highest the decrease in odds of defaulting for 1 unit increase.

We briefly introduced in the section Generalized Linear Models the way we can construct tests that are based on comparing the goodness-of-fit of nested models such as the *deviance test* that has a test statistic that is calculated as the difference between the deviance of the simpler model and the deviance of the more complex one, multiplied by -2. Moreover, this test statistic

follows a chi-squared distribution with degrees of freedom equal to the difference in the number of parameters between the two models such as 4.18. We recode this in a loop to test all coefficients combinations so that if we consider two nested models M_r, M_f so that $M_r \subset M_f$, with $\widehat{\mu}^r, \widehat{\mu}^f$ be respectively the maximum likelihood estimator under M_r, M_f and $D^* = D(y, \widehat{\mu})$ the scaled deviance statistic, then the likelihood ratio statistic is given by:

$$LRT = -2(D^*(y, \widehat{\mu}^r) - D^*(y, \widehat{\mu}^f)) \sim \chi^2_{df,1-\alpha} \tag{4.27}$$

where $df = f - r$ that are respectively number of parameters in models M_r, M_f that belong to the same EDM class, sharing the same ϕ. And thus if $\chi^2_{df,1-\alpha} < LRT$ we can reject the subset model M_r to be appropriate compared to the full one M_f.

But we will not stop our feature selection to this as the backward elimination algorithm is better than simply removing all variables at once and stopping as it allows for a more systematic and thorough evaluation of the effects of each variable on the model performances. That's why we will recode the *backward stepwise* selection procedure in order to find a submodel that may reduce the binomial deviance of the saturated one. Thus, our algorithm will be *deviance*-based as it will remove the feature that allows for the minimal submodel deviance among all features when removed. Then we will loop to find the best submodel from the previously retained one until the *deviance* does not decrease any more, we display the pseudo code of this one in the Pseudo Code for deviance based feature selection apprendix. We display the results for the first step of this algorithm and if you understood the pseudo code, we expect the model to first remove the *Scaled_revol_util* feature before entering the second iteration, going further by sub-setting models until no more improvement is found:

Feature removed	$D_{\mathcal{T}(y,\widehat{\mu}^r)}$	$\widehat{\mu}_{\mathcal{T}}$	LRT	$p_{0.05}$	
none (saturated)	4298.61	0.142	-	-	-
Scaled_loan_amnt	4501.89	0.162	-406.55	< 0.05	to include
Scaled_int_rate	4516.51	0.155	-435.80	< 0.05	to include
Scaled_annual_inc	4474.23	0.141	-351.23	< 0.05	to include
Scaled_dti	4380.79	0.135	-164.35	< 0.05	to include
Scaled_delinq_2yrs	4348.78	0.138	-100.32	< 0.05	to include
Scaled_last_pymnt_amnt	5780.54	0.149	-2963.84	< 0.05	to include
Scaled_inq_last_6mths	4342.19	0.157	-87.16	< 0.05	to include
Scaled_open_acc	4459.86	0.131	-322.495	< 0.05	to include
Scaled_revol_bal	4355.01	0.129	-112.80	< 0.05	to include
Scaled_revol_util	4333.17	0.141	-69.11	< 0.05	to include
Scaled_total_rec_late_fee	4517.97	0.140	-438.71	< 0.05	to include

Table 4.2: Features selection: Deviance test statistics for continuous

Feature removed	$D_{\mathcal{T}}(y, \widehat{\mu}^r)$	$\widehat{\mu}_{\mathcal{T}}$	LRT	$p_{0.05}$	
none (saturated)	4298.61	0.142	-	-	-
emp_length_1	4440.8	0.168	-284.37	< 0.05	to include
emp_length_2	4438.41	0.139	-279.59	< 0.05	to include
emp_length_3	4463.65	0.131	-330.08	< 0.05	to include
emp_length_4	4483.29	0.150	-369.36	< 0.05	to include
term_36 months	4438.43	0.131	-279.64	< 0.05	to include
term_60 months	4484.26	0.169	-371.29	< 0.05	to include
home_ownership_MORTGAGE	4479.82	0.156	-362.42	< 0.05	to include
home_ownership_NONE	4475.83	0.146	-354.42	< 0.05	to include
home_ownership_OTHER	4498.29	0.152	-399.3	< 0.05	to include
home_ownership_OWN	4493.24	0.139	-389.26	< 0.05	to include
home_ownership_RENT	4453.42	0.138	-309.61	< 0.05	to include
verification_status_Not Verified	4492.49	0.135	-387.75	< 0.05	to include
verification_status_Source Verified	4392.69	0.137	-188.16	< 0.05	to include
verification_status_Verified	4396.82	0.145	-196.41	< 0.05	to include
purpose_1	4427.68	0.149	-258.13	< 0.05	to include
purpose_2	4497.18	0.147	-397.14	< 0.05	to include
purpose_3	4438.73	0.143	-280.20	< 0.05	to include
addr_state_1	4455.89	0.152	-314.54	< 0.05	to include
addr_state_2	4464.93	0.156	-332.64	< 0.05	to include
addr_state_3	4427.48	0.136	-257.73	< 0.05	to include
pub_rec_bankruptcies_0	4437.47	0.146	-277.71	< 0.05	to include
pub_rec_bankruptcies_1	4401.09	0.153	-204.96	< 0.05	to include

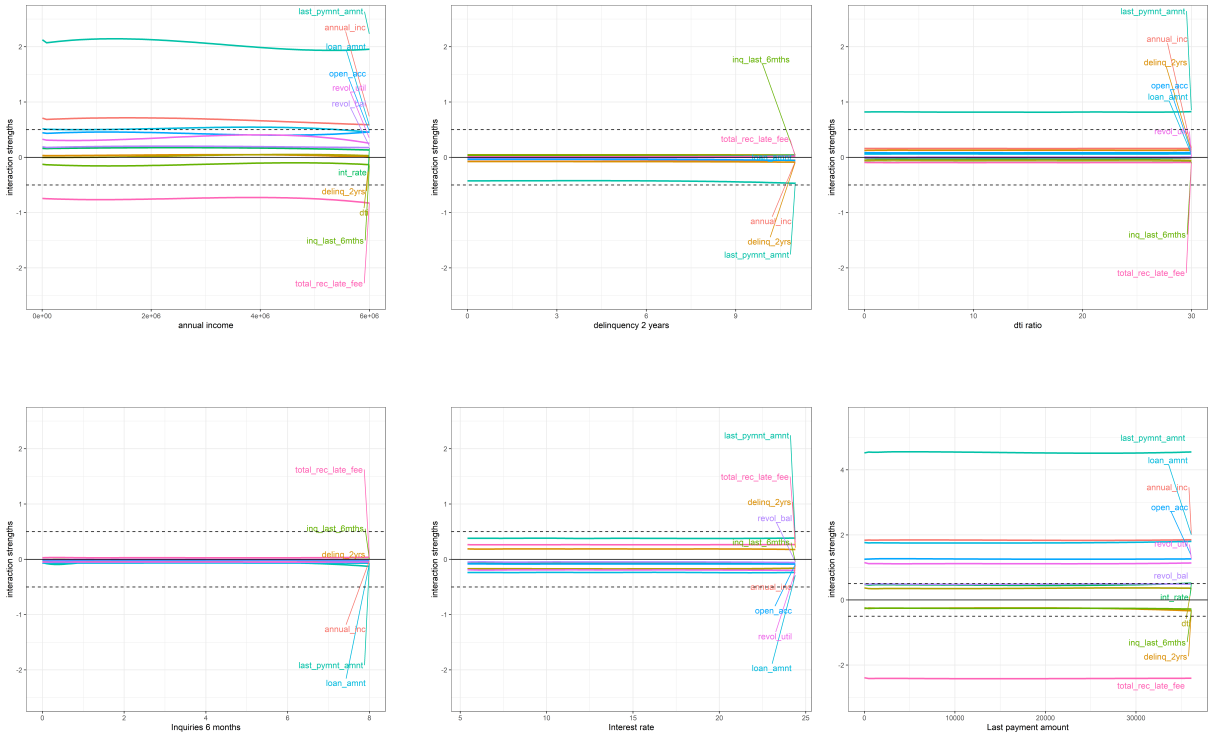
Table 4.3: Features selection: Deviance test statistics

And finally our algorithm finish after 2 iterations with a sub-model with $p - 2$ features compared to the saturated one that contains all features displayed above. It has chosen to remove *Scaled_revol_util* and *Scaled_open_acc* and retain all other features with a deviance for the test set $D_{\mathcal{T}}(y, \widehat{\mu}^r) = 4289.19$ which corresponds to a *likelihood ratio statistic* of 18.41 and a $p_{value} = 0.9998$. We recall that the null hypothesis H_0 that is tested corresponds to $\widehat{\beta}_{Scaled\_revol\_util}(x) = \widehat{\beta}_{Scaled\_open\_acc}(x) = 0$ and that the saturated model is not performing better that the reduced one when removing these 2 features for the retained *localGLMnet* architecture.

4.4.7 Interactions with spline fitting

The interaction analysis is one of the main part of this thesis as it justify the *local* part of the implementation of the *localGLMnet*. In fact as it's now clear that the powerful modern

architecture allows for feature selection, this section will allow to detect linear and non linear interactions between features. The first step will be to compute the gradients $\nabla\widehat{\beta}_j(x)$ for $1 < j < q$ that allows to study the derivative of regression attention with respect to x_k for fixed j as explained in the section Interpretability & feature selection procedure. Therefore we will construct a surrogate *keras* model as shown in the appendices in the section Python Code for Gradient extraction that start from the before-last layer *Attention* that contains the regression attentions $\widehat{\beta}(x)$. Actually, we recall that this layer has number of neurons being equal to the input layer and thus contains representation of inputs as a function of all other inputs as we construct a fully connected feed forward neural network. These representations have been learned during training and thus after convergence, we can get the output of this layer and compute the partial derivative for a regression attention $\widehat{\beta}_j(x)$ with resepct to all inputs. The strategy adopted for the implementation will be to loop over all neurons of this attention layer and to stack the partial derivative computed for all instance of the input layer. Morevoer, looking at the lateral dilatation's of the previously displayed regression attention scatter plots we expect interactions between our features and we recall that considering the gradient $\nabla\widehat{\beta}_j(x)$, then if there is no interactions with other features for feature j, then $\partial\widehat{\beta}_j(x)/\partial x_{j'} = 0$ for all $j' \neq j$ and thus, $\beta_j(x)x_j = \beta_j(x_j)x_j$ that will provide the exact functional form of $\beta_j(x)$. In fact, in that case for $\partial_{x_j}\widehat{\beta}_j$, if we get a regression line constant in 0, we can conclude that $\beta_j(x) = const$. At the opposite, if $\partial_{x_j}\widehat{\beta}_j \approx const$ and significantly different from 0, then the right functional form is not linear, and more precisely, indicates a quadratic term for feature x_j.



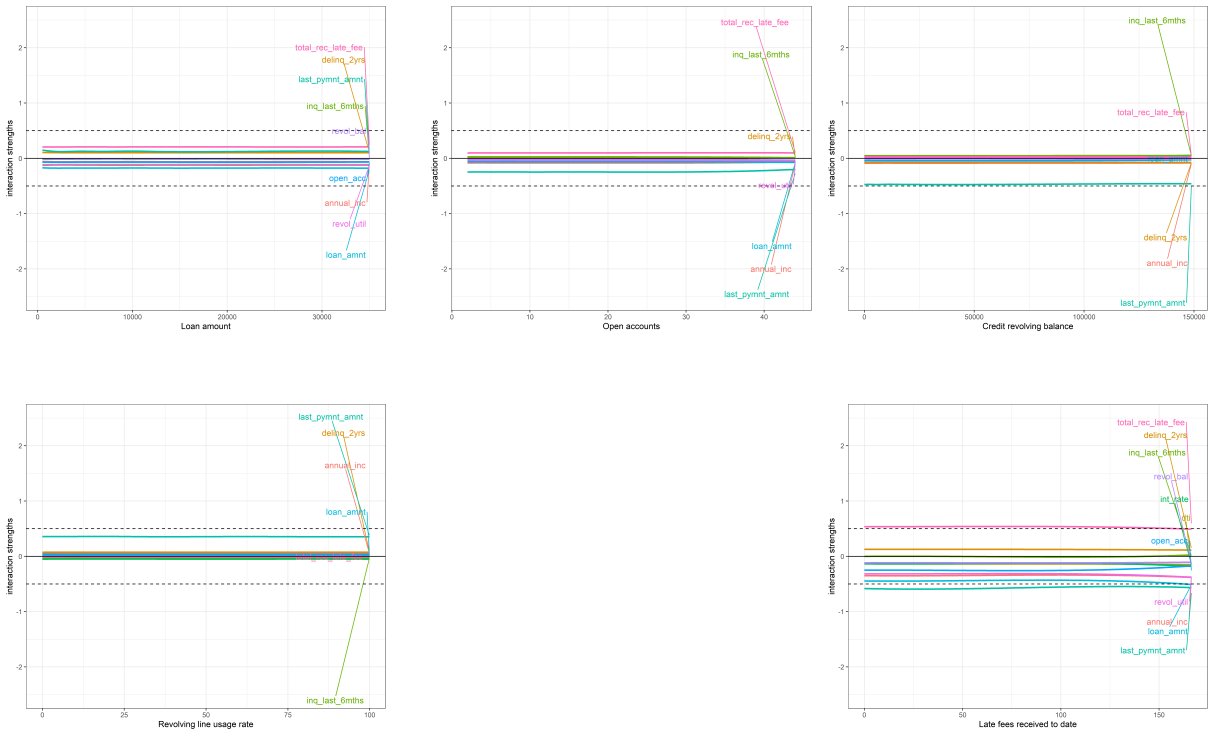


Figure 4.16: Spline fits to the gradients $\partial_{x_k}\widehat{\beta}_j(x_i)$ for continuous features, $i = 1, \ldots, n$

Thus we can try to reconstruct the right functional form of the true $\mu(x)$ regression equation. We point out that looking at the plots, we don't expect any non-linear interactions between features since all spline fitted lines seems to be constant with respect to values taken by x_j. What linear relationship concerns, we should expect the *inquiries 6 months* feature to present no interactions and to have a linear term since $\partial_{x_j}\widehat{\beta}_j(x) \approx 0$. Going further, *delinquency 2 years, dti ratio, open accounts, credit revolving balance, revolving line usage rate* show small linear interaction with the *last payment amount* feature since $\partial_{x_{j'}}\widehat{\beta}_j(x) = const \neq 0$ when $j^{'} = last\_pymnt\_amnt$ and it may explain the lateral dilation of such features in their regression attention scatter plots. This behaviour is logical for the *dti ratio* as this metric compares how much a loaner owes each month to how much he earns, as for the *revolving line usage rate* and *credit revolving balance* that are both linked with the payment amounts. For the *annual income* we notice some correlations with heavy strengths as with *last payment amount, open account, loan amount* with positive strengths and *total received late fee* with negative one. We also expect a quadratic term for *annual income*. The *last payment amount* seems to present linear interactions with *total received mate fee, loan amount, open account, revol util* with positive strengths and *total received late fee* with a negative one. The *interest rate* shows linear interactions with a large amount of features but no quadratic term, at the opposite of the *late fees received to date* that has the same behaviour but seems to present à quadratic term with a strength coefficient of 0.5.

Chapter 5

Extensions

5.1 Embedding Layer

An embedding layer is a fundamental component of a neural network that will be used for predicting the default probability for this thesis and can be seen as a mapping function that takes an input feature and maps it to a low-dimensional space, typically a vector space, where each dimension represents similarities in the input data. The purpose of the embedding layer is to learn a more compact and informative representation of the input data that can be used as input to the subsequent layers of the neural network.

Then for the embedding layer application to categorical variables, it will cluster labels that are more similar to each other in relation to the given output, the mapping function can be defined as follows:

$$\mathbf{e}_i = \mathbf{W}_i \cdot \mathbf{x}_i \tag{5.1}$$

where $\mathbf{x}_i$ is a one-hot encoding of the i-th categorical variable, $\mathbf{W}_i$ is a weight matrix, and $\mathbf{e}_i$ is the corresponding low-dimensional embedding vector. Thus, these layers have weights $\mathbf{W}_i$ that are learned during training of the model, enabling to understand similarities between these features.

For text or image features, the mapping function can be defined using pre-trained word embeddings or image embeddings that are typically trained on a large corpus of text or image data using algorithms such as *Word2Vec* or *GloVe* for text, while image embeddings are learned using convolutional neural networks on image datasets such as *ImageNet*. As it will be the main purpose of this thesis, to perform embedding mapping for text data using pre-trained word embeddings, the mapping function can be defined as follows:

$$\mathbf{e_i} = f(x_i) = E_M(x_i) \tag{5.2}$$

where $E_M(x_i)$ is the embedding representation of text x_i learned by the pre-trained model M and this mapping function can then be used to represent each sentence as a sequence of embeddings to a corresponding low-dimensional vector representation $\mathbf{e}_i$, which can be fed into our task-specific neural network model for default probability predictions. But it's important to point out that the best encoder for text embeddings representation depends on the specific task and the available data, and we will have to deal with the trade-off between task-specific embedding training and pre-trained ones. Then, when training a neural network for predicting the default probability, the embedding vectors for the different input features can be concatenated into a single input vector. The concatenated input vector can then be passed through one or more fully connected layers to generate the final output probability and the concatenation can be defined as follows:

$$\mathbf{v} = [\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n, \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m] \tag{5.3}$$

where $\mathbf{v}$ is the concatenated input vector, $\mathbf{e}_i$ are the embedding vectors for the text or categorical features, and $\mathbf{x}_i$ are the input features that do not require embedding (such as numerical ones). The concatenated input vector can then be passed through the *localGLMnet* architecture to generate the final output probability prediction.

5.1.1 Tokenization and word indexation

As we said before, there are several ways to perform the preprocessing of our data for the transformation of our text into embedding, as one strategy will be to use pretrained word embedding as *Word2Vec* that is a way to provide a starting point for the neural network's word embeddings, but creating our own word index may allow for customization, efficiency and consistency. Actually, by creating a custom word index, we then have complete control over the vocabulary used in our text data and we can customize the vocabulary to include specific words or exclude certain words based on the specific actuarial setting. For example, we can choose to only include the most frequent actuarial words in our vocabulary, which can reduce the size of our input data and thus improve the performance of the neural network. This will also be explored in what follows to compare performances.

As we have cleaned the *na* values in our previous preprocessing we can directly try to tokenize our text features as tokenization is the process of breaking down text into smaller units, such as words or subwords and is often one of the first steps in preprocessing text before it is fed into a neural network. In python the *tokenizer.texts_to_sequences* will be used to convert a sequence of text into a sequence of integers, where each integer corresponds to a specific word in the text and that takes a list of texts as input, and returns a list of lists, where each inner list contains the

integer sequence for a single text. More specifically, let S be a sequence of n words, represented as:

$$S = \{w_1, w_2, \ldots, w_n\} \tag{5.4}$$

where w_i is the i-th word in the sequence then the process of *tokenization* involves breaking down the sequence S into smaller units, called tokens. Then, let T be the set of all tokens generated from S, represented as:

$$T = \{t_1, t_2, \ldots, t_m\} \tag{5.5}$$

where t_i is the i-th token in the set and we can define tokenization as a function f that maps the sequence S to the set of tokens T, represented as:

$$f(S) = T \tag{5.6}$$

where the function f segments the input text into smaller units, such as words or subwords, based on certain criteria such as whitespace, punctuation, or character n-grams. You easily notice that the length of the embedding vector (m) and the vocabulary size of the dataset (n) are typically not the same. The choice of m depends on several factors, including the size of the dataset, the complexity of the task, and the computational resources available but in general, larger embedding sizes allow for more complex representations of words and may lead to improved performance on certain tasks, but may also require more computational resources and longer training times. Even if an increasing embedding length may increase the performances as it may be necessary to capture the nuances of the data, it's generally not recommended to use an embedding size larger than the square root of the vocabulary size, as this can lead to overfitting and poor generalization. But it's also important to consider the size of the hidden layers in the neural network when choosing the embedding size, as larger embeddings may require larger hidden layers to avoid underfitting.

Then, after tokenization, each word in a sentence is typically replaced by a numerical value based on its index in a vocabulary as let S be a sentence, w_i be the i^{th} word in the sentence, v be the vocabulary, and n be the number of words in the vocabulary, then, we can represent S as a sequence of integers $x_{1:n}$, where x_i is the index of the i^{th} word in the vocabulary:

$$S = \{w_1, w_2, \ldots, w_n\} \rightarrow x_{1:n} = \{\text{idx}(w_1), \text{idx}(w_2), \ldots \text{idx}(w_n)\} \tag{5.7}$$

Once we have represented the sentence as a sequence of integers, we can use the embedding matrix E to map each index to a corresponding embedding vector. During the processing, this embedding matrix E is typically initialized randomly and learned during training using

backpropagation as if we let e_i be the embedding vector for the i^{th} word in the vocabulary, then we can represent the sentence S as a matrix X of size $n \times d$, where d is the dimensionality of the embedding vectors:

$$S = \{w_1, w_2, \ldots, w_n\} \rightarrow X = \begin{bmatrix} e_{\text{idx}(w_1)} \\ e_{\text{idx}(w_2)} \\ \vdots \\ e_{\text{idx}(w_n)} \end{bmatrix} \tag{5.8}$$

5.1.2 Transfer Learning

As said *Transfer learning* with embedding is a powerful technique in machine learning that involves leveraging pre-trained models to solve a new problem where the pre-trained model has already learned meaningful representations of the input data and can be reused for a new predictive model. Thus, as the training of embedding with increasing number of features to be pre-processed may not be computationally efficient, the *transfer learning* is a powerful technique for text or categorical features that can help improve the accuracy of machine learning models while reducing the amount of labeled data required for training.

5.1.3 Processing and results

To compare performances and interpret our embeddings concerning the *text* features, we will compare what can be achieved with pre-trained embedding mappind such as *word2vec* and task specific training, with our own embedding learning for the default probability prediction. For this second processing, *word embeddings* that are dense word vectors to be trained will be preferred to the *one-hot encoding* which suggests replacing each sentence with a one-hot vector with length equal to the dictionary length, as it will allow to concentrate information in a low dimensional space. Thus, the embedding processing is an efficient way to concentrate information in a lower dimension space (often a few hundred dimensions) as it allows to represent words in *natural language processing* tasks by reducing the dimensionality of the word vectors while preserving their semantic relationships. When training, the embedding is learned from the text data during the training process of the *localGLMnet* architecture, precisely in the *embedding layer*, that maps each word to a point in the embedding space such that similar words are close together and dissimilar words are far apart. This means that words with similar meanings will have similar representations in the embedding space, and at the opposite words with opposite meanings will have dissimilar representations. Moreover, embeddings can be trained to handle out-of-vocabulary words by learning to generalize from similar words in the vocabulary, and thus avoid the model to break the *train, validation, test* lemma as the $\mathcal{T}$ and $\mathcal{V}$ may contain words that are not in the training set $\mathcal{D}$.

A last part to cover for this implementation is the way the output of an embedding layer has to be transformed to be concatenated to the *numeric* features before entering the *fully-feed forward* neural network. In fact, the 2D embedding output is typically transformed into a 1-dimensional vector using a *Flatten* $(\ldots_f)$ or *GlobalMaxPooling* $(\ldots_g)$ layer before being concatenated and can help to reduce the dimensionality of the input vector as an embedding layer can have a high dimensionality, especially if the vocabulary size is large or the embedding dimension is high. We describe the behaviour of these layers:

- *Flatten layer*: flattens the 2D tensor output of the embedding layer into a 1D tensor by stacking all the values of each feature map on top of each other. This means that the flattened output has the same number of elements as the product of the input dimensions.
- *GlobalMaxPooling layer*: performs a pooling operation over the entire sequence of word embeddings and returns the maximum value for each feature dimension. This means that the output has the same number of elements as the embedding dimensions.

We will compare the performances for both layer transformations but let's point out that *GlobalMaxPooling* layer can be especially effective for *text* features as it captures the most important features of the input sequence and discards the rest and thus can reduce the impact of noisy or irrelevant words in the sequence. We first decide to use the optimal architecture found previously and to add an embedding layer for each new *text* input, but once again the architecture will be tuned to fit the task and data properly in what follows. The results for the optimal architecture found in the section Architecture implementation and results is displayed in the following table:

Model	$D_{\mathcal{T}}$	$\widehat{\mu}_{\mathcal{T}}$	$\widehat{C}$	$p_{0.05}$	text features
optimal without embeddings	4298.61	0.142	56.85	0.959	-
optimal pre-trained embeddings$_f$	4320.72	0.149	58.21	0.945	emp_title
optimal pre-trained embeddings$_g$	4314.24	0.158	66.91	0.787	emp_title
optimal pre-trained embeddings$_f$	4351.66	0.168	87.26	0.198	emp_title, title
optimal pre-trained embeddings$_g$	4298.27	0.143	57.23	0.955	emp_title, title
optimal pre-trained embeddings$_f$	4563.86	0.128	84.03	0.272	emp_title, title, desc
optimal pre-trained embeddings$_g$	4317.27	0.152	61.42	0.902	emp_title, title, desc
optimal trained embeddings$_f$	7451.73	0.103	301.92	< 0.001	emp_title
optimal trained embeddings$_g$	7225.66	0.117	175.58	< 0.001	emp_title
optimal trained embeddings$_f$	7629.01	0.091	450.16	< 0.001	emp_title, title
optimal trained embeddings$_g$	7332.02	0.099	371.05	< 0.001	emp_title, title
optimal trained embeddings$_f$	7808.65	0.093	415.19	< 0.001	emp_title, title, desc
optimal trained embeddings$_g$	6882.97	0.105	308.07	< 0.001	emp_title, title, desc

Table 5.1: Advanced localGLMnet with embedding layer: performance results

Thus for the pre-trained embeddings, words in sequences have been linked with their respective *Word2Vec* embedding representations with *min_count = 1* that allows to map words that appears only once. As the default size of embedding is 100, we constructed the embedding matrix for each *text* features that have a shape $(len(v), 100)$, with each row of the embedding matrix that represents the embedding vector for a specific token in the vocabulary. In fact, we recall that when passing a sequence of tokens through an embedding layer in the embedding layer, the layer will use the embedding matrix to look up the embedding vector for each token in the sequence and the resulting output will be a sequence of embedding vectors with shape $(batch\_size, sequence\_length, embedding\_dim)$. With the *GlobalMaxPooling layer* it will be reduced to $(batch\_size, embedding\_dim)$ and the *Flatten layer* will produce a tensor of shape $(batch\_size, sequence\_length * embedding\_dim)$ before being concatenated with the numeric features. You will easily notice that the $sequence\_length$ has to be fixed, so this will be done using *padding* that allows to only retain the first i words of a tokenized sequence with $1 \leq i \leq n$, n being the size of the longest sequence in the *text* feature. For the pre-trained embedding representation we will have to add two supplementary parameters that are $weights = embedding\_matrix, trainable = False$, that indicates that the constructed embedding matrix has to be taken for the weight attributions of tokenized sequences and not trained.

Based on the above table 5.1 we can conclude that the *GlobalMaxPooling layer* seems to outperform the performances of the *Flatten layer* as expected when maintaining the optimal architecture found previously. Moreover, we clearly notice that the previously found architecture is not optimal anymore when passing new embedding inputs to the *localGLMnet* and that suggests a new *Hyperband* tuned architecture to train our embedding layers properly. However we found that the combination of *emp_title, title* features with pre-trained word embedding representations slightly improves the performances of the *saturated localGLMnet* with a mean frequency across the testing set $\widehat{\mu}_{\mathcal{T}} = 0.1432$ that is really close to the previously computed true frequency $\mu_{\mathcal{T}} = 0.1436$ with a comparable calibration across sub-groups of contract profiles.

As it's time to asses for the performances of our trained embedding representations for *text* features with the *Hyperband* tuned architecture we will introduce the procedure to assess for the learning of word representation in the *embedding layers*. In fact, we expect that similar words will have similar embedding representations. Thus, we will choose to perform *K-means* in embedding space to cluster the embedding representations of text features and grouping together embedding vectors that are similar to each other while separating dissimilar words. Inside these clusters we will measure *cosine similarity* between each word embeddings and their respective cluster center mean, that is a measure of similarity between two non-zero vectors so that the

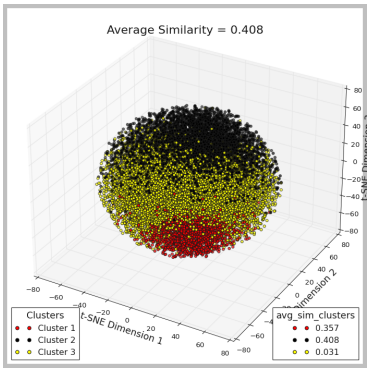
cosine similarity between two vectors u and v is defined as:

$$\text{similarity} = \cos(\theta) = \frac{u \cdot v}{|u|\,|v|} = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2}\sqrt{\sum_{i=1}^{n} v_i^2}} \tag{5.9}$$

where u and v are two vectors with n dimensions and the numerator of the equation calculating the dot product of the two vectors and the denominator the magnitude of the vectors. Then, the *cosine similarity* value ranges between -1 and 1, with 1 indicating that the two vectors are identical, 0 indicating that the vectors are orthogonal (have no correlation), and -1 indicating that the two vectors are opposite in direction. Therefore, computing the similarities within clustered embedding representations can be a good way to assess whether the embeddings have learned meaningful relationships between words as if the embeddings are well-trained, then similar words should be clustered together, and the cosine similarity within each cluster should be high. Conversely, if the embeddings are poorly trained, then similar words may not be clustered together, and the cosine similarity within each cluster may be low. Let's check the results for each model with respectively $1, 2, 3$ *text* features, keeping an eye on the calibration and similarities:

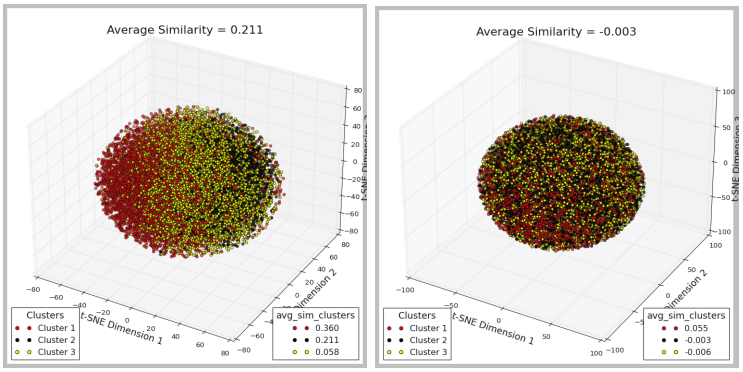
Model	$D_{\mathcal{T}}$	$\widehat{\mu}_{\mathcal{T}}$	$\widehat{C}$	$p_{0.05}$	text features
Hyperband trained embeddings$_g$	5877.16	0.139	87.73	0.1891	emp_title
Hyperband trained embeddings$_g$	5826.55	0.127	105.34	0.018	emp_title, title
Hyperband trained embeddings$_g$	5628.79	0.133	141.55	< 0.001	emp_title, title, desc

Table 5.2: Advanced localGLMnet with embedding layer: performance results (2)



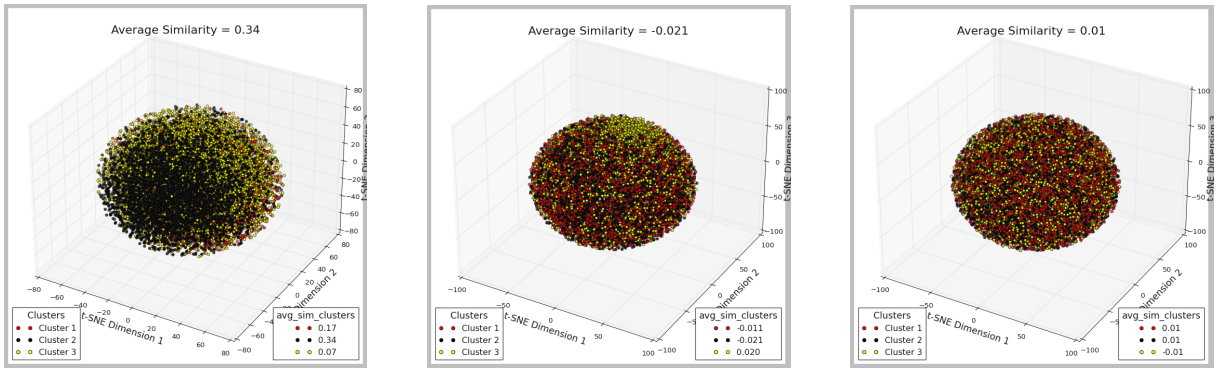
(a) emp_title feature

Figure 5.1: TSNE representation of learned embeddings with *Kmeans* (1)



(a) emp_title feature (b) title feature

Figure 5.2: TSNE representation of learned embeddings with *Kmeans* (2)



(a) emp_title feature (b) title feature (c) desc feature

Figure 5.3: TSNE representation of learned embeddings with *Kmeans* (3)

We choose to implement the well known *t-Distributed Stochastic Neighbor Embedding* to visualize trained text embedding representations (see figures 5.1,5.2, 5.3), that is a popular non-linear dimensionality reduction that is particularly useful when dealing with high-dimensional data, as it can reduce the number of dimensions while preserving the local structure of the data. Actually, maintaining the pairwise distances between the data points is achieved by modeling the high-dimensional similarities between the data points with a *Gaussian* probability distribution and the low-dimensional similarities with a *Student*'s t-distribution. If you want more insights about this, you can check the Pseudo Code for TSNE representation of embeddings section in the appendices. In summary, it's achieved by first computing the pairwise *euclidean distance* d_{ij} for each pair of high-dimensional data points x_i and x_j, computing a similarity matrix $\mathbf{P}$ that measures the pairwise similarities between the high-dimensional data points, defining a similarity matrix $\mathbf{Q}$ that measures the pairwise similarities between the low-dimensional data points and by minimizing the *Kullback-Leibler* divergence between $\mathbf{P}$ and $\mathbf{Q}$ with respect to

the low-dimensional data points using gradient descent. We refer to the paper of Cai and Ma (2022) that gives more insights over the theoretical foundations of this algorithm based on the gradient descent. Moreover, for our similarity computation, it can be improved by the *Similarity Preserving Representation Learning* of Lee, Park, and Shin (2022) that introduces the similarity-preserving embedding mapping *SET2BOX* and leverages the geometric properties of boxes to achieve accurate preservation of similarities.

As it's time to compare the results, in the table 5.2 we notice that we improve the results with hyperparameter tuning using the *Hyperband* algorithm in term of *binomial deviance loss, predicted default probability mean* $\widehat{\mu}_{\mathcal{T}}$ and *calibration* compared to what has been obtained with the optimal architecture found previously for trained embeddings. More precisely, we reach an acceptable calibration with a $p_{value} > 0.05$ for the architecture with a pipeline for the *emp_title* embedding mapping. Moreover, we can link the decreasing calibration performances with the increasing number of embedding pipelines to train before being concatenated with numeric input features, looking at the *within clusters average similarities*. In fact, even if the *title* and *desc* features does not reach a satisfying level of mapped similarities, we notice that the integration of these ones makes the similarities in cluster for *emp_title* decrease. As the number of clusters has been computed using the *wcss elbow* curve and has been each time found optimal when equal to 3, we go from $[0.357, 0.408, 0.031]$ with embedding training for *emp_title* lonely to $[0.17, 0.34, 0.07]$ with the 3 *text* features for the *within clusters average similarities* of the *emp_title* feature.

However, it is expected for task-specific trained embeddings to not always reach the same level of accuracy as pre-trained ones, even with hyper-parameter tuning and optimal neural network architectures as the training of *Word2vec* has been done on very large corpora, while our task-specific embeddings are trained on our smaller dataset. That being said, task-specific embeddings have the advantage of being tailored to the specific task at hand and the *keras* API allows us to also train pre-trained embeddings as fine-tuning pre-trained embeddings can be a good way to leverage the strengths of these ones while also tailoring the embeddings to the specific task at hand. Moreover, fine-tuning pre-trained embeddings can often lead to better performances compared to training embeddings from scratch, as the pre-trained embeddings have already learned useful information about the language that can be applied to the specific task. However, fine-tuning can still be sensitive to hyper-parameters and the specific details of the task, so it is important to once again consider the *Hyperband* hyperparameter tuner. To achieve this, we will initialize the embedding layer of your neural network with the pre-trained embeddings as it has been done previously with $weights = embedding\_matrix$, and then train the entire model on our specific default probability prediction task with $trainable = True$ in the *keras* embedding layer. Thus, during training, the pre-trained embeddings would be updated

along with the other weights in the *localGLMnet* neural network. Actually, our approach is justified by the experiments carried out in the paper of Xu and Yang (2023) that discusses the challenges associated with using pre-trained embeddings in e-commerce machine learning systems and highlights the conclusions drawn from a study supported by theoretical and experimental evidence. The significance of the work extends beyond e-commerce, providing insights for building robust machine learning systems that utilize pretrained embeddings. However, we display the results obtained with *hyperband* optimal architecture for our trained pre-trained embeddings:

Model	$D_{\mathcal{T}}$	$\widehat{\mu}_{\mathcal{T}}$	$\widehat{C}$	$p_{0.05}$	text features
Hyperb. trained pre-t embeddings$_g$	5749.49	0.136	89.28	0.159	emp_title
Hyperb. trained pre-t embeddings$_g$	5663.06	0.141	91.35	0.126	emp_title, title
Hyperb. trained pre-t embeddings$_g$	5127.70	0.146	63.02	0.875	emp_title, title, desc

Table 5.3: Advanced localGLMnet with embedding layer: performance results (3)

What is really interesting in the results we obtained with fine tuning our pre-trained embeddings is that apparently they outperform embedding representations learned from scratch in term of *deviance loss* and *calibration* but still lead to a lower prediction accuracy than *Word2Vec* embedding representations. The last cited behaviour is not surprising at all as fine-tuning a pre-trained model requires a substantial amount of task-specific data to effectively adjust the embeddings and if the dataset is relatively small, it may not provide enough information to update the embeddings in a way that improves performances. For instance, the *Google News Word2Vec* model used to train the embedding representation was trained on a massive dataset containing about 100 billion words from news articles and if the 35000 occurrences of text features in our dataset contains biased language or reflects societal biases, fine-tuned *Word2Vec* representations can capture and amplify those biases in the learned embeddings. Moreover, if the training data is imbalanced in terms of representation or coverage of certain groups or topics, it can also result in biased embeddings that might assign more importance to the dominant or prevalent patterns, leading to underrepresentation or skewed representation of certain groups or concepts. It's important to be aware of these potential biases when using pre-trained *Word2Vec* embeddings or training your own models as evaluating and mitigating bias in word embeddings is an active area of research, and various techniques have been proposed to address this issue, such as *Adversarial Debiasing* explained by Kenna (2021) and that can be identified by some measures such as the *associated mean average cosine similarity* introduced by Manzini, Lim, Tsvetkov, and Black (2019) that follows the spirit of what we have done previously.

5.2 Shapley Values

As the new architecture of the *LocalGlMnet* can be also interpreted in terms of *Shapley Values* we will introduce this concept and extend our analysis. As *Shapley values* is a method used to attribute the importance of each feature in a prediction made by a machine learning model, it is based on the concept of cooperative game theory, where each feature is considered as a player in a game, and the prediction is the reward that they receive after playing the game. Thus, the *Shapley value* of a player is the average contribution that they make to the reward across all possible coalitions of players. Mathematically, the *Shapley value* of player j is defined as:

$$\phi_j(v) = \frac{1}{N!} \sum_{S \subseteq N \setminus \{j\}} \sum_{\pi \in \Pi(S)} (v_{\pi(S) \cup \{j\}} - v_{\pi(S)}) \tag{5.10}$$

where v is the total reward, N is the set of all players (features), S is a subset of players (excluding j), π is a permutation of S, and $|\pi(S)|$ is the number of players before j in the order π. The intuition behind this formula is that the *Shapley value* of player j is the sum of the marginal contributions that they make to the reward across all possible orders of players and the denominator $N!$ normalizes the contributions to ensure that the total reward is distributed among all players.

To apply the *Shapley values* theory to a machine learning model, we consider the output of the model $f(x)$ as the reward v, where x is the input feature vector. We define a reference point x_0, which is usually the mean or median of the training data, and compute the difference between the output of the model at x and x_0 for each subset of features and then we find the *Shapley value* of each feature that is the average of these differences across all possible subsets of features. Mathematically, the Shapley value of feature j for input x is:

$$\phi_j(v) = \frac{1}{M} \sum_{S \subseteq N \setminus \{j\}} \sum_{\pi \in \Pi(S)} (f_{\pi(S) \cup \{j\}}(x) - f_{\pi(S)}(x)) \tag{5.11}$$

where M is the total number of possible subsets of features and thus we can use these *Shapley values* to explain the prediction of a machine learning model by attributing the contribution of each feature to the final output; this can help to identify which features are most important for the prediction, and to detect any biases or inconsistencies in the model. Finally, the results can be used to obtain an additive decomposition of the deep learning model's prediction, where the contribution of each feature is added together to obtain the final prediction. The additive decomposition is obtained such that:

$$f(x) = \phi_0 + \sum_{j=1}^{p} \phi_j x_j \tag{5.12}$$

where $f(x)$ is the output value of our model for the set of inputs x, and ϕ_0 is the model's baseline prediction that is the average prediction across the training set. In the case of the *LocalGLMnet* we want to estimate the mean such that $f(x) = \widetilde{\mu}(x)$, the joint response.

As it's not computationally efficient when the number of input features increases, the approximation of the *Shapley value* is based on the assumption that the features are independent and we can approximate the previous expression for player i as follows:

$$\phi_j(v) \approx \frac{1}{M} \sum_{S \subseteq N \setminus j} (|S|!(N - |S| - 1)!) \frac{v_{S \cup j} + v_S - v_N}{2} \tag{5.13}$$

where v_N is the model prediction when all features are present, and v_S and $v_{S \cup j}$ are the model predictions when the subset S of features and the subset $S \cup j$ are present, respectively. But as this approximation is based on the assumption that the contribution of each feature is independent of the other features that allows to estimate the contribution of each feature by comparing the model predictions when the feature is included and excluded from the model, it may not hold true when input features are highly correlated. The factor $|S|!(N - |S| - 1)!$ in the approximation takes into account the number of possible orderings of the features in S, and the $v_{S \cup j} + v_S - v_N$ term is the difference in the model predictions when feature i is added to the subset S of features.

That's why the *localGLMnet* seems to be more suitable as it directly assumes an additive decomposition of the output mean $\mu(x)$ after the link function $g(.)$ has been applied, as we have described the way it's computed in the architecture explanation. This means that the interpretation of the model's predictions becomes an integral part of the model assumptions for the *LocalGLMnet* as the regression attentions $\beta_j(x)$ is comparable to the *Shapley value* $\phi_j(v)$ in term of interpretations and we point out that the architecture is able to directly obtain the importance of each feature without having to fit a separate model.

5.2.1 Surrogate Shapley values for localGLMnet architecture

This final part will help in assessing the validity of the *localGLMnet* model constructed previously by providing insights into how the neural network architecture is making predictions as it can help in identifying which features of the input data are most important for the model's predictions, and how these features interact with each other to produce the final output. Actually, in the presence of correlated features, *Shapley values* can still provide insights into the collective importance of

feature groups or interactions as they can identify groups of features that have strong joint effects on the model's predictions. However as said previously, isolating the specific contribution of each correlated feature individually may be difficult due to their inter-dependencies. Additionally, $SHAP$ can also help in identifying possible biases in the model's predictions by visualizing the *Shapley values* for different subgroups of the data we are working on. We first choose a background dataset that will be a sub matrix with the same number of columns p and 1000 rows randomly selected and that will be used to compute the reference value from which the deviation will be computed so that instance level explanations are found with:

$$p_i - p_0 = \sum_{j=1}^{p} \phi_{i,j,0} \tag{5.14}$$

where $\phi_{i,j,0}$ is the contribution of each feature $x_{i,j}$ to the prediction deviation from the background defined default probability p_0 with respect to the predicted value p_i for contract i. The background has been chosen properly as it has been advanced that too small background samples may lead to fluctuation of Shapley explanations, as advanced in the empirical study of Yuan, Liu, Kang, Miao, and Wu (2023). Furthermore, we can average feature importance across all contracts by computing this:

$$V_{j,0} = \sum_{i=1}^{n} |\phi_{i,j,0}| \tag{5.15}$$

We can summarize the analysis of *Shapley values* by computing the impact of each feature on the model output for all instances in the test set $\mathcal{T}$ to assess for decision made by the *localGLMnet* for high and low values of our explanatory features. Thus, the following plots are going to navigate through *local* and *global* explanations of the feature contribution understanding of our neural network. We will end by providing a clear visual explanation of the contribution of each feature to the model's prediction for a particular contract, by stacking all instances of the test set by similarities that allows to understand patterns and trends in the feature importance across different contracts. We will display and try to interpret these plots below but let's point out that our base value p_0 has been computed and is of 0.32 and even if it's far away from the true mean value $\mu_{\mathcal{D}} = 0.1464$ the interpretation of the results should be based on the *Shapley values* themselves, and not solely on p_0. In fact, even if p_0 is not close to the true mean output value, the *Shapley values* can still provide useful insights into how the model is making predictions and which features are most important in determining the output default probabilities.

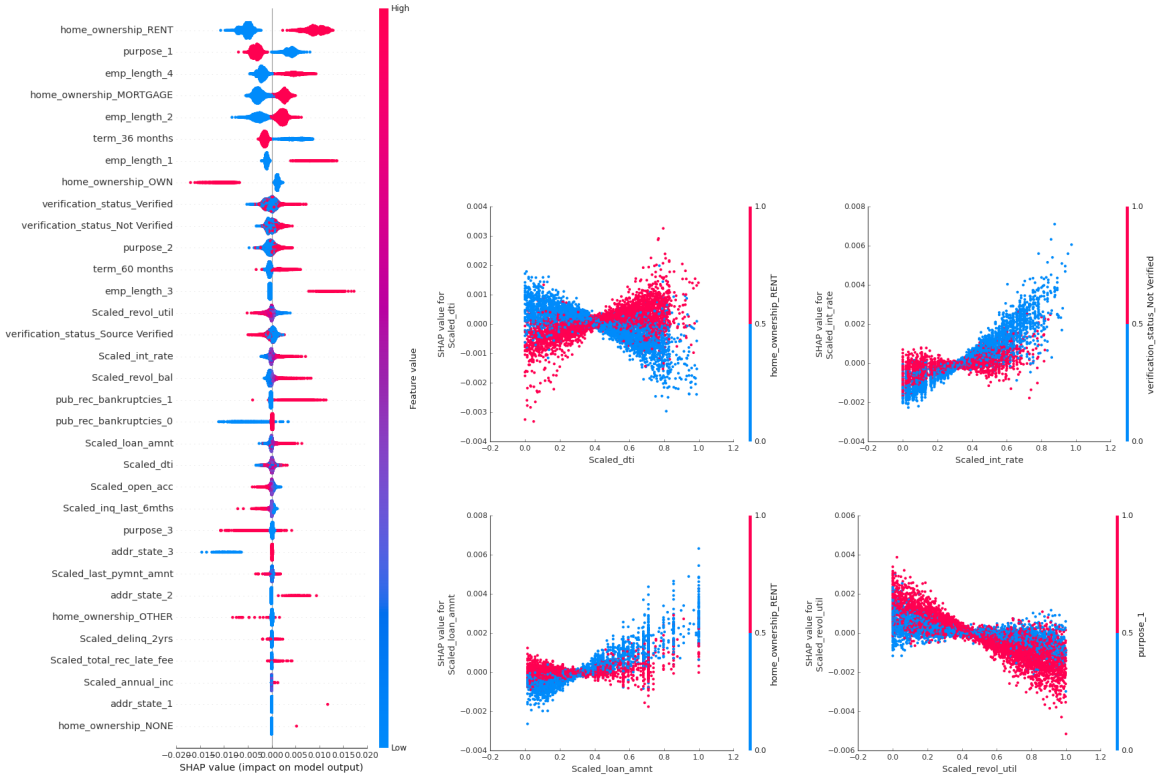


Figure 5.4: Global impact of features on model output & dependence plots

The plot on the left side summarizes the *Shapley values*, thus the impact computed on the model output for all features in the dataset. The points are colored following the values taken by the features and we can already notice that for our binary features *home_ownership_RENT, emp_length, home_ownership_MORTGAGE, term_60_months, pub_rec_bankruptcies_1*, being part of these groups increases the default probabilities, what has been concluded in the Odds of default analysis part of this thesis. In addition, owning your own home means that the loan contract is less likely to default and the *localGLMnet* seems to understand that a highest interest rate increases the default probability. Finally, the plots on the right are displayed to understand dependencies and show how the predicted output value changes as the input feature value increases, while also accounting for the effect of other features in the model. Here the vertical axis still shows the SHAP value for the input feature, which represents how much the feature contributes to the model's output value for a given data point and thus we can already say that not being a renter changes the impact of the loan amount and of the *dti* ratio on the default probability. In fact, it may make sense that when the *dti* ratio increases and the person who made the contract is a tenant, the probability of default increases. Finally, we can see that the *purpose_1* clustered label of the purpose feature, that contains levels such as *vacation, educational,*

medical, renewable energy, induces a negative slope to the impact of the *revol_util* feature on the output default probability. Actually, individuals who responsibly manage their credit and utilize a higher percentage of their available credit for specific purposes may demonstrate more stable financial behavior. It could indicate better planning, discipline, or a higher ability to repay debts and indicates that individuals who have higher revolving utilization while using credit for vacation, education, medical, or renewable energy purposes may exhibit lower default risk compared to those with lower utilization. For example, educational investments often lead to acquiring skills and knowledge that enhance job prospects and increase income potential as higher income levels generally improve individuals' ability to repay their debts, reducing the likelihood of default. In the same way, individuals who invest in their education, health, vacation, or renewable energy projects often exhibit a forward-thinking mindset as they prioritize long-term goals and are more likely to make responsible financial decisions and this mindset can translate into better financial management, including timely repayment of debts.

Chapter 6

Conclusion and discussion

In conclusion, this master's thesis has focused on the development and evaluation of the novel architecture of the *localGLMnet*, aiming to improve predictive performance while maintaining interpretability. By comparing the results of this architecture with statistical basic models, such as logistic regression and machine learning neural networks, we have gained valuable insights into its effectiveness and potential advantages. Throughout the thesis, we have meticulously designed and implemented the specific architecture, taking into consideration its unique features and capabilities. Especially, our approach combines the strengths of traditional statistical such as *glm* models with the flexibility and complexity of *fully-feed forward* neural networks, aiming to strike a balance between predictive power and interpretability by training *regression attentions*. These results highlight the importance of considering the interpretability aspect, particularly in domains where transparency and explainability are crucial, which allows the application of the *localGLMnet* structure to be applied to a wide range of usages in domains like *healthcare outcome prediction, customer churn analysis, fraud detection* or *credit risk assessment* in a comparable way of what has been done for this thesis.

6.1 Conclusion

The results of our comparative analysis revealed notable advancements achieved by the proposed architecture as first of all, it consistently outperformed logistic regression in terms of both predictive accuracy and calibration and demonstrated competitive performance compared to machine learning neural networks. We will summarize the results obtained with the implemented architectures as we recall that the metrics of interest are the *Deviance Loss* as it measures the discrepancy between the predicted probabilities and the observed outcomes and minimizing this metric is desirable as it indicates better overall predictive performance, the *predicted Mean Frequency* as a close match between the predicted and true mean frequencies in $\mathcal{T}$ indicates that

the model is accurately capturing the underlying default patterns and finally the *Calibration in subgroups* as it is important to assess calibration in different subgroups, especially if there are variations in default patterns across different segments in such a way that a well-calibrated model will provide reliable and accurate predictions within each subgroup.

Model	$D_{\mathcal{T}}$	$\widehat{\mu}_{\mathcal{T}}$	$\widehat{C}$	$p_{0.05}$	text features
1 layer NN (K = 25, λ = 1 e-09)	5378.152	0.141	90.28	0.114	without text
LocalGLMnet (saturated)	4298.61	0.142	56.85	0.959	without text
LocalGLMnet $-(revol\_util, open\_acc)$	4289.19	0.143	55.72	0.962	without text
LocalGLMnet tr. emb_g (scratch)	5877.16	0.139	87.73	0.189	emp_title
LocalGLMnet emb_g (W_2V)	4298.27	0.143	57.23	0.955	emp_title, title
LocalGLMnet tr. emb_g (W_2V)	5127.70	0.146	63.02	0.875	emp_title, title, desc

Table 6.1: Comparative of optimal *LocalGLmnet performances*

We point out that we retained the models based on the closer *predicted Mean Frequency* in $\mathcal{T}$ compared to the true one that is of $14,36\%$. Actually, the results obtained give us a wide range of different *LocalGLMnet* implementations depending on the features available as we recall that in actuarial setting, some risk factors may not be allowed for our portfolio pricing. So, if the main objective of an insurance company is to estimate loss provisions or reserves then the *predicted mean frequency* is essential to assess the potential losses due to default events and thus we can retain *LocalGLMnet* with $p-2$ features and no text features or the *saturated LocalGLMnet* with both the occupation listed by the borrower on the loan application and the borrower-provided loan title features that lead to $\widehat{\mu}_{\mathcal{T}} = 14.3\%$. On the other hand, when evaluating model fairness and non-discrimination behaviour only, *Calibration in subgroups* is of main concern to ensure that the model's predictions are not biased or discriminatory across different demographic or characteristic subgroups as evaluating calibration helps identify potential disparities and ensures fair treatment in decision-making processes. In that case, we point out that we can also retain the *saturated LocalGLMnet* without text features and the *LocalGLMnet* with the 3 text features as the results in term of *Hosmer-Lemeshow* calibration test are still acceptable.

Importantly, while achieving improved predictive performance, our architecture maintained a high level of interpretability, allowing for a clear understanding of the underlying factors driving the predictions and their intrinsic interactions as the *keras API* tools were used appropriately for surrogate models construction, gradients of $\widehat{\beta}(x)$ derivation and embedding layers integration. In addition, the surrogate *Shapley values* computation has allowed to understand dependences between categorical features and continuous ones as the paper of Richman and Wüthrich (2021) only focus on the interpretation of volatility of *regression attentions* box-plots.

6.2 Discussion about improvements

Moving forward, further research and development can be pursued to enhance the proposed architecture. This may include exploring additional techniques to optimize its performance, investigating the scalability of the model to larger datasets, and examining its applicability to specific real-world problems. Actually, for a wide range of input features it may be suitable to allow the architecture to use the *unit-wise adaptative dropout*. In fact, instead of adjusting the dropout rate at the layer level, *unit-wise adaptive dropout* adjusts the dropout rate for individual units within a layer, and the paper of Keshari, Singh, and Vatsa (2018) demonstrates how to take advantage of this *guided dropout*. We can imagine an architecture that implements this regularization for the before last layer containing our *regression attentions* as units that contribute less to the network's overall performance or have lower activation values may have a higher dropout rate, and based on a well chosen *validation loss* it may allow for the integration of a kind of *wrapper* feature selection procedure. This may be an alternative to the tedious procedure of noise injection described in the reference paper of Richman and Wüthrich (2021).

In addition, even if we apply the architecture to a specific prediction task for the *default frequency* with a *localGLMnet* under binomial distribution we point out that choosing the appropriate link function on the last layer and the corresponding deviance loss, we can generalize the architecture for any response that gets a distribution from the *exponential distribution* class. Furthermore, we can explore an improved architecture inspired by the *Explainable Neural Network* of Vaughan, Sudjianto, Brahimi, Chen, and Nair (2018) for a *double glm* fitting when dealing with both frequencies and severity's. In that case we can use two parallel *localGLMnet* to train the two components of the response and adding a *dot layer* to combine the two predictions before entering the last *severity prediction* node and with a joint *deviance loss*, note that in that case it may not be *tweedie*.

Chapter 7

Appendices

7.1 Preprocessing in R

7.1.1 Cleaning by interpolation

```r
predicted_vals <- c()

for (col in missing_cols) {

  y_train <- loan_complete[[col]]
  x_train <- loan_complete[,!(names(loan_complete) %in% missing_cols)]
  x_train <- x_train[,!(names(x_train) %in% pb_variable_level)]
  # Train the regression model
  formula_str <- paste(col, "~", paste(predictor_cols, collapse = " + "))
  model <- lm(formula_str, data = loan_complete)

  # Use the trained model to predict missing values in the test set
  na_rows <- is.na(loan_missing[[col]])

  x_test <- loan_missing[na_rows,!(names(loan_missing) %in% missing_cols)]
  x_test <- x_test[,!(names(x_test) %in% pb_variable_level)]
  y_pred <- predict(model, newdata = x_test)


  # Replace the missing values in the original dataframe with the predicted values
  predicted_vals[[col]] <- y_pred
}
```

7.1.2 Levels reduction with K-means (example with emp_length)

```
xtabs(~emp_length+loan_status, data = new_data)
tab <- as.data.frame(rowPerc(xtabs(~emp_length+loan_status, data = new_data)))
tab <- cbind(tab[tab$loan_status == "Charged Off",], tab[tab$loan_status == "Fully Paid",])
tab <- tab[, c(1,3,6)]
colnames(tab) <- c("emp_length", "Charged_Off", "Fully_Paid")


ggplot(tab, aes(x = Charged_Off, y = Fully_Paid)) + geom_point(aes(color = emp_length))

rownames(tab) <- tab$emp_length
tab <- tab[, -1]

fviz_nbclust(tab, kmeans, method = "silhouette", k.max = 8)
fviz_nbclust(tab, kmeans, method = "wss", k.max = 8)

km_emp_length <- kmeans(tab, 4, nstart = 25)

levels(new_data$emp_length) <- (as.data.frame(km_emp_length$cluster))[,1]

fviz_cluster(km_emp_length, tab, ggtheme = theme_bw(), labelsize = 15) +
    ggtitle(label="emp_length levels clustering")
```

7.2 R code for logistic regression and stepwise selection

```

#######################
# logistic regression #
#######################


Dev.Ber = function(y,p){
  eps=1e-16
  -2*sum(y*log((p+eps)/(1-p+eps)) + log(1-p+eps))
}

odds_to_be_preproc <- new_data[, c("delinq_2yrs", "total_rec_late_fee")]

drop <- c("delinq_2yrs", "recoveries", "total_rec_late_fee", "desc", "title", "emp_title",
    "recoveries", "inq_last_6mths", "")
new_data <- new_data[,!(names(new_data) %in% drop)]

num_vars <- names(new_data)[sapply(new_data, is.numeric)]

```

```
19
20 # Cut numeric features into quartiles
21 for (var in num_vars) {
22   breaks <- unique(quantile(new_data[[var]], probs = seq(0, 1, 0.25)))
23   new_data[[var]] <- as.factor(cut(new_data[[var]], breaks = breaks, include.lowest = TRUE,
       labels = c("_first_q", "_second_q", "_third_q", "_fourth_q")))
24 }
25
26 new_data <- cbind(new_data, odds_to_be_preproc)
27
28
29 cut_points1 <- c(-1,0,3,4,11)
30 cut_point2 <- c(-1,0,15,30,45,60,181)
31
32 # Cut the continuous feature using the specified cut points
33 new_data$delinq_2yrs <- cut(new_data$delinq_2yrs, breaks = cut_points1)
34 new_data$total_rec_late_fee <- cut(new_data$total_rec_late_fee, breaks = cut_point2)
35
36 missing_counts <- colSums(is.na(new_data))
37 print(missing_counts)
38
39
40
41 Categ.to.Quant<-function(data,factor,removeLast=TRUE)
42 {
43   y = paste("data$",sep = "",factor)
44   x = eval(parse(text=y))
45   ndata = length(x)       #number of lines in the dataset
46   nlgen = length(levels(x)) #number of levels
47   if (!removeLast)
48   {nlgen = nlgen+1} #number of levels
49   lev   = levels(x)
50   z     = matrix(0,ndata,nlgen-1)
51   nv    = vector("character",nlgen-1)
52   for (ct in 1:nlgen-1)
53   {
54     z[,ct] = ifelse(x==lev[ct],1,0)
55     nv[ct] = paste(factor,sep="",lev[ct])
56   }
57   colnames(z)=nv
58   #remove the column
59   data <- data[, ! names(data) %in% factor, drop = F]
60   data <- data.frame(data,z)
61   return(data)
62 }
63
```

```r
cols_with_periodicity <- grep("month|year", names(new_data))
df_glm <- subset(new_data, select = -cols_with_periodicity)

for (col in names(df_glm)) {
  if (col != "loan_status") {
    df_glm <- Categ.to.Quant(df_glm, col)
  }
}

df_glm$loan_status <- ifelse(df_glm$loan_status == "Charged Off", 1, 0)

set.seed(87031800)

train_idx <- sample(nrow(df_glm), nrow(df_glm) * 0.8)
train <- df_glm[train_idx, ]
test <- df_glm[-train_idx, ]


# Build a logistic regression model using all predictors
model <- glm(loan_status ~ ., family = binomial(link="logit"), data = train)
summary(model)

step.model <- stepAIC(model, direction = "both", trace = TRUE)
summary(step.model)

predict.train <- predict.glm(model, train, type = "response")
predict.test <- predict.glm(model, test, type = "response")

# Make predictions on the test set
predict.train.restr <- predict.glm(step.model, train, type = "response")
predict.test.rest <- predict.glm(step.model, test, type = "response")

# Evaluate the performance of the model

Dev.Ber(train$loan_status, predict.train)
Dev.Ber(test$loan_status, predict.test)

Dev.Ber(train$loan_status, predict.train.restr)
Dev.Ber(test$loan_status, predict.test.rest)

AIC(model)
AIC(step.model)

#########################
# Sensivity analysis s #
#########################
```

```r

library(leaps)
y <- df_glm$loan_status
X <- as.matrix(df_glm[, -1])
set.seed(87031800)
# Define the number of repetitions and subsets
nreps <- 10
nsubsets <- 5

# Create a matrix to store the results
results <- matrix("", nsubsets, nreps)

# Perform stepwise selection on each subset of the data
for (i in 1:nreps) {
  # Randomly split the data into subsets
  subsets <- split(sample(nrow(X)), rep(1:nsubsets, length.out = nrow(X)))

  # Perform stepwise selection on each subset
  for (j in 1:nsubsets) {
    # Select the subset
    Xsubset <- X[subsets[[j]], ]
    ysubset <- y[subsets[[j]]]

    # Perform stepwise selection using AIC as the criterion
    model <- regsubsets(Xsubset, ysubset, method = "backward", nvmax = ncol(Xsubset),
                        really.big = TRUE, intercept = TRUE, criterion = "aic")

    # Store the selected features in the results matrix
    results[j, i] <- paste0(colnames(Xsubset)[summary(model)$which], collapse = ",")
  }
}

# Print the results
print(results)
```

7.3 R code for elastic net with CV

```r
    # Split data into 5 folds

folds <- createFolds(train$loan_status, k = 5)

# Define lambda and alpha values to search over
lambda_vals <- c(0.1,0.05, 0.01,0.005, 0.001,0.0005, 0.0001, 0.00005, 0.00005)
alpha_vals <- seq(0, 1, length = 11)[-1]
```

```r

# Initialize matrix to store cross-validation results
cv_results <- matrix(NA, nrow = length(lambda_vals) * length(alpha_vals), ncol = 5)
colnames(cv_results) <- c("lambda", "alpha", "number of folds", "dev.bern.train",
    "dev.bern.test")
row_idx <- 1

# Loop over lambda and alpha values
for (i in 1:length(lambda_vals)) {
  for (j in 1:length(alpha_vals)) {
    # Initialize vector to store Dev.Bern results for this combination of lambda and alpha
    dev_bern_results <- rep(NA, 5)
    dev_bern_results_train <- rep(NA, 5)
    # Loop over folds
    nfolds <- 0
    for (fold in 1:5) {
      set.seed(87031800)
      # Split data into training and test sets for this fold
      train_idx <- setdiff(seq_len(nrow(df_glm)), folds[[fold]])
      test_idx <- folds[[fold]]
      x_train <- as.matrix(df_glm[train_idx, -1])
      y_train <- df_glm$loan_status[train_idx]
      x_test <- as.matrix(df_glm[test_idx, -1])
      y_test <- df_glm$loan_status[test_idx]

      # Fit model with elastic net regularization
      model <- glmnet(x_train, y_train, family = "binomial", lambda = lambda_vals[i], alpha =
          alpha_vals[j])

      # Compute predicted probabilities for test set
      y_pred <- predict(model, newx = x_test, type = "response")
      y_pred_train <- predict(model, newx = x_train, type = "response")
      # Compute Dev.Bern loss for test set
      dev_bern <- Dev.Ber(y_test, y_pred)
      dev_bern_train <- Dev.Ber(y_train, y_pred_train)

      # Store results for this fold
      dev_bern_results[fold] <- dev_bern
      dev_bern_results_train[fold] <- dev_bern_train
      nfolds <- nfolds + 1
    }

    # Compute mean Dev.Bern loss across folds for this combination of lambda and alpha
    mean_dev_bern <- mean(dev_bern_results)
    mean_dev_bern_train <- mean(dev_bern_results_train)
    # Store results in cv_results matrix
```

```r
    cv_results[row_idx, 1] <- lambda_vals[i]
    cv_results[row_idx, 2] <- alpha_vals[j]
    cv_results[row_idx, 3:5] <- c(nfolds, mean_dev_bern_train, mean_dev_bern)

    # Increment row index
    row_idx <- row_idx + 1
  }
}
```

7.4 Python Code LocalGLMnet

7.4.1 Python Code for Hyperband keras tuner

```python
def build_model(hp):
    inp_num = keras.layers.Input(shape=(Xtrain.shape[1],), name='Input_var')
    x = inp_num
    # Tune the number of layers.
    for i in range(hp.Int("num_layers", 1, 6)):
        x = layers.Dense(
            units=hp.Int(f"units_{i}", min_value=5, max_value=65, step=1),
            activation=hp.Choice("activation", ["relu"]),
            kernel_regularizer=regularizers.l2(hp.Float(f"l2_reg_{i}", 0.0, 0.1, step=0.01)),
        )(x)
    Attention = layers.Dense(Xtrain.shape[1], activation="linear", name='Attention')(x)
    Dot = layers.Dot(axes=1)([inp_num, Attention])
    Response = layers.Dense(1, activation=tf.keras.activations.sigmoid, name='Response')(Dot)
    # Tune the learning rate
    learning_rate = hp.Choice("learning_rate", values=[1e-5, 1e-4, 1e-3])
    model = keras.Model(inputs=inp_num, outputs=Response)

    # Compile the model with the custom loss function
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
        loss=Deviance(eps=1e-05, w=1)
    )
    return model
```

```python
# Define the hyperparameter search space
tuner = keras_tuner.Hyperband(
    build_model,
    objective="val_loss",
    max_epochs=1000,
    factor=3,
```

```
    seed = 87031800,
    directory="tuning_dir",
    project_name="my_model_bayes_hyperband_final_full",
    hyperband_iterations=3
)


# Start the hyperparameter search

tuner.search(Xtrain, Ytrain, epochs=200, validation_split=0.2, batch_size=len(Xtrain),
    callbacks=[tf.keras.callbacks.EarlyStopping('val_loss', patience=20)])
```

7.4.2 Python Code for optimal localGLMnet

```
inp_num = keras.layers.Input(shape=(Xtrain.shape[1],), name = 'Input_var')
dense1 = keras.layers.Dense(28, activation=tf.nn.relu, kernel_regularizer=regularizers.l2(0.1),
    name = '1')(inp_num)
dense2 = keras.layers.Dense(23, activation=tf.nn.relu,
    kernel_regularizer=regularizers.l2(0.08), name = '2')(dense1)
dense3 = keras.layers.Dense(59, activation=tf.nn.relu, name = '3')(dense2)
dense4 = keras.layers.Dense(19, activation=tf.nn.relu,
    kernel_regularizer=regularizers.l2(0.04), name = '4')(dense3)
dense5 = keras.layers.Dense(39, activation=tf.nn.relu, name = '5')(dense4)
dense6 = keras.layers.Dense(51, activation=tf.nn.relu,
    kernel_regularizer=regularizers.l2(0.01), name = '6')(dense5)
Attention = keras.layers.Dense(Xtrain.shape[1], activation="linear", name = 'Attention')(dense6)

Dot = keras.layers.Dot(axes=1)([inp_num, Attention])

Response = keras.layers.Dense(1, activation= tf.keras.activations.sigmoid, name =
    'Response')(Dot)
model = keras.Model(inputs=inp_num, outputs=Response)
print(model.summary())

callback = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=20)
model.compile(loss= Deviance(eps=1e-5, w=1), optimizer=tf.optimizers.Adam(0.001))
history = model.fit(Xtrain, Ytrain, epochs=1000, batch_size = len(Xtrain), callbacks =
    [callback], validation_split=0.2, verbose=2)
```

7.4.3 Pyhton Code for Coefficient extraction

```
zz = keras.Model(inputs = model.inputs, outputs = model.get_layer("Attention").output)
print(zz.summary())
```

```
b0 = float(model.get_layer("Response").get_weights()[0])
beta_x = zz.predict(Xtrain)
beta_x = beta_x * b0
beta_x_df = pd.DataFrame(beta_x)
beta_x_df.columns = ['Beta_' + col_name + "*x" for col_name in Xtrain.columns]
beta_x_df.to_csv('beta_x.csv', index=False)

# concat coeff and plots
Xtrain = Xtrain.reset_index(drop=True)

coef_plot_train = pd.concat([Xtrain, beta_x_df], axis=1)

### sample to plot

nsample = len(coef_plot_train)
np.random.seed(87031800)
idx = np.random.choice(coef_plot_train.shape[0], nsample, replace=False)

beta_to_plot = coef_plot_train.iloc[idx]
train_to_plot = Xtrain.iloc[idx]

# define the line size and quant_rand values

line_size = 1
quant_rand = 0.5

num_names.append("random")

# get the list of feature names and beta feature names
name_train = [col for col in Xtrain.columns if any(name in col for name in num_names)]
name_beta = [col for col in beta_x_df.columns if any(name in col for name in num_names)]


# loop over each feature and beta feature pair
for feature, beta_feature in zip(name_train, name_beta):
    # create a data frame with the data to plot
    dat_plt = pd.DataFrame({'var': beta_to_plot[feature],
                            'bx': beta_to_plot[beta_feature],
                            'col': ['black'] * nsample})
    # create the scatter plot
    fig = plt.figure()
    ax = sns.scatterplot(data=dat_plt, x='var', y='bx', color='black',alpha=0.5, sizes=[1])
    ax.axhline(y=0, color='red', linewidth=line_size)
    ax.axhline(y=-0.5, color='green', linewidth=line_size)
    ax.axhline(y=0.5, color='green', linewidth=line_size)
    ax.axhline(y=-1/4, color='orange', linewidth=line_size, linestyle='dashed')
```

```
    ax.axhline(y=1/4, color='orange', linewidth=line_size, linestyle='dashed')
    ax.axvspan(xmin=min(dat_plt['var']), xmax=max(dat_plt['var']), ymin=-quant_rand,
        ymax=quant_rand, alpha=0.002, facecolor='green')
    ax.set(title="Regression attention",
           xlabel=feature,
           ylabel="regression attention")

    # add a smooth line using UnivariateSpline
    from scipy.interpolate import UnivariateSpline
    sorted_data = dat_plt.sort_values('var')
    spl = UnivariateSpline(sorted_data['var'], sorted_data['bx'], k=4)
    x_new = np.linspace(sorted_data['var'].min(), sorted_data['var'].max(), 300)
    smooth_bx = spl(x_new)
    ax.plot(x_new, smooth_bx, color='purple', linewidth=2.5)

    # show the plot
    fig.show()
```

7.4.4 Pseudo Code for deviance based feature selection

Algorithm 8 Feature selection algorithm for minimizing binomial deviance

Require: Input data X and labels Y, initial model with all features, binomial deviance named Deviance to be minimized

Initialize *current_deviance* with ∞, *remaining_cols* with all feature columns, *best_deviance* with *current_deviance*, *best_removed_col* with *None*, *num_vars_saturated* with the number of features, *num_vars_reduced* with *num_vars_saturated*, and an empty list *deviances*
while len(*remaining_cols*) > 0 **do**
  num_vars_reduced $\leftarrow$ *num_vars_reduced* - 1
  for col in *remaining_cols* **do**
    reduced_Xtrain $\leftarrow$ *Xtrain*.drop(col, axis=1)
    reduced_Xtest $\leftarrow$ *Xtest*.drop(col, axis=1)
    Define model architecture using *reduced_Xtrain* and train with Deviance loss using *Deviance*($eps = 1e-5, w = 1$) and *Adam* optimizer with a learning rate of 0.001 for 1000 epochs and a batch size of len(*reduced_Xtrain*), using early stopping callback with a patience of 20
    Compute the deviance of the reduced model on the reduced test set and store it in *deviance_value*
    Compute the $\widehat{\mu}$ of the reduced model and store it
    Compute the test statistic and $p - value$ based on *deviance_value* and *deviance_saturated* using chi2.cdf function with df = *num_vars_saturated* $-$ *num_vars_reduced*
    if *p_value* < 0.05 **then**
      indic $\leftarrow$ 'to include'
    else
      indic $\leftarrow$ 'not to include'
    end if
    Store the results in *deviance_dict* with the removed feature as key
    Check if the deviance of the reduced model is better than the current best deviance. If yes, update *best_deviance* with *deviance_value* and *best_removed_col* with col
  end for
  if *best_deviance* $<$ *current_deviance* **then**
    Update *current_deviance* with *best_deviance*, add (*best_removed_col*, *current_deviance*) to *deviances*, remove *best_removed_col* from *remaining_cols*, *Xtrain*, and *Xtest*
  else
    Break the while loop
  end if
end while

7.4.5 Python Code for Gradient extraction

```
class GradientLayer(tf.keras.layers.Layer):
```

```
    def __init__(self, target_tensor_idx, **kwargs):
        self.target_tensor_idx = target_tensor_idx
        super(GradientLayer, self).__init__(**kwargs)

    def call(self, inputs):
        target_tensor = inputs[self.target_tensor_idx]
        grad_fn = K.gradients(target_tensor, inputs)
        return grad_fn

    def compute_output_shape(self, input_shape):
        return input_shape
```

```
for j in range(Xtrain.shape[1]):
    beta_j = tf.keras.layers.Lambda(lambda x: x[:, j])(Attention)
    grad_layer = GradientLayer(target_tensor_idx=0)
    grad = grad_layer([beta_j, inp_num])
    model_grad = keras.Model(inputs=inp_num, outputs=grad)
    grad_beta = model_grad.predict(Xtrain)
    grad_computed = pd.DataFrame(grad_beta[1])
    varname = Xtrain.columns[j]
    grad_computed.to_csv(f"grad_compute_{varname}.csv", index=False)
```

7.4.6 Hyperband optimal hyperparameters with embeddings

Hyperparameter	Value
num_layers	4
units_0	93
activation	relu
l2_reg_0	0.07
units_1	97
l2_reg_1	0.05
units_2	69
l2_reg_2	0.0
units_3	93
l2_reg_3	0.05
learning_rate	0.001
tuner/epochs	112

(a) Optimal hyperparameters 1 text features

Hyperparameter	Value
num_layers	10
units_0	73
activation	relu
l2_reg_0	0.06
units_1	41
l2_reg_1	0.06
units_2	51
l2_reg_2	0.0
units_3	51
l2_reg_3	0.09
learning_rate	0.0001
units_4	81
l2_reg_4	0.1
units_5	31
l2_reg_5	0.01
units_6	29
l2_reg_6	0.02
units_7	81
l2_reg_7	0.07
units_8	35
l2_reg_8	0.06
units_9	75
l2_reg_9	0.09
tuner/epochs	334

(b) Optimal hyperparameters 2 text features

Hyperparameter	Value
num_layers	5
units_0	75
activation	relu
l2_reg_0	0.01
units_1	73
l2_reg_1	0.06
units_2	87
l2_reg_2	0.02
units_3	53
l2_reg_3	0.0
learning_rate	0.001
units_4	71
l2_reg_4	0.1
tuner/epochs	112

(c) Optimal hyperparameters 3 text features

Figure 7.1: Hyperband tuned with embeddings

7.4.7 Pseudo Code for TSNE representation of embeddings

Algorithm 9 t-SNE algorithm for dimensionality reduction

Require: High-dimensional data set $\mathbf{X} = \mathbf{x}_1, \ldots, \mathbf{x}_N$, number of dimensions d

1: Compute the pairwise similarities $p_{j|i}$ using Gaussian kernel with

$$p_{j|i} = \frac{\exp(-|\mathbf{x}_i - \mathbf{x}j|^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-|\mathbf{x}_i - \mathbf{x}_k|^2/2\sigma_i^2)}$$

2: Initialize the low-dimensional embedding $\mathbf{Y} = \mathbf{y}_1, \ldots, \mathbf{y}_N$ randomly

3: **for** $t = 1, \ldots, T$ **do**

4: Compute the pairwise similarities $q_{j|i}$ using Student-t distribution with one degree of freedom:

$$q_{j|i} = \frac{(1+|\mathbf{y}_i - \mathbf{y}_j|^2)^{-1}}{\sum k \neq i (1+|\mathbf{y}_i - \mathbf{y}_k|^2)^{-1}}$$

5: Compute the gradient $\frac{\partial C}{\partial \mathbf{y}_i}$ of the Kullback-Leibler divergence cost function with respect to $\mathbf{y}i$:

$$\frac{\partial C}{\partial \mathbf{y}i} = 4 \sum_j (pj|i - qj|i)(\mathbf{y}_i - \mathbf{y}_j)(1 + |\mathbf{y}_i - \mathbf{y}_j|^2)^{-1}$$

6: Update the low-dimensional embedding using gradient descent with momentum:

$$\mathbf{Y} \leftarrow \mathbf{Y} - \eta \frac{\partial C}{\partial \mathbf{Y}} + \alpha(\mathbf{Y}^{t-1} - \mathbf{Y}^{t-2})$$

7: **end for**

8: **return** The low-dimensional embedding $\mathbf{Y}$

7.4.8 Python code for TSNE representation of embeddings

```
embedding_layer1 = model.get_layer('embedding_et')
embedding_layer2 = model.get_layer('embedding_d')
embedding_layer3 = model.get_layer('embedding_t')

learned_embeddings1 = embedding_layer1.get_weights()[0]
learned_embeddings2 = embedding_layer2.get_weights()[0]
learned_embeddings3 = embedding_layer3.get_weights()[0]

from sklearn.cluster import KMeans
# set the range of number of clusters to try
cluster_range = range(1, 31)

# define a function to compute the WCSS for each number of clusters
def compute_wcss(embedding):
    wcss_list = []
    for n_clusters in cluster_range:
        kmeans = KMeans(n_clusters=n_clusters, random_state=0)
        kmeans.fit(embedding)
        wcss = kmeans.inertia_
```

```
        wcss_list.append(wcss)
    return wcss_list

# compute the WCSS for each embedding space
wcss_list1 = compute_wcss(learned_embeddings1)
wcss_list2 = compute_wcss(learned_embeddings2)
wcss_list3 = compute_wcss(learned_embeddings3)

# plot the WCSS against the number of clusters
plt.plot(cluster_range, wcss_list1, label='Embedding 1')
plt.plot(cluster_range, wcss_list2, label='Embedding 2')
plt.plot(cluster_range, wcss_list3, label='Embedding 3')
plt.xlabel('Number of clusters')
plt.ylabel('Within-cluster sum of squares (WCSS)')
plt.title('Elbow Method for Optimal Number of Clusters')
plt.legend()
plt.show()


kmeans1 = KMeans(n_clusters=3, random_state=0).fit(learned_embeddings1)
labels1 = kmeans1.labels_
kmeans2 = KMeans(n_clusters=3, random_state=0).fit(learned_embeddings2)
labels2 = kmeans2.labels_
kmeans3 = KMeans(n_clusters=3, random_state=0).fit(learned_embeddings3)
labels3 = kmeans3.labels_


# create a dictionary that maps index to word for each feature
index_to_word1 = {index: word for word, index in word_index1.items()}
index_to_word2 = {index: word for word, index in word_index2.items()}
index_to_word3 = {index: word for word, index in word_index3.items()}

# create a dictionary that maps index to embedding for each feature
index_to_embedding1 = {index: embedding for index, embedding in enumerate(learned_embeddings1)}
index_to_embedding2 = {index: embedding for index, embedding in enumerate(learned_embeddings2)}
index_to_embedding3 = {index: embedding for index, embedding in enumerate(learned_embeddings3)}

# create a dictionary that maps cluster label to embeddings for each feature
cluster_embedding_dict1 = {}
cluster_embedding_dict2 = {}
cluster_embedding_dict3 = {}

for i, cluster_label in enumerate(labels1):
    # get the word index for the first feature
    word_index1 = index_to_word1.get(i+1) # add 1 to index to account for reserved index 0

```

```
    # get the embedding for the first feature
    embedding1 = index_to_embedding1[i]

    # add the embedding to the dictionary for the first feature
    if cluster_label not in cluster_embedding_dict1:
        cluster_embedding_dict1[cluster_label] = [embedding1]
    else:
        cluster_embedding_dict1[cluster_label].append(embedding1)

for i, cluster_label in enumerate(labels2):
    # get the word index for the second feature
    word_index2 = index_to_word2.get(i+1) # add 1 to index to account for reserved index 0

    # get the embedding for the second feature
    embedding2 = index_to_embedding2[i]

    # add the embedding to the dictionary for the second feature
    if cluster_label not in cluster_embedding_dict2:
        cluster_embedding_dict2[cluster_label] = [embedding2]
    else:
        cluster_embedding_dict2[cluster_label].append(embedding2)

for i, cluster_label in enumerate(labels3):
    # get the word index for the third feature
    word_index3 = index_to_word3.get(i+1) # add 1 to index to account for reserved index 0

    # get the embedding for the third feature
    embedding3 = index_to_embedding3[i]

    # add the embedding to the dictionary for the third feature
    if cluster_label not in cluster_embedding_dict3:
        cluster_embedding_dict3[cluster_label] = [embedding3]
    else:
        cluster_embedding_dict3[cluster_label].append(embedding3)

from sklearn.metrics.pairwise import cosine_similarity

# compute the average cosine similarity for each cluster and feature
avg_similarity_dict1 = {}
avg_similarity_dict2 = {}
avg_similarity_dict3 = {}

for cluster_label in cluster_embedding_dict1:
    # compute the average embedding for each cluster and feature
    avg_embedding1 = np.mean(cluster_embedding_dict1[cluster_label], axis=0)
    avg_embedding2 = np.mean(cluster_embedding_dict2[cluster_label], axis=0)
```

```
    avg_embedding3 = np.mean(cluster_embedding_dict3[cluster_label], axis=0)

    # compute the cosine similarity between each word embedding and the average embedding
    similarity1 = cosine_similarity(avg_embedding1.reshape(1,-1), learned_embeddings1)
    similarity2 = cosine_similarity(avg_embedding2.reshape(1,-1), learned_embeddings2)
    similarity3 = cosine_similarity(avg_embedding3.reshape(1,-1), learned_embeddings3)

    # get the indices of the words in the cluster
    cluster_indices = [i for i, label in enumerate(labels1) if label == cluster_label]

    # compute the average cosine similarity for each feature
    avg_similarity1 = np.mean(similarity1[:, cluster_indices], axis=1)
    avg_similarity2 = np.mean(similarity2[:, cluster_indices], axis=1)
    avg_similarity3 = np.mean(similarity3[:, cluster_indices], axis=1)

    # add the average cosine similarity to the dictionary for each feature
    avg_similarity_dict1[cluster_label] = avg_similarity1[0]
    avg_similarity_dict2[cluster_label] = avg_similarity2[0]
    avg_similarity_dict3[cluster_label] = avg_similarity3[0]


tsne_embeddings = TSNE(n_components=3).fit_transform(learned_embeddings3)

import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import numpy as np

# Define custom colormap
colors = ['red', 'black', 'yellow']
cmap = mcolors.ListedColormap(colors)

fig = plt.figure(figsize=(11, 11))
ax = fig.add_subplot(111, projection='3d')
scatter = ax.scatter(tsne_embeddings[:, 0], tsne_embeddings[:, 1], tsne_embeddings[:, 2],
    c=labels3, cmap=cmap)

# Add average similarity values to corresponding clusters
unique_labels = np.unique(labels3)

avg_sim_labels = []
for label in unique_labels:
    avg_sim = avg_similarity_dict3[label]
    avg_sim_str = f"{avg_sim:.3f}"
    avg_sim_labels.append(avg_sim_str)

# Create custom legend
```

```
handles, labels = scatter.legend_elements()
legend_labels = ['Cluster 1', 'Cluster 2', 'Cluster 3']
legend_handles = handles
legend = ax.legend(legend_handles, legend_labels, loc='lower left', title='Clusters',
    frameon=True)
legend.get_title().set_fontsize(16)

# Add second legend for average similarity
ax.add_artist(legend)
avg_sim_legend = ax.legend(legend_handles, avg_sim_labels, loc='lower right',
    title='avg_sim_clusters', frameon=True)
avg_sim_legend.get_title().set_fontsize(16)

# Set plot title
avg_similarity3f = avg_similarity3[0]
ax.set_title(f'Average Similarity = {avg_similarity3f:.3f}', fontsize=20)

# Set axis labels
ax.set_xlabel('t-SNE Dimension 1', fontsize=16)
ax.set_ylabel('t-SNE Dimension 2', fontsize=16)
ax.set_zlabel('t-SNE Dimension 3', fontsize=16)

plt.show()
```

7.4.9 Hyperband with trained pre-trained embeddings

Hyperparameter	Value
num_layers	6
units_0	91
activation	relu
l2_reg_0	0.02
units_1	73
l2_reg_1	0.09
units_2	63
l2_reg_2	0.02
units_3	61
l2_reg_3	0.05
learning_rate	0.0001
units_4	83
l2_reg_4	0.08
units_5	99
l2_reg_5	0.02
tuner/epochs	1000

(a) Optimal hyperparameters 1 text features

Hyperparameter	Value
num_layers	4
units_0	87
activation	relu
l2_reg_0	0.1
units_1	33
l2_reg_1	0.07
units_2	97
l2_reg_2	0.1
units_3	95
l2_reg_3	0.04
learning_rate	0.001
tuner/epochs	112

(b) Optimal hyperparameters 2 text features

Hyperparameter	Value
num_layers	5
units_0	87
activation	relu
l2_reg_0	0.04
units_1	83
l2_reg_1	0.1
units_2	85
l2_reg_2	0.01
units_3	91
l2_reg_3	0.02
learning_rate	0.001
units_4	87
l2_reg_4	0.08
tuner/epochs	112

(c) Optimal hyperparameters 3 text features

Figure 7.2: Hyperband tuned with trained pre-trained embeddings

References

Bahdanau, D., Cho, K., & Bengio, Y. (2016). *Neural machine translation by jointly learning to align and translate.* Retrieved from https://arxiv.org/pdf/1409.0473.pdf

Cai, T. T., & Ma, R. (2022). *Theoretical foundations of t-sne for visualizing high-dimensional clustered data.* Retrieved from https://arxiv.org/pdf/2105.07536.pdf

Dimitriadis, T., Dümbgen, L., Henzi, A., Puke, M., & Ziegel, J. (2022). Honest calibration assessment for binary outcome predictions. *Biometrika*. Retrieved from https://doi.org/10.1093%2Fbiomet%2Fasac068 doi: 10.1093/biomet/asac068

Geiping, J., Goldblum, M., Pope, P. E., Moeller, M., & Goldstein, T. (2022). *Stochastic training is not necessary for generalization.* Retrieved from https://arxiv.org/pdf/2109.14119.pdf

Grari, V., Charpentier, A., & Detyniecki, M. (2022). *A fair pricing model via adversarial learning.* Retrieved from https://arxiv.org/pdf/2202.12008.pdf

Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning: Data mining, inference and prediction* (2nd ed.). Springer. Retrieved from http://www-stat.stanford.edu/~tibs/ElemStatLearn/

Hrinchuk, O., Khrulkov, V., Mirvakhabova, L., Orlova, E., & Oseledets, I. (2020). *Tensorized embedding layers for efficient model compression.* Retrieved from https://arxiv.org/pdf/1901.10787.pdf

Kenna, D. (2021). *Using adversarial debiasing to remove bias from word embeddings.* Retrieved from https://arxiv.org/pdf/2107.10251.pdf

Keshari, R., Singh, R., & Vatsa, M. (2018). *Guided dropout.* Retrieved from https://arxiv.org/pdf/1812.03965.pdf

Labach, A., Salehinejad, H., & Valaee, S. (2019). *Survey of dropout methods for deep neural*

networks. Retrieved from https://arxiv.org/pdf/1904.13310.pdf

Lee, G., Park, C., & Shin, K. (2022). *Set2box: Similarity preserving representation learning of sets.* Retrieved from https://arxiv.org/pdf/2210.03282.pdf

Lehmann, E. L. (2006). On likelihood ratio tests. , 1–8. Retrieved from https://doi.org/10.1214%2F074921706000000356 doi: 10.1214/074921706000000356

Li, L., Jamieson, K., DeSalvo, G., Rostamizadeh, A., & Talwalkar, A. (2018). Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, *18*(185), 1–52. Retrieved from http://jmlr.org/papers/v18/16-558.html

Li, Y., Shen, Y., Jiang, H., Zhang, W., Li, J., Liu, J., ... Cui, B. (2022). *Hyper-tune: Towards efficient hyper-parameter tuning at scale.* Retrieved from https://arxiv.org/pdf/2201.06834.pdf

Lu, J. (2022). *Gradient descent, stochastic optimization, and other tales.* Retrieved from https://arxiv.org/pdf/2205.00832.pdf

Lu, Y., & Lu, J. (2020). *A universal approximation theorem of deep neural networks for expressing probability distributions.* Retrieved from https://arxiv.org/pdf/2004.08867.pdf

Manzini, T., Lim, Y. C., Tsvetkov, Y., & Black, A. W. (2019). *Black is to criminal as caucasian is to police: Detecting and removing multiclass bias in word embeddings.* Retrieved from https://arxiv.org/pdf/1904.04047.pdf

Neyshabur, B., Tomioka, R., & Srebro, N. (2015). *In search of the real inductive bias: On the role of implicit regularization in deep learning.* Retrieved from https://arxiv.org/pdf/1412.6614.pdf

Parhi, R., & Nowak, R. D. (2022). What Kinds of Functions Do Deep Neural Networks Learn? Insights from Variational Spline Theory. *SIAM Journal on Mathematics of Data Science*, *4*(2), 464–489. Retrieved from https://doi.org/10.1137%2F21m1418642 doi: 10.1137/21m1418642

Richman, R., & Wüthrich, M. V. (2021). Localglmnet: interpretable deep learning for tabular data. *CoRR*, *abs/2107.11059*. Retrieved from https://arxiv.org/abs/2107.11059

Tahmassebi, A. (2018). Multi-stage optimization of a deep model: A case study on ground motion modeling. Retrieved from https://www.researchgate.net/publication/327761873_Multi-stage_optimization_of_a_deep_model_A_case_study_on_ground_motion_modeling/link/5ba3168e92851ca9ed174f23/download doi:

10.1371/journal.pone.0203829

Vaughan, J., Sudjianto, A., Brahimi, E., Chen, J., & Nair, V. N. (2018). *Explainable neural networks based on additive index models.* Retrieved from https://arxiv.org/pdf/1806.01933.pdf

Wüthrich, M. V. (2022). Bias regularization in neural network models for general insurance pricing. *European Actuarial Journal*, *10*(2), 179–202. Retrieved from https://doi.org/10.1007/s13385-019-00215-z doi: 10.1007/s13385-019-00215-z

Xu, D., & Yang, B. (2023). Pretrained embeddings for e-commerce machine learning: When it fails and why? In *Companion proceedings of the ACM web conference 2023.* ACM. Retrieved from https://doi.org/10.1145%2F3543873.3587669 doi: 10.1145/3543873.3587669

Yuan, H., Liu, M., Kang, L., Miao, C., & Wu, Y. (2023). *An empirical study of the effect of background data size on the stability of shapley additive explanations (shap) for deep learning models.* Retrieved from https://arxiv.org/pdf/2204.11351.pdf