Supervised Learning with Generalized Linear Models
A matricial formulation

Bassam El Khoury Seguias

April 29, 2019

1 Introduction

The purpose of this note is to provide a **matricial** formulation of **statistical learning models** derived from the class of **exponential distributions with dispersion parameter**. We assume that the reader is comfortable with linear algebra and multivariable calculus, has an understanding of basic probability theory and is familiar with **supervised learning** concepts.

A number of regression and classification models commonly used in supervised learning settings turn out to be specific cases derived from the family of **exponential distributions**. This note is organized as follows:

1. Section 2 describes the family of exponential distributions and their associated **Generalized Linear Model (GLM)**. The family described in [3] counts a significant number of distributions including e.g., the univariate Gaussian, Bernoulli, Poisson, Geometric, and Multinomial cases. Other distributions such as the multivariate Gaussian lend themselves to a natural generalization of this model. In order to do so, we extend the family of exponential distributions with dispersion parameter [3] to include symmetric positive definite dispersion matrices.

2. Section 3 derives the GLM's **Cost Function** and its corresponding **Gradient** and **Hessian** all expressed in component form. We derive the expressions associated with the general case that includes a dispersion matrix. We also derive simplified versions for the specific case when the dispersion matrix is a positive scalar multiple of the identity matrix.

3. In Section 4, we limit ourselves to distributions whose dispersion matrix is a positive scalar multiple of the identity matrix. These are precisely the ones described in [3]. We express their associated Cost Function, Gradient and Hessian using concise matrix notation. We will separately analyze the case of the multivariate Gaussian distribution and derive its associated Cost Function and Gradient in matrix form in section 7.

4. Section 5 provides a matricial formulation of three numerical algorithms that can be used to minimize the Cost Function. They include the **Batch Gradient Descent**

(BGD), **Stochastic Gradient Descent (SGD)** and **Newton Raphson (NR)** methods.

5. Section 6 applies the matricial formulation to a select set of exponential distributions whose dispersion matrix is a positive scalar multiple of the identity. In particular, we consider the following:

   i. Univariate Gaussian distribution (which yields the familiar **linear regression** model)

   ii. Bernoulli distribution (which yields the familiar **logistic regression** model)

   iii. Poisson distribution

   iv. Geometric distribution

   v. Multinomial distribution (which yields the familiar **softmax regression** model)

6. Section 7 treats the case of the **multivariate Gaussian** distribution. It is an example of an exponential distribution with dispersion matrix that is not necessarily a positive scalar multiple of the identity matrix. In this case, the dispersion matrix turns out to be the precision matrix which is the inverse of the covariance matrix. We derive the corresponding Cost Function in component form and also express it using matrix notation. We then derive the Cost function's Gradient, express it in matrix notation and show how to to minimize the Cost Function using BGD. We finally consider the specific case of a non-weighted Cost Function without regularization and derive a closed-form solution for the optimal values of its minimizing parameters.

7. Section 8 provides a python script that implements the GLM Supervised Learning class using the matrix notation. We limit ourselves to cases where the dispersion matrix is a positive scalar multiple of the identity matrix. The code provided is meant for educational purposes and we recommend relying on existing and tested packages (e.g., scikit-learn) to run specific predictive models.

2 Generalized Linear Models (GLM)

Discriminative supervised learning models: Loosely speaking, a supervised learning model is an algorithm that when fed a **training set** (i.e., a set of inputs and their corresponding outputs) derives an **optimal** function that can make "good" output predictions when given new, previously unseen inputs.

More formally, let $\mathcal{X}$ an $\mathcal{Y}$ denote the input and output spaces respectively. Let

$$\{(x^{(i)}, y^{(i)}),\ i = 1, .., m\} \subset \mathcal{X} \text{ x } \mathcal{Y}$$

denote a given training set for a finite positive integer m. The supervised learning algorithm will output a function $h : \mathcal{X} \to \mathcal{Y}$, that optimizes a certain performance metric usually expressed in the form of a **Cost Function**. The function h can then be

applied to an input $x \in \mathcal{X}$ to predict its corresponding output $y \in \mathcal{Y}$. Whenever $\mathcal{Y}$ is limited to a discrete set of values, we refer to the learning problem as a **classification**. Otherwise, we call it a **regression**.

The exercise of conducting predictions in a deterministic world is futile. Inject that world with a dose of randomness and that exercise becomes worthwhile. In order to model randomness, we usually lean on probabilistic descriptions of the distribution of relevant variables. More specifically, we may assume certain distributions on the set of inputs, the set of outputs, or joint distributions on inputs and outputs taken together. For the purpose of this note, we limit ourselves to models that make assumptions on the distribution of the output given the input, without giving any consideration to the underlying distribution of the inputs themselves. Such models are referred to as **discriminative** learning models.

Generalized Linear models (GLM): In essence, a GLM consists of three elements.

Element #1: A random component modeled as an instance from a family of probability distributions of the form:

$$\boxed{p(y;\ \eta, \rho) = b(y, \rho)\ e^{\rho\ [\ \eta^T\ T(y)\ -\ a(\eta)\ ]}} \tag{1}$$

Modulo variables and maps naming, this family of distributions corresponds to the one introduced in [3]. It is defined for positive dispersion parameters $\rho \in \mathbb{R}^+$. We now introduce a more general version where the dispersion parameter ρ could be a symmetric positive definite matrix $\Lambda \in \mathbb{S}^p_{++},\ p \geq 1$. We define the enlarged family of exponential distributions to include those of the following form:

$$\boxed{\begin{aligned} p(y;\ \eta, \Lambda) \ &=\ b(y, \Lambda)\ e^{[\ \eta^T\ \Lambda\ T(y)\ -\ [q(\eta)]^T\ \Lambda\ q(\eta)\ ]} \\ &\equiv\ b(y, \Lambda)\ e^{[\ \eta^T\ \Lambda\ T(y)\ -\ c(\eta,\Lambda)\ ]} \end{aligned}} \tag{2}$$

where we define $c(\eta, \Lambda)$ to be equal to $[q(\eta)]^T\ \Lambda\ q(\eta)$. The maps and parameters appearing in the expression above are described as follows:

- $\eta \in \mathbb{R}^p$ is known as the **natural parameter** vector. We will soon impose a relationship between the input x and η, which will link the input to the output y.
- Λ is a symmetric element of $\mathbb{S}^p_{++}$ (i.e., a $(p \times p)$ symmetric positive-definite matrix). We refer to it as the **dispersion parameter** matrix.
- $b : \mathbb{R}^r \times \mathbb{S}^p_{++} \to \mathbb{R}^+$ is known as the **non-negative base measure**. It maps (y, Λ) to the positive scalar value $b(y, \Lambda)$.
- $T(y) \in \mathbb{R}^p$ is a **sufficient statistic** of $y \in \mathbb{R}^r$ and has the same dimension as η. For all practical purposes, this means that the probability of a random variable

taking on a particular value when conditioned on y is equal to the probability of it taking the same value when conditioned on $T(y)$. In other terms, whatever can be learned by conditioning on y can also be learned by conditioning on $T(y)$.

- $c : \mathbb{R}^p \times \mathbb{S}^p_{++} \rightarrow \mathbb{R}$ is known as the **log-partition function**. It maps (η, Λ) to $c(\eta, \Lambda) = [q(\eta)]^T \ \Lambda \ q(\eta)$, where q is a vector-valued map from $\mathbb{R}^p$ into $\mathbb{R}^p$ that depends only on η. We denote the components of the column vector $q(\eta)$ by $[q_1(\eta) \ .. \ q_p(\eta)]^T$, where each $q_i, i \in \{1, \ .., \ p\}$ is a map from $\mathbb{R}^p$ into $\mathbb{R}$.

  The rationale for the *log-partition* nomenclature stems from it being chosen to ensure that $p(y; \ \eta, \Lambda)$ integrates to 1. Doing so allows us to express $c(\eta, \Lambda)$ as a function of the other parameters:

$$\int_{-\infty}^{\infty} p(y; \ \eta, \Lambda) \ dy \ = \ 1 \iff \int_{-\infty}^{\infty} b(y, \Lambda) \ e^{[\ \eta^T \ \Lambda \ T(y) \ - \ c(\eta, \Lambda) \ ]} \ dy \ = \ 1 \iff$$

$$ln \ \{ \ \int_{-\infty}^{\infty} b(y, \Lambda) \ e^{[\ \eta^T \ \Lambda \ T(y) \ - \ c(\eta, \Lambda) \ ]} \ dy \ \} \ = \ 0 \iff$$

$$ln \ (\ e^{-c(\eta, \Lambda)} \ ) \ + \ ln \ \{ \ \int_{-\infty}^{\infty} b(y, \Lambda) \ e^{[\ \eta^T \ \Lambda \ T(y) \ ]} \ dy \ \} \ = \ 0 \iff$$

$$\boxed{c(\eta, \Lambda) \equiv [q(\eta)]^T \ \Lambda \ q(\eta) = ln \ \{ \ \int_{-\infty}^{\infty} b(y, \Lambda) \ e^{[\ \eta^T \Lambda \ T(y) \ ]} \ dy \ \}} \tag{3}$$

In what follows, we derive expressions for the log-partition function's Gradient and Hessian:

The log-partition function's Gradient: We start by defining the one-form Gradient of c with respect to η to be the quantity:

$$(\bigtriangledown_\eta \ c) \ \equiv \ [\ \frac{\partial c}{\partial \eta_1} \ .. \ \frac{\partial c}{\partial \eta_p} \ ]$$

In Euclidean $p-$space, the associated column vector Gradient is denoted by $(\bigtriangledown_\eta \ c)^T$. $\forall j \in \{1, \ ..., \ p\}$, we can write:

$$\frac{\partial c}{\partial \eta_j} \ = \ \frac{\partial}{\partial \eta_j} \ ln \ \{ \ \int_{-\infty}^{\infty} b(y, \Lambda) \ e^{[\ \eta^T \ \Lambda \ T(y) \ ]} \ dy \ \} =$$

$$\{ \ \int_{-\infty}^{\infty} b(y, \Lambda) \ e^{[\ \eta^T \ \Lambda \ T(y) \ ]} \ dy \ \}^{-1} \ \{ \ \int_{-\infty}^{\infty} [\Lambda \ T(y)]_j \ b(y, \Lambda) \ e^{[\ \eta^T \ \Lambda \ T(y) \ ]} \ dy \ \} =$$

$$\{ \ e^{-c(\eta, \Lambda)} \ \} \ \{ \ \int_{-\infty}^{\infty} [\Lambda \ T(y)]_j \ b(y, \Lambda) \ e^{[\ \eta^T \ \Lambda \ T(y) \ ]} \ dy \ \} =$$

$$\int_{-\infty}^{\infty} [\Lambda \ T(y)]_j \ b(y, \Lambda) \ e^{[\ \eta^T \ \Lambda \ T(y) \ - \ c(\eta, \Lambda) \ ]} \ dy \ = \ E[\ [\Lambda \ T(y)]_j; \ \eta, \Lambda \ ]$$

As a result, we conclude that:

$$\boxed{(\bigtriangledown_\eta \ c)^T = \Lambda \ E[\ T(y); \ \eta, \Lambda \ ]} \tag{4}$$

The log-partition function's Hessian: We first compute the second derivative of c with respect to the vector η as follows:

$$\frac{\partial^2 c}{\partial \eta_j \partial \eta_k} \;=\; \frac{\partial}{\partial \eta_k} \int_{-\infty}^{\infty} [\Lambda\, T(y)]_j \; b(y,\Lambda)\; e^{[\; \eta^T \;\Lambda\; T(y) \;-\; c(\eta,\Lambda)\;]}\, dy \;=$$

$$\int_{-\infty}^{\infty} [\Lambda\, T(y)]_j \; b(y,\Lambda)\; [\; [\Lambda\, T(y)]_k \;-\; \frac{\partial c}{\partial \eta_k}\;]\; e^{[\; \eta^T \;\Lambda\; T(y) \;-\; c(\eta,\Lambda)\;]}\, dy \;=$$

$$E\{\; [\Lambda\, T(y)]_j \; [\; [\Lambda\, T(y)]_k \;-\; \frac{\partial c}{\partial \eta_k}\;];\; \eta, \Lambda\;\} \;=$$

$$E\{\; [\Lambda\, T(y)]_j \; [\Lambda\, T(y)]_k;\; \eta, \Lambda\;\} \;-\; E\{\; [\Lambda\, T(y)]_j \; E[\; [\Lambda\, T(y)]_k;\; \eta, \Lambda\;]\;\} =$$

$$E\{\; [\Lambda\, T(y)]_j \; [\Lambda\, T(y)]_k;\; \eta, \Lambda\;\} \;-\; E\{\; [\Lambda\, T(y)]_j;\; \eta, \Lambda\;\} \;\times\; E\{\; [\Lambda\, T(y)]_k;\; \eta, \Lambda\;\}$$

We conclude that:

$$\boxed{\frac{\partial^2 c}{\partial \eta_j \partial \eta_k} \;=\; Cov[\; [\Lambda\, T(y)]_j \;,\; [\Lambda\, T(y)]_k;\; \eta, \Lambda\;]} \tag{5}$$

An important implication is that the Hessian of c with respect to η is a covariance matrix and is hence symmetric positive semi-definite. This demonstrates that c **is convex in** η. Furthermore, c is clearly linear in Λ since $c(\eta,\Lambda) \;=\; [q(\eta)]^T \;\Lambda\; q(\eta)$. **As a result,** c **is also convex in** Λ.

Element #2: A mean function $h : \mathbb{R}^p \times \mathbb{S}^p_{++} \to \mathbb{R}^p$ that maps (η, Λ) to the expected value of the sufficient statistic $T(y)$:

$$\boxed{h(\eta,\Lambda) \;=\; E[\; T(y);\; \eta, \Lambda\;]} \tag{6}$$

$$E[\; T(y);\; \eta, \Lambda\;] \;\equiv\; \begin{bmatrix} E[\; (T(y))_1;\; \eta, \Lambda\;] \\ ... \\ E[\; (T(y))_p;\; \eta, \Lambda\;] \end{bmatrix}$$

We can express the mean function h in terms of the log-partition function c, or equivalently in terms of the map q. To do so, we first define the derivative operator of the vector-valued map q with respect to η to be the quantity:

$$(\mathcal{D}_\eta\; q) \;=\; \begin{bmatrix} \frac{\partial q_1}{\partial \eta_1} & .. & \frac{\partial q_1}{\partial \eta_p} \\ .. & .. & .. \\ \frac{\partial q_p}{\partial \eta_1} & .. & \frac{\partial q_p}{\partial \eta_p} \end{bmatrix}$$

Equations (4) and (6) show that:

$$\boxed{h(\eta,\Lambda) = \;E[\; T(y);\; \eta, \Lambda\;] = (\Lambda)^{-1}\; (\bigtriangledown_\eta\; c)^T} \tag{7}$$

We could also invoke the chain rule and write:

$$(\bigtriangledown_\eta\ c) = (\bigtriangledown_q\ c)\ (\mathcal{D}_\eta\ q) = \bigtriangledown_q[\ [q(\eta)]^T\ \Lambda\ q(\eta)\ ]\ (\mathcal{D}_\eta\ q)\ =$$

$$[\ (\Lambda + \Lambda^T)\ q(\eta)\ ]^T\ (\mathcal{D}_\eta\ q) = 2\ [q(\eta)]^T\ \Lambda\ (\mathcal{D}_\eta\ q)\ \ \text{(since } \Lambda \text{ is symmetric)}$$

It follows that:

$$\boxed{(\bigtriangledown_\eta\ c)^T\ =\ 2\ (\mathcal{D}_\eta\ q)^T\ \Lambda\ q(\eta)} \tag{8}$$

And hence, that:

$$\boxed{h(\eta, \Lambda) =\ E[\ T(y);\ \eta, \Lambda\ ]\ =\ 2\ (\Lambda)^{-1}\ (\mathcal{D}_\eta\ q)^T\ \Lambda\ q(\eta)} \tag{9}$$

Element #3: A design criteria that imposes a linear relationship between the natural parameter vector η and the input x. Note that the linearity condition is a matter of choice and one could theoretically investigate more complex choices including e.g., quadratic or higher order relationships. The linearity condition is expressed as $\eta = \Theta x$, where:

- $\eta \in \mathbb{R}^p\ (p \geq 1)$
- $x \in \mathbb{R}^{n+1}\ (n \geq 1)$ is the **design vector** given by $[1\ x_1\ ...\ x_n]^T$. The $x_i\ (i = 1, ..., n)$ are the n features (i.e., components of the design vector) and the leading 1 accounts for the intercept term.
- Θ is the coefficient matrix $\in \mathbb{R}^{p\times(n+1)}$. Note that Θ reduces to a row vector whenever $p = 1$.

The design criteria establishes a link between input and output. In other terms, knowledge of $x,\ \Theta,\ \Lambda$ and the mean function h allow one to compute:

$$h(\eta, \Lambda) = h(\Theta x, \Lambda) = E[(T(y))\ |\ x;\ \Theta, \Lambda]$$

Subsequently, one can make informed predictions about the output given a certain input as we will later see in sections 6 and 7.

The special case of a dispersion parameter: In what follows, we consider the special case of a dispersion matrix Λ equal to a positive scalar multiple $\rho \in \mathbb{R}^+$ of the $(p \times p)$ identity matrix I_p. We write $\Lambda = \rho \times I_p$. This case includes many of the known probability distributions including the univariate Gaussian, Bernoulli, Poisson, Geometric and Multinomial cases that we will revisit in section 6. As expected, this particular case lends itself to further simplification of the mean and the log-partition functions, as well as the latter's Gradient and Hessian expressions:

- First, note that in this case, the form of the distribution reduces to:

$$\boxed{p(y;\ \eta, \rho) = b(y, \rho)\ e^{\rho\ [\ \eta^T\ T(y)\ -\ [q(\eta)]^T\ q(\eta)\ ]}} \tag{10}$$

By defining $a : \mathbb{R}^p \to \mathbb{R}$ to be the map taking η to $a(\eta) = [q(\eta)]^T\ q(\eta)$, we can rewrite the probability distribution as:

$$\boxed{p(y;\ \eta, \rho) = b(y, \rho)\ e^{\rho\ [\ \eta^T\ T(y)\ -\ a(\eta)\ ]}} \tag{11}$$

It becomes clear that the log-partition function can be expressed as:

$$\boxed{c(\eta, \rho) = \rho\ a(\eta)} \tag{12}$$

- The Gradient of the log-partition function can also be simplified and written as:

$$\boxed{(\bigtriangledown_\eta c)^T = \rho\ (\bigtriangledown_\eta a)^T} \tag{13}$$

Equation (13) coupled with equation (4) demonstrate that:

$$\boxed{h(\eta) = E[\ T(y);\ \eta\ ] = (\bigtriangledown_\eta\ a)^T} \tag{14}$$

An important observation is that in this case, **the mean function h does not depend on the dispersion parameter ρ. It is completely determined by the natural parameter η.**

- Lastly, note that equation (12) shows that:

$$\boxed{\frac{\partial^2 c}{\partial \eta_j \partial \eta_k}\ =\ \rho\ \frac{\partial^2 a}{\partial \eta_j \partial \eta_k}} \tag{15}$$

Coupled with equation (5), equation (12) allows us to conclude that:

$$\boxed{Cov[\ [T(y)]_j,\ [T(y)]_k;\ \eta, \rho\ ]\ =\ \frac{1}{\rho}\ \frac{\partial^2 a}{\partial \eta_j \partial \eta_k}} \tag{16}$$

An important implication is that the Hessian of a with respect to η is a positive multiple of a covariance matrix and is hence positive semi-definite. This shows that a **is convex in** η.

3 GLM's Cost Function, Gradient and Hessian

The long form basic Cost Function: In order to compute $h(\Theta x, \Lambda)$ and conduct predictions on a given input x, one first needs to decide on the dispersion matrix Λ and the matrix Θ of coefficients. The performance of the predictive model will be dictated by the choice of Θ and Λ. In our supervised learning setting, these two matrices will be jointly determined by:

- The pre-defined training set $\cup_{i=1}^{m} \{(x^{(i)}, y^{(i)})\}$ for a given integer $m \geq 1$.
- The choice of an objective function to optimize. We refer to it as the **Cost Function**, denote it by J and define it as a map that takes Θ and Λ as inputs and that outputs a real number. For the purpose of this note, we derive J by applying the principle of **Maximum Likelihood** which we describe next.

Define the likelihood function L associated with a training set $\cup_{i=1}^{m} \{(x^{(i)}, y^{(i)})\}$ to be

$$L : \mathbb{R}^{p\times(n+1)} \times \mathbb{R}^{p\times p} \rightarrow (0, 1]$$

$$(\Theta, \Lambda) \rightarrow L(\Theta, \Lambda) = p(y^{(1)}, ..., y^{(m)} \mid x^{(1)}, ..., x^{(m)};\ \Theta, \Lambda)$$

Our objective is to find the matrices Θ and Λ that maximize L. To proceed further, we assume that $\forall i \in \{1, ..., m\},\ y^{(i)}$ depends only on $x^{(i)}$. We get:

$$L(\Theta, \Lambda) \ =$$

$$p(y^{(1)} \mid y^{(2)}, ..., y^{(m)}, x^{(1)}, ..., x^{(m)};\ \Theta, \Lambda) \ \times \ p(y^{(2)}, ..., y^{(m)} \mid x^{(1)}, ..., x^{(m)};\ \Theta, \Lambda) \ =$$

$$p(y^{(1)} \mid x^{(1)};\ \Theta, \Lambda) \ \times \ p(y^{(2)}, ..., y^{(m)} \mid x^{(1)}, ..., x^{(m)};\ \Theta, \Lambda) \ =$$

$$\Pi_{i=1}^{m}\ p(y^{(i)} \mid x^{(i)};\ \Theta, \Lambda)$$

The presence of products coupled with the exponential nature of the conditional probability distribution makes it more appealing to invoke the natural logarithm function. Most importantly, the logarithm function is increasing on the subset of positive real numbers. This implies that maximizing the likelihood function L is equivalent to maximizing the log-likelihood function $l \ \equiv \ ln(L)$ over all possible choices of Θ and Λ. We write:

$$l(\Theta, \Lambda) = ln\ [\ \Pi_{i=1}^{m}\ p(y^{(i)} \mid x^{(i)};\ \Theta, \Lambda)\ ] \ = \ \Sigma_{i=1}^{m}\ ln\ [\ p(y^{(i)} \mid x^{(i)};\ \Theta, \Lambda)\ ] \ =$$

$$\Sigma_{i=1}^{m}\ ln\ [\ b(y^{(i)}, \Lambda)\ e^{[\ (x^{(i)})^T\ \Theta^T\ \Lambda\ T(y^{(i)})\ -\ c(\Theta x^{(i)}, \Lambda)\ ]}] \ =$$

$$\Sigma_{i=1}^{m}\ ln\ (\ b(y^{(i)}, \Lambda)\ ) \ + \ \Sigma_{i=1}^{m}\ [\ (x^{(i)})^T\ \Theta^T\ \Lambda\ T(y^{(i)}) \ - \ c(\Theta x^{(i)}, \Lambda)\ ]$$

Finally, note that maximizing l is equivalent to minimizing $(-\frac{l}{m})$. We thus define the **long form basic Cost Function** to be the function:

$$J^{(LB)} : \mathbb{R}^{p\times(n+1)} \times \mathbb{R}^{p\times p} \ \rightarrow \mathbb{R} \quad \text{that maps } (\Theta, \Lambda) \text{ to:}$$

$$\boxed{J^{(LB)}(\Theta, \Lambda) = \frac{1}{m} \Sigma_{i=1}^{m} \left[c(\Theta x^{(i)}, \Lambda) - (x^{(i)})^T \Theta^T \Lambda T(y^{(i)}) - ln \left(b(y^{(i)}, \Lambda) \right) \right]} \quad (17)$$

We use the descriptor *long form* to distinguish it from a shorter form associated with the special case of a dispersion matrix Λ equal to a scalar multiple of the identity matrix. On the other hand, the *basic* attribute will be contrasted with a more *general* counterpart that will incorporate weights and a regularization parameter as we will see shortly.

The optimal values Θ^* and Λ^* must satisfy:

$$(\Theta^*, \Lambda^*) = argmin_{(\Theta,\Lambda)} \ J^{(LB)}(\Theta, \Lambda)$$

The long form general Cost Function: We can generalize further the Cost Function by accounting for two additional factors:

1. At times, when conducting a prediction on an input $x \in \mathbb{R}^{n+1}$, one might want to give more **weight** to training set points that are in the vicinity of x. One common way of doing so is by invoking a particular weight function w defined as follows:

$$w : \mathbb{R}^{n+1} \times \mathbb{R}^{n+1} \rightarrow \mathbb{R}$$

$$(x, t) \ \rightarrow \ e^{-\frac{1}{2\tau^2} \ (t-x)^T \ (t-x)}$$

   Given an input x on which a prediction needs to be conducted, one can evaluate w at the different training points $x^{(i)}, i \in \{1, ..., m\}$. We let $w_x^{(i)}$ denote the quantity $e^{-\frac{1}{2\tau^2} \ (x^{(i)}-x)^T \ (x^{(i)}-x)}$ and refer to it as the weight attributed to the i^{th} training point associated with input x.

   Different values of x yield different weights attributed to the same training point. Moreover, larger values of the bandwidth paramter τ result in more inputs in the neighborhood of x being attributed higher weights. In the extreme case when $\tau \rightarrow \infty$, all training set inputs get a weight of 1.

2. It is common practice in convex optimization to introduce a regularization component to the objective function. The purpose of it is to penalize high values of the optimization variables. In our case, the optimization variables consist of:

   - The matrix of coefficients $\Theta = [\Theta_1 \ .. \ \Theta_p]^T$ where $\Theta_j = [\theta_{j1} \ ... \ \theta_{j(n+1)}]^T$ for $j \in \{1, ..., p\}$.
   - The dispersion matrix $\Lambda = [\Lambda_1 \ .. \ \Lambda_p]^T$ where $\Lambda_j = [\rho_{j1} \ ... \ \rho_{jp}]^T$ for $j \in \{1, ..., p\}$.

   The regularization term is usually proportional to the size of the variable, where size is measured according to some norm (e.g., L2 or L1). In our case, we add a

regularization term given by $\frac{\lambda}{2}\ \Sigma_{j=1}^{p}\ [\Theta_j^T\Theta_j\ +\Lambda_j^T\Lambda_j]$. The λ variable is the proportionality constant and the other factor is the sum of the squares of the L2 norm of each Θ_j and each $\Lambda_j,\ j \in \{1, ..., p\}$.

Given a specific x and λ, we denote the corresponding **long form general Cost Function** by $J_{x,\lambda}^{(LG)}$. The subscripts are meant to highlight the potential dependence on weights (as dictated by the input x) and on the regularization parameter λ. We have:

$$J_{x,\lambda}^{(LG)} : \mathbb{R}^{p\times(n+1)} \times \mathbb{R}^{p\times p} \to \mathbb{R}$$

$$(\Theta, \Lambda) \to$$

$$\boxed{\begin{array}{c} J_{x,\lambda}^{(LG)}\ (\Theta,\Lambda) = \frac{1}{m}\ \Sigma_{i=1}^{m}\ w_x^{(i)}\ \{\ c(\Theta x^{(i)}, \Lambda)\ - \\ (x^{(i)})^T\ \Theta^T\ \Lambda\ T(y^{(i)})\ -\ ln\ (b(y^{(i)}, \Lambda))\ \}\ +\ \frac{\lambda}{2}\ \Sigma_{j=1}^{p}\ [\ \Theta_j^T\Theta_j\ +\Lambda_j^T\Lambda_j\ ] \end{array}} \tag{18}$$

An important observation is that $J_{x,\lambda}^{(LG)}$ is convex in Θ. To see why, recall from section 2 that the map c is convex in η. Furthermore, $\eta = \Theta x^{(i)}$ is linear in Θ and so $c(\Theta x^{(i)})$ is convex in Θ. Moreover, $-(x^{(i)})^T\ \Theta^T\ T(y^i)$ is linear in Θ and thus convex. Finally, one can easily show that $\Sigma_{j=1}^{p}\ \Theta_j^T\ \Theta_j$ is also convex in Θ. Being a positively scaled sum of convex functions, $J_{x,\lambda}^{(LG)}$ is thus convex in Θ. This in turn implies the existence of a globally minimizing value Θ^*.

In addition, note that all the terms appearing in $J_{x,\lambda}^{(LG)}$ are convex in Λ (recall that we've seen in section 2 that the log-partition function c is convex in Λ), except possibly for the term $-ln(\ b(y^{(i)},\ \Lambda)\ )$. If $b(y^{(i)},\ \Lambda)$ is convex in Λ, then so will $-ln(\ b(y^{(i)},\ \Lambda)\ )$, and as a result, so will $J_{x,\lambda}^{(LG)}$. This would then imply the existence of a globally minimizing Λ^*.

The long form general Cost Function's Gradient: We define the Gradient of $J_{x,\lambda}^{(LG)}$ with respect to matrices Θ and Λ to be the map:

$$\bigtriangledown\ (J_{x,\lambda}^{(LG)}) : \mathbb{R}^{p\times(n+1)} \times \mathbb{R}^{p\times p} \to \mathbb{R}^{p\times(p+n+1)}$$

$$(\Theta, \Lambda)\ \to$$

$$\boxed{\bigtriangledown_{\Theta,\Lambda}\ (J_{x,\lambda}^{(LG)})\ =\ [\ \bigtriangledown_{\Theta}\ (J_{x,\lambda}^{(LG)})\quad \bigtriangledown_{\Lambda}\ (J_{x,\lambda}^{(LG)})\ ]} \tag{19}$$

where, $\bigtriangledown_{\Theta}\ (J_{x,\lambda}^{(LG)})$ and $\bigtriangledown_{\Lambda}\ (J_{x,\lambda}^{(LG)})$ are defined as follows:

$$\bigtriangledown_{\Theta}\ (J_{x,\lambda}^{(LG)}) = \begin{bmatrix} \bigtriangledown_{\Theta_1}\ (J_{x,\lambda}^{(LG)}) \\ ... \\ \bigtriangledown_{\Theta_p}\ (J_{x,\lambda}^{(LG)}) \end{bmatrix} = \begin{bmatrix} \frac{\partial J_{x,\lambda}^{(LG)}}{\partial\theta_{11}} & .. & \frac{\partial J_{x,\lambda}^{(LG)}}{\partial\theta_{1(n+1)}} \\ .. & .. & .. \\ \frac{\partial J_{x,\lambda}^{(LG)}}{\partial\theta_{p1}} & .. & \frac{\partial J_{x,\lambda}^{(LG)}}{\partial\theta_{p(n+1)}} \end{bmatrix}$$

and

$$\bigtriangledown_{\Lambda}\ (J^{(LG)}_{x,\lambda}) = \begin{bmatrix} \bigtriangledown_{\Lambda_1}\ (J^{(LG)}_{x,\lambda}) \\ ... \\ \bigtriangledown_{\Lambda_p}\ (J^{(LG)}_{x,\lambda}) \end{bmatrix} = \begin{bmatrix} \frac{\partial J^{(LG)}_{x,\lambda}}{\partial \rho_{11}} & .. & \frac{\partial J^{(LG)}_{x,\lambda}}{\partial \rho_{1p}} \\ .. & .. & .. \\ \frac{\partial J^{(LG)}_{x,\lambda}}{\partial \rho_{p1}} & .. & \frac{\partial J^{(LG)}_{x,\lambda}}{\partial \rho_{pp}} \end{bmatrix}$$

Note that $\forall i \in \{1, ..., p\},\ \bigtriangledown_{\Theta_i}\ (J^{(LG)}_{x,\lambda})$ and $\bigtriangledown_{\Lambda_i}\ (J^{(LG)}_{x,\lambda}),$ are taken to be one-forms (being covariant derivatives of the scalar function $J^{(LG)}_{x,\lambda}$ in the directions of vector Θ_i and Λ_i respectively). We subsequently represent them in Euclidean space as row vectors.

For any $1 \leq k \leq p$ and $1 \leq j \leq (n+1),$ The $(kj)^{th}$ component of $\bigtriangledown_{\Theta}(J^{(LG)}_{x,\lambda})$ is given by:

$$\frac{\partial J^{(LG)}_{x,\lambda}}{\partial \theta_{kj}} \ = \ \frac{1}{m}\ \Sigma^{m}_{i=1}\ w^{(i)}_{x}\ [\ \Sigma^{p}_{s=1}\ \frac{\partial c}{\partial \eta_s}(\Theta x^{(i)}, \Lambda)\ \frac{\partial \eta_s}{\partial \theta_{kj}} - x^{(i)}_{j}\ [\Lambda\ T(y^{(i)})]_k\ ]\ +\ \lambda \theta_{kj}$$

Recall that by design, we chose $\eta \equiv [\eta_1\ ..\ \eta_p]^T = \Theta x$, and so $\eta_s = \Theta^T_s x$. As a result, the only value of s for which η_s contains the term θ_{kj} is $s = k$. We simplify to obtain:

$$\frac{\partial J^{(LG)}_{x,\lambda}}{\partial \theta_{kj}} \ = \ \frac{1}{m}\ \Sigma^{m}_{i=1}\ w^{(i)}_{x}\ [\ \frac{\partial c}{\partial \eta_k}(\Theta x^{(i)}, \Lambda)\ \frac{\partial \eta_k}{\partial \theta_{kj}} - x^{(i)}_{j}\ [\Lambda\ T(y^{(i)})]_k\ ]\ +\ \lambda \theta_{kj}$$

And hence conclude that:

$$\boxed{\frac{\partial J^{(LG)}_{x,\lambda}}{\partial \theta_{kj}} \ = \ \frac{1}{m}\ \Sigma^{m}_{i=1}\ w^{(i)}_{x}\ [\ \frac{\partial c}{\partial \eta_k}(\Theta x^{(i)}, \Lambda)\ x^{(i)}_{j} - x^{(i)}_{j}\ [\Lambda\ T(y^{(i)})]_k\ ]\ +\ \lambda \theta_{kj}} \tag{20}$$

Similarly, one finds that for $1 \leq k, j \leq p,$ The $(kj)^{th}$ component of $\bigtriangledown_{\Lambda}(J^{(LG)}_{x,\lambda})$ is:

$$\boxed{\begin{gathered} \frac{\partial J^{(LG)}_{x,\lambda}}{\partial \rho_{kj}} = \frac{1}{m}\ \Sigma^{m}_{i=1}\ w^{(i)}_{x}\ \{\ \frac{\partial c}{\partial \rho_{kj}}(\Theta x^{(i)}, \Lambda)\ - \\ [\Theta x^{(i)}]_k\ [T(y^{(i)})]_j - \frac{1}{b(y^{(i)}, \Lambda)} \frac{\partial b}{\partial \rho_{kj}}(y^{(i)}, \Lambda)\ \}\ +\ \lambda \rho_{kj} \end{gathered}} \tag{21}$$

The long form general Cost Function's Hessian: Let α be the column vector $\in \mathbb{R}^{p(p+n+1)}$ whose components are given by:

$$[\theta_{11}\ ..\ \theta_{1(n+1)}\ ..\ \theta_{p1}\ ..\ \theta_{p(n+1)}\ \rho_{11}\ ..\ \rho_{1p}\ ..\ \rho_{p1}\ ..\ \rho_{pp}]^T$$

We define the Hessian of the Cost Function $J^{(LG)}_{x,\lambda}$ with respect to matrices Θ and Λ to be the map:

$$H(J^{(LG)}_{x,\lambda}) : \mathbb{R}^{p\times(n+1)} \times \mathbb{R}^{p\times p} \rightarrow \mathbb{R}^{p(p+n+1)\times p(p+n+1)}$$

$$(\Theta,\Lambda) \rightarrow H_{\Theta,\Lambda}\ (J^{(LG)}_{x,\lambda}) = \begin{bmatrix} \frac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\alpha_1\partial\alpha_1} & .. & \frac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\alpha_1\partial\alpha_{p(p+n+1)}} \\ .. & .. & .. \\ \frac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\alpha_{p(p+n+1)}\partial\alpha_1} & .. & \frac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\alpha_{p(p+n+1)}\partial\alpha_{p(p+n+1)}} \end{bmatrix}$$

We consider three cases:

- For $1 \leq u,k \leq p,\ \ 1 \leq v,j \leq (n+1),$ the $\frac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\theta_{uv}\partial\theta_{kj}}$ component of $H_{\Theta,\Lambda}\ (J^{(LG)}_{x,\lambda})$ is:

$$\frac{\partial}{\partial\theta_{uv}}\{\ \frac{1}{m}\ \Sigma_{i=1}^{m}\ w_x^{(i)}\ [\ \frac{\partial c}{\partial\eta_k}(\Theta x^{(i)},\Lambda)\ x_j^{(i)}\ -\ x_j^{(i)}\ [\Lambda\ T(y^{(i)})]_k\ ]\ +\ \lambda\theta_{kj}\ \} =$$

$$\frac{1}{m}\ \Sigma_{i=1}^{m}\ w_x^{(i)}\ x_j^{(i)}\ \frac{\partial}{\partial\theta_{uv}}\ [\ \frac{\partial c}{\partial\eta_k}(\Theta x^{(i)},\Lambda)\ ]\ +\ \lambda\delta_{uk}\delta_{vj}$$

  Let $f \equiv \frac{\partial c}{\partial\eta_k}$ and g the map that takes (η,Λ) to $g(\eta,\Lambda) = (\Theta x^{(i)},\Lambda)$. The chain rule allows us to write:

$$\frac{\partial}{\partial\theta_{uv}}(f\circ g) = \Sigma_{s=1}^{p}\ \frac{\partial f}{\partial\eta_s}(\Theta x^{(i)},\Lambda)\ \frac{\partial\eta_s}{\partial\theta_{uv}} = \Sigma_{s=1}^{p}\ \frac{\partial^2 c}{\partial\eta_s\partial\eta_k}(\Theta x^{(i)},\Lambda)\ \frac{\partial\eta_s}{\partial\theta_{uv}}$$

  Since $\eta_s = \Theta_s^T x$, we get $\frac{\partial\eta_s}{\partial\theta_{uv}} = 0$ whenever $s \neq u$. As a result:

$$\frac{\partial}{\partial\theta_{uv}}(f\circ g) = \frac{\partial^2 c}{\partial\eta_u\partial\eta_k}(\Theta x^{(i)},\Lambda)\ x_v^{(i)}$$

  We conclude that:

$$\boxed{\frac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\theta_{uv}\partial\theta_{kj}} = \frac{1}{m}\ \Sigma_{i=1}^{m}\ w_x^{(i)}\ x_j^{(i)}\ x_v^{(i)}\ \frac{\partial^2 c}{\partial\eta_u\partial\eta_k}(\Theta x^{(i)},\Lambda)\ +\ \lambda\delta_{uk}\delta_{vj}} \tag{22}$$

- For $1 \leq u,k,v,j \leq p,$ the $\frac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\rho_{uv}\partial\rho_{kj}}$ component of $H_{\Theta,\Lambda}\ (J^{(LG)}_{x,\lambda})$ is given by:

$$\frac{\partial}{\partial\rho_{uv}}\ \{\frac{1}{m}\ \Sigma_{i=1}^{m}\ w_x^{(i)}\ [\ \frac{\partial c}{\partial\rho_{kj}}(\Theta x^{(i)},\Lambda)\ -$$

$$[\Theta x^{(i)}]_k\ [T(y^{(i)})]_j - \frac{1}{b(y^{(i)},\Lambda)}\ \frac{\partial b}{\partial\rho_{kj}}(y^{(i)},\Lambda)\ ]\ +\ \lambda\rho_{kj}\}$$

  We get:

$$\boxed{\begin{array}{c}\dfrac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\rho_{uv}\partial\rho_{kj}} \;=\; \dfrac{1}{m}\,\Sigma_{i=1}^{m}\; w_x^{(i)}\;\{\;\dfrac{\partial^2 c}{\partial\rho_{uv}\partial\rho_{kj}}(\Theta x^{(i)},\Lambda)\;+\\ \dfrac{1}{[b(y^{(i)},\Lambda)]^2}\;[\;\dfrac{\partial b}{\partial\rho_{uv}}(y^{(i)},\Lambda)\;\;\dfrac{\partial b}{\partial\rho_{kj}}(y^{(i)},\Lambda)\;-\;b(y^{(i)},\Lambda)\;\dfrac{\partial^2 b}{\partial\rho_{uv}\partial\rho_{kj}}(y^{(i)},\Lambda)\;]\;\}\\ +\;\lambda\delta_{uk}\delta_{vj}\end{array}} \tag{23}$$

- For $1 \leq u,k,v \leq p,\;\; 1 \leq j \leq (n+1)$, the $\frac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\rho_{uv}\partial\theta_{kj}}$ and $\frac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\theta_{kj}\partial\rho_{uv}}$ components of $H_{\Theta,\Lambda}\;(J^{(LG)}_{x,\lambda})$ are equal and are given by:

$$\tfrac{\partial}{\partial\rho_{uv}}\{\;\tfrac{1}{m}\;\Sigma_{i=1}^{m}\;w_x^{(i)}\;[\;\tfrac{\partial c}{\partial\eta_k}(\Theta x^{(i)},\Lambda)\;x_j^{(i)}\;-\;x_j^{(i)}[\Lambda\;T(y^{(i)})]_k\;]\;+\;\lambda\theta_{kj}\;\}$$

  We get:

$$\boxed{\begin{array}{c}\dfrac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\rho_{uv}\partial\theta_{kj}} \;=\; \dfrac{\partial^2 J^{(LG)}_{x,\lambda}}{\partial\theta_{kj}\partial\rho_{uv}} \;=\\ \dfrac{1}{m}\;\Sigma_{i=1}^{m}\;w_x^{(i)}\;[\;\dfrac{\partial^2 c}{\partial\rho_{uv}\partial\eta_k}(\Theta x^{(i)},\Lambda)\;x_j^{(i)}\;-\;x_j^{(i)}[T(y^{(i)})]_v\;\delta_{uk}\;]\end{array}} \tag{24}$$

The short form basic Cost Function: With the exception of the multivariate Gaussian distribution, all the other probability distributions that we consider in this note have a dispersion matrix that is a positive scalar multiple of the identity matrix. For this particular case, equations (12) and (14) in section 2 showed that:

- The log partition function is given by $c(\eta,\rho) = \rho\;a(\eta)$, where ρ is now a positive scalar and η is the natural parameter vector.
- The mean function h is independent of the dispersion parameter ρ and depends solely on the natural parameter η. We write $h(\eta) = h(\Theta x)$ for a given coefficient matrix Θ and input vector x.

In what follows, we simplify the expression of the Cost Function associated with such a case and derive simpler formulae for the components of its Gradient and its Hessian. By substituting the matrix Λ with ρI_p (where $\rho \in \mathbb{R}^+$ and $p \geq 1$) in equation (17), and by applying equation (12), we can write:

$$J^{(LB)}(\Theta,\rho) \;=\; \tfrac{1}{m}\;\Sigma_{i=1}^{m}\;[\;\rho\;a(\Theta x^{(i)})\;-\;\rho\;(x^{(i)})^T\;\Theta^T\;T(y^{(i)})\;-\;ln\;(\;b(y^{(i)},\rho)\;)\;]$$

Minimizing $J^{(LB)}$ with respect to Θ is equivalent to minimizing the following quantity with respect to Θ:

$$\tfrac{1}{m}\;\Sigma_{i=1}^{m}\;[\;\rho\;a(\Theta x^{(i)})\;-\;\rho\;(x^{(i)})^T\;\Theta^T\;T(y^{(i)})\;]$$

And since ρ is a positive scalar, the optimal value of Θ that minimizes this quantity is the same as the one that minimizes the following quantity:

$$\boxed{J^{(SB)}(\Theta) \;=\; \frac{1}{m}\,\Sigma_{i=1}^{m}\,[\;a(\Theta x^{(i)}) \;-\; (x^{(i)})^T\;\Theta^T\;T(y^{(i)})\;]} \tag{25}$$

We refer to $J^{(SB)}$ as the **short form basic Cost Function**. An important observation is that $J^{(SB)}$ depends exclusively on Θ. And since in this particular case the mean function h also depends only on Θ, one can find the optimal coefficient matrix by minimizing $J^{(SB)}$ and then plugging it into h to conduct predictions.

Equally important is the fact that $J^{(SB)}$ is convex in Θ. The proof is similar to the one we previously used to establish the convexity of $J_{x,\lambda}^{(LG)}$ in Θ.

The short form general Cost Function: The same approach used to derive the long form general Cost Function from its basic counterpart can be applied to obtain the following short form general Cost Function:

Given a specific x and λ, the corresponding **short form general Cost Function** is the map $J_{x,\lambda}^{(SG)} : \mathbb{R}^{p\times(n+1)} \;\rightarrow\; \mathbb{R}$ that takes Θ to:

$$\boxed{J_{x,\lambda}^{(SG)}(\Theta) \;=\; \frac{1}{m}\,\Sigma_{i=1}^{m}\,w_x^{(i)}[\;a(\Theta x^{(i)}) \;-\; (x^{(i)})^T\;\Theta^T\;T(y^{(i)})\;] \;+\; \frac{\lambda}{2}\,\Sigma_{j=1}^{p}\,\Theta_j^T\Theta_j} \tag{26}$$

The short form general Cost Function's Gradient: We define the Gradient of $J_{x,\lambda}^{(SG)}$ with respect to the matrix Θ to be the map:

$$\bigtriangledown\,(J_{x,\lambda}^{(SG)}) : \mathbb{R}^{p\times(n+1)} \;\rightarrow\; \mathbb{R}^{p\times(n+1)}$$

$$\Theta \;\rightarrow\; \bigtriangledown_\Theta\,(J_{x,\lambda}^{(SG)}) = \begin{bmatrix} \bigtriangledown_{\Theta_1}\,(J_{x,\lambda}^{(SG)}) \\ \cdots \\ \bigtriangledown_{\Theta_p}\,(J_{x,\lambda}^{(SG)}) \end{bmatrix} = \begin{bmatrix} \frac{\partial J_{x,\lambda}^{(SG)}}{\partial\theta_{11}} & .. & \frac{\partial J_{x,\lambda}^{(SG)}}{\partial\theta_{1(n+1)}} \\ .. & .. & .. \\ \frac{\partial J_{x,\lambda}^{(SG)}}{\partial\theta_{p1}} & .. & \frac{\partial J_{x,\lambda}^{(SG)}}{\partial\theta_{p(n+1)}} \end{bmatrix}$$

For $1 \leq k \leq p$ and $1 \leq j \leq (n+1)$, The $(kj)^{th}$ component of $\bigtriangledown_\Theta\,(J_{x,\lambda}^{(SG)})$ is given by:

$$\frac{\partial J_{x,\lambda}^{(SG)}}{\partial\theta_{kj}} \;=\; \tfrac{1}{m}\,\Sigma_{i=1}^{m}\,w_x^{(i)}\;[\;\Sigma_{s=1}^{p}\,\tfrac{\partial a}{\partial\eta_s}(\Theta x^{(i)})\;\tfrac{\partial\eta_s}{\partial\theta_{kj}} \;-\, x_j^{(i)}\;[T(y^{(i)})]_k\;] \;+\; \lambda\theta_{kj}$$

Recall that by design, we chose $\eta \equiv [\eta_1\;..\;\eta_p]^T = \Theta x$, and so $\eta_s = \Theta_s^T x$. As a result, the only value of s for which η_s contains the term θ_{kj} is $s = k$. This allows us to write

$$\frac{\partial J_{x,\lambda}^{(SG)}}{\partial\theta_{kj}} \;=\; \tfrac{1}{m}\,\Sigma_{i=1}^{m}\,w_x^{(i)}\;[\;\tfrac{\partial a}{\partial\eta_k}(\Theta x^{(i)})\;\tfrac{\partial\eta_k}{\partial\theta_{kj}} \;-\; x_j^{(i)}\;[T(y^{(i)})]_k\;] \;+\; \lambda\theta_{kj}$$

And hence conclude that:

$$\boxed{\frac{\partial J^{(SG)}_{x,\lambda}}{\partial \theta_{kj}} \;=\; \frac{1}{m}\,\Sigma^{m}_{i=1}\; w^{(i)}_{x}\;\left[\;\frac{\partial a}{\partial \eta_k}(\Theta x^{(i)})\;x^{(i)}_{j}\;-\;x^{(i)}_{j}\;[T(y^{(i)})]_k\;\right]\;+\;\lambda\theta_{kj}} \tag{27}$$

The short form general Cost Function's Hessian: We define the Hessian of $J^{(SG)}_{x,\lambda}$ with respect to the matrix Θ to be the map:

$$H(J^{(SG)}_{x,\lambda}) : \mathbb{R}^{p\times(n+1)} \;\rightarrow\; \mathbb{R}^{p(n+1)\times p(n+1)}$$

$$\Theta \;\rightarrow\; H_{\Theta}\;(J^{(SG)}_{x,\lambda}) = \begin{bmatrix} (H_{11})_{\Theta} & .. & (H_{1p})_{\Theta} \\ .. & .. & .. \\ (H_{p1})_{\Theta} & .. & (H_{pp})_{\Theta} \end{bmatrix} (J^{(SG)}_{x,\lambda})$$

where $\forall u,k \in \{1,...,p\}$, the block matrix $(H_{uk})_{\Theta}\;(J^{(SG)}_{x,\lambda}) \in \mathbb{R}^{(n+1)\times(n+1)}$ is given by

$$(H_{uk})_{\Theta}\;(J^{(SG)}_{x,\lambda}) = \begin{bmatrix} \frac{\partial^2 J^{(SG)}_{x,\lambda}}{\partial\theta_{u1}\partial\theta_{k1}} & .. & \frac{\partial^2 J^{(SG)}_{x,\lambda}}{\partial\theta_{u1}\partial\theta_{k(n+1)}} \\ .. & .. & .. \\ \frac{\partial^2 J^{(SG)}_{x,\lambda}}{\partial\theta_{u(n+1)}\partial\theta_{k1}} & .. & \frac{\partial^2 J^{(SG)}_{x,\lambda}}{\partial\theta_{u(n+1)}\partial\theta_{k(n+1)}} \end{bmatrix}$$

For $1 \leq u,k \leq p$ and $1 \leq v,j \leq (n+1)$, the components of $H_{\Theta}\;(J^{(SG)}_{x,\lambda})$ are given by

$$\frac{\partial^2 J^{(SG)}_{x,\lambda}}{\partial\theta_{uv}\partial\theta_{kj}} \;=\; \frac{\partial}{\partial\theta_{uv}}\{\;\frac{1}{m}\;\Sigma^{m}_{i=1}\;w^{(i)}_{x}\;[\;\frac{\partial a}{\partial\eta_k}(\Theta x^{(i)})\;x^{(i)}_{j}\;-\;x^{(i)}_{j}\;[T(y^{(i)})]_k\;]\;+\;\lambda\theta_{kj}\;\}\;=$$

$$\frac{1}{m}\;\Sigma^{m}_{i=1}\;w^{(i)}_{x}\;x^{(i)}_{j}\;\frac{\partial}{\partial\theta_{uv}}\;[\;\frac{\partial a}{\partial\eta_k}(\Theta x^{(i)})\;]\;+\;\lambda\delta_{uk}\delta_{vj}$$

Let $f \equiv \frac{\partial a}{\partial\eta_k}$ and g be the function mapping η to $g(\eta) = \Theta x^{(i)}$. The chain rule allows us to write:

$$\frac{\partial}{\partial\theta_{uv}}(f\circ g) \;=\; \Sigma^{p}_{s=1}\;\frac{\partial f}{\partial\eta_s}(\Theta x^{(i)})\;\frac{\partial\eta_s}{\partial\theta_{uv}} \;=\; \Sigma^{p}_{s=1}\;\frac{\partial^2 a}{\partial\eta_s\partial\eta_k}(\Theta x^{(i)})\;\frac{\partial\eta_s}{\partial\theta_{uv}}$$

Since $\eta_s = \Theta^{T}_{s}x$, we conclude that:

$$\boxed{\frac{\partial^2 J^{(SG)}_{x,\lambda}}{\partial\theta_{uv}\partial\theta_{kj}} \;=\; \frac{1}{m}\;\Sigma^{m}_{i=1}\;w^{(i)}_{x}\;x^{(i)}_{j}\;x^{(i)}_{v}\;\frac{\partial^2 a}{\partial\eta_u\partial\eta_k}(\Theta x^{(i)})\;+\;\lambda\delta_{uk}\delta_{vj}} \tag{28}$$

4 Matricial formulation of the Cost Function, its Gradient, and its Hessian

In this section, we limit ourselves to the short form general Cost Function $J_{x,\lambda}^{(SG)}$. Since there is no room for confusion, we will drop the superscript (SG), refer to it simply as the Cost Function and denote it by $J_{(x,\lambda)}$. In order to derive a concise notation for $J_{x,\lambda}$, its Gradient $\bigtriangledown(J_{x,\lambda})$, and its Hessian $H(J_{x,\lambda})$, we first introduce relevant vectorial and matricial quantities. In what follows, we recall that p denotes the dimension of the sufficient statistic $T(y)$, r denotes that of y, n denotes the number of input features and m denotes the number of training examples.

- The **coefficient matrix** $\Theta \in \mathbb{R}^{p\times(n+1)}$ is given by:

$$\Theta = \begin{bmatrix} \Theta_1^T \\ ... \\ \Theta_p^T \end{bmatrix} = \begin{bmatrix} \theta_{11} & .. & \theta_{1(n+1)} \\ .. & .. & .. \\ \theta_{p1} & .. & \theta_{p(n+1)} \end{bmatrix}$$

- The **design matrix** $X \in \mathbb{R}^{m\times(n+1)}$ is given by:

$$X = \begin{bmatrix} x^{(1)T} \\ ... \\ x^{(m)T} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & .. & x_n^{(1)} \\ 1 & .. & .. & .. \\ 1 & x_1^{(m)} & .. & x_n^{(m)} \end{bmatrix}$$

- The **target matrix** $Y \in \mathbb{R}^{m\times r}$ for some $r \geq p$ is given by:

$$Y = \begin{bmatrix} y^{(1)T} \\ ... \\ y^{(m)T} \end{bmatrix} = \begin{bmatrix} y_1^{(1)} & .. & y_r^{(1)} \\ .. & .. & .. \\ y_1^{(m)} & .. & y_r^{(m)} \end{bmatrix}$$

- The **sufficient statistic matrix** $T \in \mathbb{R}^{m\times p}$ is given by:

$$T = \begin{bmatrix} T(y^{(1)})^T \\ ... \\ T(y^{(m)})^T \end{bmatrix} = \begin{bmatrix} t_1^{(1)} & .. & t_p^{(1)} \\ .. & .. & .. \\ t_1^{(m)} & .. & t_p^{(m)} \end{bmatrix}$$

- The **weight matrix** $W_x \in \mathbb{R}^{m\times m}$ associated with input x and weight function $w_x : \mathbb{R}^{n+1} \to \mathbb{R}$ is given by:

$$W_x = \begin{bmatrix} w_x^{(1)} & 0 & 0 & .. & 0 \\ 0 & w_x^{(2)} & 0 & .. & 0 \\ .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & w_x^{(m)} \end{bmatrix}$$

- The **regularization matrix** $L_\lambda \in \mathbb{R}^{(n+1)\times(n+1)}$ associated with parameter $\lambda \in \mathbb{R}$ is given by:

$$L_\lambda = \begin{bmatrix} \lambda & 0 & 0 & .. & 0 \\ 0 & \lambda & 0 & .. & 0 \\ .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & \lambda \end{bmatrix}$$

- The **log-partition vector** $A_\Theta \in \mathbb{R}^m$ associated with the log-partition function $a : \mathbb{R}^p \to \mathbb{R}$ that maps $\eta \equiv [\eta_1 \ ... \ \eta_p]^T$ to $a(\eta) = [q(\eta)]^T \ q(\eta)$ is given by:

$$A_\Theta = \begin{bmatrix} a(\Theta x^{(1)}) \\ ... \\ a(\Theta x^{(m)}) \end{bmatrix}$$

- The **log-partition Gradient matrix** $D_\Theta \in \mathbb{R}^{p\times m}$ is given by:

$$D_\Theta = \begin{bmatrix} \frac{\partial a}{\partial \eta_1}(\Theta x^{(1)}) & .. & \frac{\partial a}{\partial \eta_1}(\Theta x^{(m)}) \\ .. & .. & .. \\ \frac{\partial a}{\partial \eta_p}(\Theta x^{(1)}) & .. & \frac{\partial a}{\partial \eta_p}(\Theta x^{(m)}) \end{bmatrix}$$

- The **second order diagonal matrix** $(S_{uk})_\Theta \in \mathbb{R}^{m\times m}$ where $u, k \in \{1, ..., p\}$ is given by:

$$(S_{uk})_\Theta = \begin{bmatrix} \frac{\partial^2 a}{\partial \eta_u \partial \eta_k}(\Theta x^{(1)}) & 0 & 0 & .. & 0 \\ 0 & \frac{\partial^2 a}{\partial \eta_u \partial \eta_k}(\Theta x^{(2)}) & 0 & .. & 0 \\ .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & \frac{\partial^2 a}{\partial \eta_u \partial \eta_k}(\Theta x^{(m)}) \end{bmatrix}$$

- The **unit vector** $\mathbb{1}_m \in \mathbb{R}^m$ is given by:

$$\mathbb{1}_m = \begin{bmatrix} 1 \\ ... \\ 1 \end{bmatrix}$$

The Cost Function in matrix form: Suppose $U \in \mathbb{R}^{a\times b},\ M \in \mathbb{R}^{b\times c},\ V \in \mathbb{R}^{c\times d}$, and let $u^{(i)T}$ denote the i^{th} row of U and $v^{(j)}$ the j^{th} column of V. If $P \equiv UMV \in \mathbb{R}^{a\times d}$, one can easily see that $P_{ij} = u^{(i)T} M v^{(j)}$.

Substituting U, M and V with matrices $X,\ \Theta^T$ and T^T respectively, one concludes that the diagonal elements of $X\ \Theta^T\ T^T$ are given by $x^{(i)T}\ \Theta^T\ T(y^{(i)}),\ i \in \{1, ..., m\}$. Furthermore, multiplying this matricial product by the diagonal weight matrix W_x, one can see that the diagonal elements of the product $X\ \Theta^T\ T^T\ W_x$ are given by $w_x^{(i)}\ x^{(i)T}\ \Theta^T\ T(y^{(i)}),\ i \in \{1, ..., m\}$.

As a result, we can rewrite equation (26) more concisely in matrix form as:

$$\boxed{J_{x,\lambda}(\Theta) \ = \ \frac{1}{m}\ [\ \mathbb{1}_m^T\ W_x\ A_\Theta \ - \ Tr(X\ \Theta^T\ T^T\ W_x)\ ]\ +\ \frac{\lambda}{2}\ Tr(\ \Theta\ \Theta^T\ )} \qquad (29)$$

The Gradient of the Cost Function in matrix form: Using the same observation made in the previous paragraph regarding the product of three matrices, one can rewrite equation (27) in a more concise matrix notation as follows:

$$\boxed{\bigtriangledown_{\Theta}\ (J_{x,\lambda})\ =\ \frac{1}{m}\ [\ (D_{\Theta}\ -\ T^{T})\ W_{x}\ X\ ]\ +\ \Theta\ L_{\lambda}} \tag{30}$$

The Hessian of the Cost Function in matrix form: Similary, we can rewrite equation (28) in matrix notation as follows:

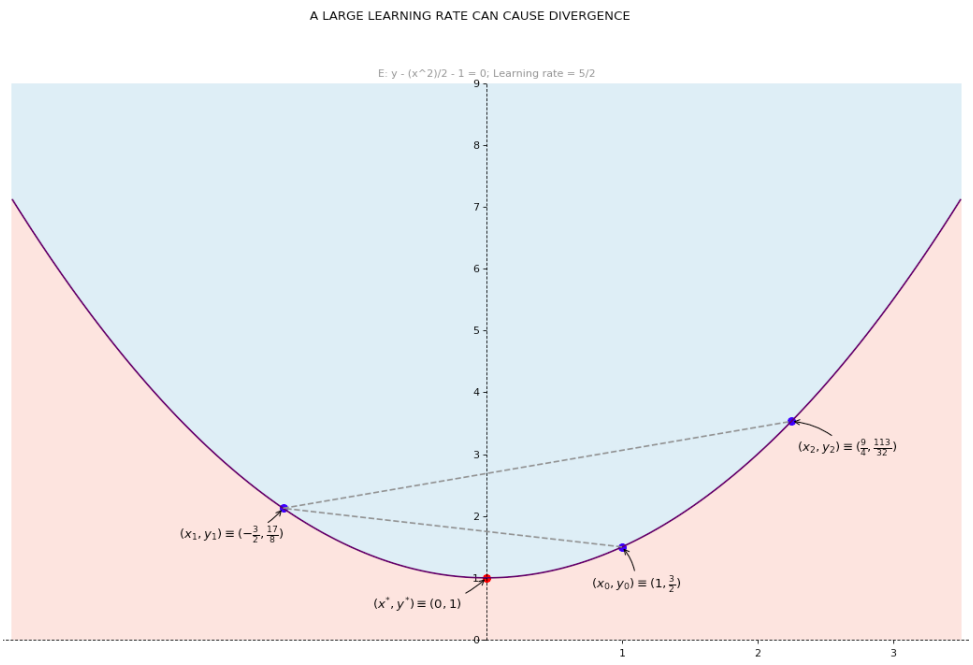
$$\boxed{\begin{array}{c} \forall u,k \in \{1,...,p\}, \\ (H_{uk})_{\Theta}\ (J_{x,\lambda})\ =\ \frac{1}{m}\ X^{T}\ (S_{uk})_{\Theta}\ W_{x}\ X \quad ,if\ u \neq k \\ (H_{uk})_{\Theta}\ (J_{x,\lambda})\ =\ \frac{1}{m}\ X^{T}\ (S_{uk})_{\Theta}\ W_{x}\ X\ +\ L_{\lambda} \quad ,if\ u = k \end{array}} \tag{31}$$

5 Algorithms to minimize the convex Cost Function

In this section too, we limit ourselves to the short form general Cost Function $J_{x,\lambda}$. Our objective is to find the optimal $\Theta^{*} = argmin_{\Theta}\ J_{x,\lambda}(\Theta)$. If the weight functions $w_{x}^{(i)}$ are independent of x for all training examples $i = 1, ..., m$, then Θ^{*} will not depend on x and once it is computed for a given input, it can be stored and used for all other inputs. This is known as a parametric setting. If the weight functions depend on x, then each x will have a different Θ_{x}^{*} associated with it. This procedure is known as non-parametric and is clearly computationally more demanding than its parametric counterpart. In what follows, we introduce three numerical methods for finding Θ^{*} :

1. **Batch Gradient Descent (BGD):** Suppose $f : \mathbb{R} \rightarrow \mathbb{R}$ is a differentiable convex function of one variable. Starting at any $x_0 \in \mathbb{R}$, one can get $f'(x_0) \equiv \frac{df}{dx}(x_0)$. If its negative (positive), then at x_0 the function is decreasing (increasing). As a result, to get closer to the value of x^{*} that minimizes f, one can try a value $x_1 > x_0$ $(x_1 < x_0)$.

   One way of updating the value of x_0 is by letting $x_1 \leftarrow x_0 - \alpha f'(x_0)$, for some positive learning rate $\alpha \in \mathbb{R}^{+}$. This procedure can be repeated until convergence. Note however, that the choice of α is critical since too large a value will cause divergence, while a value that is too small will cause slow convergence.



The same can be said of differentiable functions of p variables ($p > 1$) where this logic gets applied to each variable. In this context, f' is replaced by the Gradient of f and the learning rate α by the learning rate matrix $R \in \mathbb{R}^{p \times p}$ defined as:

$$R = \begin{bmatrix} \alpha & 0 & 0 & .. & 0 \\ 0 & \alpha & 0 & .. & 0 \\ .. & .. & .. & .. & .. \\ 0 & 0 & 0 & .. & \alpha \end{bmatrix}$$

One can estimate the optimal matrix Θ^* by running the following algorithm known as Batch Gradient Descent (BGD). We let *iter* denote an iteration count variable and *max-iter* denote the maximum number of allowed iterations before testing for convergence:

$\Theta^{(curr)} \leftarrow Initialize()$
(It could be initialized to e.g., the zero p by $(n+1)$ matrix).

For $(iter < max\text{-}iter)$
{
$\Theta^{(new)} = \Theta^{(curr)} - R \ \bigtriangledown_{\Theta^{(curr)}} (J_{x,\lambda})$

$\Theta^{(curr)} \leftarrow \Theta^{(new)}$

Update all the Θ-dependent quantities including
i.) A_Θ
ii.) D_Θ
iii.) $J_{x,\lambda}(\Theta)$
iv.) $\bigtriangledown_\Theta \ (J_{x,\lambda})$
}

So $\forall k \in \{1,..,p\},\ j \in \{1,..,(n+1)\}$, the following update is performed:

$$(\theta_{kj})^{(new)} \leftarrow (\theta_{kj})^{(curr)} - \alpha\ \frac{\partial J_{x,\lambda}}{\partial (\theta_{kj})^{(curr)}},$$

where we have:

$$\frac{\partial J_{x,\lambda}}{\partial \theta_{kj}} \ = \ \frac{1}{m}\ \Sigma_{i=1}^{m}\ w_x^{(i)}\ [\ \frac{\partial a}{\partial \eta_k}(\Theta x^{(i)})\ x_j^{(i)}\ -\ x_j^{(i)}\ t_k^{(i)}\ ]\ +\ \lambda\theta_{kj} \quad \text{and} \quad t_k^{(i)} \equiv [T(y^{(i)})]_k$$

Note that BGD requires that all the entries of the new coefficient matrix $\Theta^{(new)}$ be simultaneously updated before they can be used in the next iteration.

2. **Stochastic Gradient Descent (SGD):** In BGD, everytime we updated the value of θ_{kj} we had to perform a summation over all training examples as mandated by $\frac{\partial J_{x,\lambda}}{\partial \theta_{kj}}$. In SGD, we drop the summation in order to achieve faster updates of the coefficients. In other terms, for each training example $1 \leq i \leq m$, SGD conducts the following update round $\forall k \in \{1,..,p\},\ j \in \{1,..,(n+1)\}$:

$$(\theta_{kj})^{(new)} \leftarrow (\theta_{kj})^{(curr)}\ -\ \frac{\alpha}{m}\ w_x^{(i)}\ [\ \frac{\partial a}{\partial \eta_k}(\Theta^{(curr)}\ x^{(i)})\ x_j^{(i)}\ -\ x_j^{(i)}\ t_k^{(i)}\ ]\ +\ \lambda(\theta_{kj})^{(curr)}$$

The entries of the new coefficient matrix $\Theta^{(new)}$ must all get simultaneously updated for each training example in each iteration before they can be used in the next instance. SGD usually achieves acceptable convergence faster than BGD, especially when the size m of the training set is very large

In order to describe the SGD algorithm in a more concise matrix form, we define a set of m new matrices $Stoch_{\Theta}^{(i)}(J_{x,\lambda})\ \in \mathbb{R}^{p\times(n+1)},\ 1 \leq i \leq m$ given by:

$$\begin{bmatrix} \frac{w_x^{(i)}}{m}\ [\frac{\partial a}{\partial \eta_1}(\Theta x^{(i)})x_1^{(i)} - x_1^{(i)}t_1^{(i)}] + \lambda\theta_{11} & .. & \frac{w_x^{(i)}}{m}\ [\frac{\partial a}{\partial \eta_1}(\Theta x^{(i)})x_{(n+1)}^{(i)} - x_{(n+1)}^{(i)}t_1^{(i)}] + \lambda\theta_{1(n+1)} \\ .. & .. & .. \\ .. & .. & .. \\ \frac{w_x^{(i)}}{m}\ [\frac{\partial a}{\partial \eta_p}(\Theta x^{(i)})x_1^{(i)} - x_1^{(i)}t_p^{(i)}] + \lambda\theta_{p1} & .. & \frac{w_x^{(i)}}{m}\ [\frac{\partial a}{\partial \eta_p}(\Theta x^{(i)})x_{(n+1)}^{(i)} - x_{(n+1)}^{(i)}t_p^{(i)}] + \lambda\theta_{p(n+1)} \end{bmatrix}$$

SGD is a variant of BGD that runs the following algorithm:

$\Theta^{(curr)}\ \leftarrow\ Initialize()$
(It could be initialized to e.g., the zero p by $(n+1)$ matrix).

For $(iter < max\text{-}iter)$
{
  For $(i = 1, ..., m)$
  {
    $\Theta^{(new)} = \Theta^{(curr)} - R\ Stoch_{\Theta^{(curr)}}^{(i)}(J_{x,\lambda})$

    $\Theta^{(curr)} \leftarrow \Theta^{(new)}$

Update all the Θ-dependent quantities including
i.) A_Θ
ii.) D_Θ
iv.) $J_{x,\lambda}(\Theta)$
v.) $Stoch_\Theta^{(i \pmod m) + 1)}(J_{x,\lambda})$
}
}

3. **Newton-Raphson (NR)**: The choice of the learning rate α used in BGD and SGD is of special importance since it can either cause divergence or help achieve faster convergence. One limiting factor in the implementation of the previous Gradient descent algorithms is the fixed nature of α. Computational complexity aside, it would be beneficial if at every iteration, the algorithm could dynamically choose an appropriate α so as to reduce the number of steps needed to ahieve convergence. Enters the Newton Raphson (NR) method.

   To motivate it, we lean on a Taylor series expansion of the Cost Function $J_{x,\lambda}$ about a point (i.e., coefficient matrix) $\Theta^{(curr)}$ up to first order. We get:

$$J_{x,\lambda}(\Theta) \approx J_{x,\lambda}(\Theta^{(curr)}) \ + \ \Sigma_{u=1}^{p} \ \Sigma_{k=1}^{n+1} \ \frac{\partial J_{x,\lambda}}{\partial \theta_{uk}}(\Theta^{(curr)}) \ \ (\theta_{uk} - \theta_{uk}^{(curr)})$$

   If we want to get to the optimal Θ in one step, we need to set $\frac{\partial J_{x,\lambda}}{\partial \theta_{ij}}$ to 0 when evaluated at Θ, $\forall i \in \{1, ..., p\}$ and $j \in \{1, ..., (n+1)\}$. Taking the first derivative with respect to θ_{ij} of the right and left hand sides of the approximation, and noting that $J_{x,\lambda}(\Theta^{(curr)})$ is a constant and $\forall \ 1 \leq u, k \leq p, \ \theta_{uk}^{(curr)}$ are constants, we get:

$$\frac{\partial J_{x,\lambda}}{\partial \theta_{ij}}(\Theta) \approx 0 \ + \ \Sigma_{u=1}^{p} \ \Sigma_{k=1}^{n+1} \ \{ \ \frac{\partial^2 J_{x,\lambda}}{\partial \theta_{ij} \partial \theta_{uk}}(\Theta^{(curr)}) \ \ (\theta_{uk} - \theta_{uk}^{(curr)}) \ \} \ + \ \frac{\partial J_{x,\lambda}}{\partial \theta_{ij}}(\Theta^{(curr)})$$

   Setting $\frac{\partial J_{x,\lambda}}{\partial \theta_{ij}}(\Theta)$ to 0 for all i $\in \{1, .., p\}, \ j \in \{1, .., (n+1)\}$, we get:

$$\Sigma_{u=1}^{p} \ \Sigma_{k=1}^{n+1} \ \{ \ \frac{\partial^2 J_{x,\lambda}}{\partial \theta_{ij} \partial \theta_{uk}}(\Theta^{(curr)}) \ \ (\theta_{uk} - \theta_{uk}^{(curr)}) \ \} \ + \ \frac{\partial J_{x,\lambda}}{\partial \theta_{ij}}(\Theta^{(curr)}) = 0$$

   Recall that Θ and $\bigtriangledown_\Theta \ (J_{x,\lambda})$ are both elements of $\mathbb{R}^{p \times (n+1)}$. We define the vectorized versions of Θ and $\bigtriangledown_\Theta \ (J_{x,\lambda})$ to be the elements of $\mathbb{R}^{p(n+1)}$ given by:

$$vect(\Theta) \equiv \begin{bmatrix} \Theta_1 \\ .. \\ .. \\ \Theta_p \end{bmatrix} = \begin{bmatrix} \theta_{11} \\ .. \\ \theta_{1(n+1)} \\ .. \\ .. \\ \theta_{p1} \\ .. \\ \theta_{p(n+1)} \end{bmatrix} \qquad vect(\bigtriangledown_\Theta \ (J_{x,\lambda})) \equiv \begin{bmatrix} [\bigtriangledown_{\Theta_1} \ (J_{x,\lambda})]^T \\ .. \\ .. \\ [\bigtriangledown_{\Theta_p} \ (J_{x,\lambda})]^T \end{bmatrix} = \begin{bmatrix} \frac{\partial J_{x,\lambda}}{\partial \theta_{11}} \\ .. \\ \frac{\partial J_{x,\lambda}}{\partial \theta_{1(n+1)}} \\ .. \\ .. \\ \frac{\partial J_{x,\lambda}}{\partial \theta_{p1}} \\ .. \\ \frac{\partial J_{x,\lambda}}{\partial \theta_{p(n+1)}} \end{bmatrix}$$

We can subsequently write the above conditions in a more concise matrix form as follows:

$$H_{\Theta^{(curr)}}(J_{x,\lambda})\ vect(\Theta - \Theta^{(curr)})\ +\ vect(\bigtriangledown_{\Theta^{(curr)}}\ (J_{x,\lambda})) = 0$$

Which gives the following NR algorithm:

$\Theta^{(curr)} \leftarrow Initialize()$
(It could be initialized to e.g., the zero p by $(n+1)$ matrix).

For $(iter < max\text{-}iter)$
{
$vect(\Theta^{(new)}) \leftarrow vect(\Theta^{(curr)}) - [H_{\Theta^{(curr)}}\ (J_{x,\lambda})]^{-1}\ vect(\bigtriangledown_{\Theta^{(curr)}}\ (J_{x,\lambda}))$

$\Theta^{(curr)} \leftarrow \Theta^{(new)}$

Update all the Θ-dependent quantities including
i.) A_Θ
ii.) D_Θ
iii.) $(S_{uk})_\Theta,\ u,k = 1,...,p$
iv.) $J_{x,\lambda}(\Theta)$
v.) $\bigtriangledown_\Theta\ (J_{x,\lambda})$
vi.) $H_\Theta\ (J_{x,\lambda})$
}

Here too, the update rule is simultaneous. This means that we keep using $\Theta^{(curr)}$ until all entries have been calculated, at which point we update $\Theta^{(curr)}$ to $\Theta^{(new)}$.

NR is usually faster to converge when measured in terms of number of iterations. However, convergence depends on the invertibility of the Hessian at each iteration. This is not always guaranteed (e.g., multinomial classification with more than two classes). Alternatively, one could circumvent this problem by applying a modified version of NR as described in e.g., [2]. Furthermore, even when convergence is guaranteed, NR can be computationally taxing due to the matrix inversion operation at each iteration. Taking that into account, usage of NR is usually preferred whenever the dimensions of H are small (i.e., small p and n).

6 Specific distribution examples

In what follows, we consider a number of probability distributions whose dispersion matrix is of the form ρI_p where ρ is a positive dispersion scalar and I_p is the $(p \times p)$ identity matrix for $p \geq 1$. In order to devise the relevant discriminative supervised learning model associated with an instance of the class of such distributions, we proceed as follows:

Step 1: Identify the dimensions of the target matrix $Y \in \mathbb{R}^{m \times r}$, where m is the

number of training examples and r the dimension of each output.

Step 2: Identify the natural parameter $\eta \in \mathbb{R}^p$ and dispersion parameter $\rho \in \mathbb{R}$ associated with the exponential distribution, compute the sufficient statistic matrix $T \in \mathbb{R}^{m \times p}$, compute the non-negative base measure $b(y, \rho)$ and derive the log-partition function $a : \mathbb{R}^p \to \mathbb{R}$. Identify the dimensions of the coefficient matrix $\Theta \in \mathbb{R}^{p \times (n+1)}$.

Step 3: Compute the set of p functions $\frac{\partial a}{\partial \eta_k} : \mathbb{R}^p \to \mathbb{R},\ \forall k \in \{1, ..., p\}$.

Step 4: If needed (e.g., in the case of NR algorithm), compute the set of p^2 functions $\frac{\partial^2 a}{\partial \eta_u \partial \eta_k} : \mathbb{R}^p \to \mathbb{R},\ \forall u, k \in \{1, ..., p\}$.

Step 5: Compute the log-partition vector $A_\Theta \in \mathbb{R}^m$.

Step 6: Compute the log-partition Gradient matrix $D_\Theta \in \mathbb{R}^{p \times m}$.

Step 7: If needed (e.g., in the case of NR algorithm), compute the set of p^2 second order diagonal matrices $(S_{uk})_\Theta \in \mathbb{R}^{m \times m}$.

Step 8: Compute the Cost Function $J_{x,\lambda}(\Theta)$, its Gradient $\bigtriangledown_\Theta\ (J_{x,\lambda})$ and its Hessian $H_\Theta\ (J_{x,\lambda})$.

Step 9: Calculate the optimal Θ^* using e.g., BGD, SGD, NR or using a closed form solution if applicable.

Step 10: Test the model on an adequate test set and then conduct predictions on new input using the mean function h that maps $\eta = \Theta x$ to

$$h(\eta) = \begin{bmatrix} \frac{\partial a}{\partial \eta_1}(\Theta\ x) \\ .. \\ \frac{\partial a}{\partial \eta_p}(\Theta\ x) \end{bmatrix} = E[T(y) \mid x; \Theta]$$

***i.* The univariate Gaussian distribution:** The corresponding probability distribution is:

$$p(y;\ \mu, \sigma) = \frac{1}{\sqrt{(2\pi\sigma^2)}} e^{-\frac{1}{2\sigma^2}\ (y-\mu)^2}$$

We rewite it in an equivalent form that makes it easier to identify as a member of the exponential family:

$$p(y;\ \mu, \sigma) = [\ \frac{1}{\sqrt{(2\pi\sigma^2)}}\ e^{-\frac{1}{2\sigma^2}\ y^2}\ ]\ e^{[\ \frac{1}{\sigma^2}\ (\ \mu y\ -\ \frac{1}{2}\ \mu^2\ )\ ]}$$

Step 1: The target matrix $Y \in \mathbb{R}^{m \times 1}$. In other terms, each training example $i \in \{1, .., m\}$ has a univariate output y associated with it.

Step 2: We identify the following quantities:

- The natural parameter $\eta = \mu \in \mathbb{R}$

- The sufficient statistic is $T(y) = y$. Matrix $T = Y \in \mathbb{R}^{m\times 1}$ (here, $p = 1$)
- The dispersion matrix is $\rho I_p = \frac{1}{\sigma^2} I_1 = \frac{1}{\sigma^2}$
- The non-negative base measure is $b(y, \rho) \; = \; \sqrt{\frac{\rho}{(2\pi)}}\; e^{-\frac{1}{2}\;\rho y^2}$
- The log-partition function maps $\eta \in \mathbb{R}$ to:

$$a(\eta) \; = \; \frac{\mu^2}{2} \; = \; \frac{\eta^2}{2} \; \in \mathbb{R}$$

- The coefficient matrix is a row vector $\Theta \in \mathbb{R}^{1\times(n+1)}$.

Step 3: The function $\frac{\partial a}{\partial \eta} : \mathbb{R} \to \mathbb{R}$ is the identity map that takes η to η.

Step 4: If needed (e.g., in the case of NR algorithm), the function $\frac{\partial^2 a}{\partial \eta^2} : \mathbb{R} \to \mathbb{R}$, maps η to the constant value 1.

Step 5: The log-partition vector is given by:

$$A_\Theta \; = \; \frac{1}{2} \begin{bmatrix} (\Theta\; x^{(1)})^2 \\ .. \\ .. \\ (\Theta\; x^{(m)})^2 \end{bmatrix} \in \mathbb{R}^m$$

Step 6: The log-partition Gradient matrix is:

$$D_\Theta = [\; \Theta\; x^{(1)} \; .. \; \Theta\; x^{(m)} \;] \; \in \mathbb{R}^{1\times m}$$

Step 7: If needed (e.g., in the case of NR algorithm), the second order diagonal matrix $(S_{11})_\Theta \in \mathbb{R}^{m\times m}$ is the m by m identity matrix I_m.

Step 8: Compute the Cost Function $J_{x,\lambda}(\Theta)$, its Gradient $\bigtriangledown_\Theta(J_{x,\lambda})$ and its Hessian $H_\Theta(J_{x,\lambda})$.

Step 9: Calculate the optimal Θ^* using e.g., BGD, SGD, NR.

Step 10: Test the model on an adequate test set and then conduct predictions on new input using the mean function h that maps $\eta = \Theta^*\; x$ to:

$$h(\eta) \; = \; \frac{\partial a}{\partial \eta}(\Theta^*\; x) = \Theta^*\; x.$$

The supervised learning model associated with the univariate Gaussian distribution coincides with the familiar linear regression model when we consider the short form basic Cost Function $J^{(SB)}$. Recall that the basic form has no dependence on either the regularization parameter λ or the input x. In this case, the weight matrix W_x is equal to I_m and the regularization matrix L_λ is equal to $0^{(n+1)\times(n+1)}$.

To see this equivalence, we rewrite equation (25) as:

$$J^{(SB)}(\Theta) \;=\; \frac{1}{m}\,\Sigma_{i=1}^{m}\,[\;a(\Theta\;x^{(i)}) \;-\; [T(y^{(i)})]^T\;\Theta\;x^{(i)}\;]\;=$$
$$\frac{1}{m}\,\Sigma_{i=1}^{m}\,[\;\frac{1}{2}\;(\Theta\;x^{(i)})^2 \;-\; y^{(i)}\;\Theta x^{(i)}\;]\;=$$
$$\frac{1}{2m}\,\Sigma_{i=1}^{m}\,(y^{(i)}-\Theta x^{(i)})^2 \;-\; \frac{1}{2m}\,\Sigma_{i=1}^{m}\,(y^{(i)})^2$$

Minimizing $J^{(SB)}$ over Θ is equivalent to minimizing $\Sigma_{i=1}^{m}\,(y^{(i)}-\Theta x^{(i)})^2$ over Θ. We thus retrieve the familiar least-square model.

It also turns out that one can calculate the optimal coefficient matrix Θ^* in closed form. Recalling that $J^{(SB)}$ is convex in Θ, we can find Θ^* by setting $\bigtriangledown_\Theta\;J^{(SB)}$ equal 0. Substituting W_x with $I_m,\;L_\lambda$ with $0^{(n+1)\times(n+1)},\;T$ with Y and D_Θ with $\Theta\;X^T$ in equation (30), we get:

$$\bigtriangledown_\Theta\;(J^{(SB)})\;=\;\tfrac{1}{m}\;[\;(\Theta\;X^T\;-\;Y^T)\;X\;]$$

Setting this Gradient to 0 leads to the **normal equation** that allows us to solve for Θ^* in closed form:

$$\Theta^* = (Y^T\;X)\;(X^T\;X)^{-1}$$

ii. **The Bernoulli (Binomial) distribution:** The corresponding probability distribution is:

$$p(y;\;\phi) = \phi^y\;(1-\phi)^{(1-y)}$$

where the outcome y is an element of $\{0,1\}$ and where $p(y=1)\;=\;\phi$ and $p(y=0)\;=\;(1-\phi)$. We rewrite it in an equivalent form that makes it easier to identify as a member of the exponential family:

$$p(y;\;\phi)\;=\;e^{[\;y\;ln(\phi)\;+\;(1-y)\;ln(1-\phi)\;]}\;=\;e^{[\;y\;ln(\frac{\phi}{1-\phi})\;+\;ln(1-\phi)\;]}$$

Step 1: The target matrix $Y \in \{0,1\}^{m\times 1}$. In other terms, each training example $i \in \{1,..,m\}$ has a binary output y associated with it.

Step 2: We identify the following quantities:

- The natural parameter $\eta\;=\;ln(\frac{\phi}{1-\phi}) \in \mathbb{R}\;$ (and so $\;\phi\;=\;\frac{e^\eta}{1+e^\eta}$)
- The sufficient statistic is $T(y)=y.\;$ Matrix $T=Y \in \{0,1\}^{m\times 1}\;$ (here, $p=1$)
- The dispersion matrix is $\rho = I_p = I_1 = 1$
- The non-negative base measure is $b(y,\rho)=b(y)=1$
- The log-partition function maps $\eta \in \mathbb{R}$ to:

$$a(\eta) \;=\; -ln(\;1-\phi\;) \;=\; ln(\;1+e^{\eta}\;) \in \mathbb{R}$$

- The coefficient matrix is a row vector $\Theta \in \mathbb{R}^{1\times(n+1)}$.

Step 3: The function $\frac{\partial a}{\partial \eta} : \mathbb{R} \to \mathbb{R}$ maps η to $\frac{e^{\eta}}{1\;+\;e^{\eta}}$

Step 4: If needed (e.g., in the case of NR algorithm), the function $\frac{\partial^2 a}{\partial \eta^2} : \mathbb{R} \to \mathbb{R}$, maps η to $\frac{e^{\eta}}{(1\;+\;e^{\eta})^2}$

Step 5: The log-partition vector is given by:

$$A_{\Theta} = \begin{bmatrix} ln(\;1\;+\;e^{\Theta x^{(1)}}\;) \\ .. \\ ln(\;1\;+\;e^{\Theta x^{(m)}}\;) \end{bmatrix} \in \mathbb{R}^m$$

Step 6: The log-partition Gradient matrix is:

$$D_{\Theta} = [\; \frac{e^{\Theta x^{(1)}}}{1\;+\;e^{\Theta x^{(1)}}} \;\; .. \;\; \frac{e^{\Theta x^{(m)}}}{1\;+\;e^{\Theta x^{(m)}}}\;] \;\in \mathbb{R}^{1\times m}$$

Step 7: If needed (e.g., in the case of NR algorithm), the second order diagonal matrix $(S_{11})_{\Theta} \in \mathbb{R}^{m\times m}$ is given by:

$$(S_{11})_{\Theta} = \begin{bmatrix} \frac{e^{\Theta x^{(1)}}}{(1\;+\;e^{\Theta x^{(1)}})^2} & 0 & .. & 0 & 0 \\ 0 & \frac{e^{\Theta x^{(2)}}}{(1\;+\;e^{\Theta x^{(2)}})^2} & .. & 0 & 0 \\ 0 & 0 & .. & 0 & 0 \\ 0 & 0 & .. & 0 & \frac{e^{\Theta x^{(m)}}}{(1\;+\;e^{\Theta x^{(m)}})^2} \end{bmatrix}$$

Step 8: Compute the Cost Function $J_{x,\lambda}(\Theta)$, its Gradient $\bigtriangledown_{\Theta}\;(J_{x,\lambda})$ and its Hessian $H_{\Theta}\;(J_{x,\lambda})$.

Step 9: Calculate the optimal Θ^* using e.g., BGD, SGD, NR.

Step 10: Test the model on an adequate test set and then conduct predictions on new input using the mean function h (also known as the **sigmoid function**) that maps $\eta = \Theta^* x$ to:

$$h(\eta) = \frac{\partial a}{\partial \eta}(\Theta^*\;x) = \frac{e^{\Theta^* x}}{1\;+\;e^{\Theta^* x}} = \frac{1}{1\;+\;e^{-\Theta^* x}}$$

Note that:

$$E[\;y\;|\;x;\;\Theta\;] = p(y=1\;|\;x;\;\Theta)\times 1\;+\;p(y=0\;|\;x;\;\Theta)\times 0\;=\;p(y=1\;|\;x;\;\Theta)$$

Moreover, we know that:

$$E[\ y \mid x;\ \Theta\ ] = E[\ T(y) \mid x;\ \Theta\ ] = h(\Theta^*\ x)$$

As a result, we conclude that:

$$p(y = 1 \mid x; \Theta) = h(\Theta^*\ x)$$

This is none else than the familiar **logistic regression** model where we predict $y = 1$ whenever $h(\Theta^*\ x) \geq \frac{1}{2}$ and $y = 0$ otherwise. This classification highlights the presence of a **decision boundary** given by the set of inputs x that satisfy $\Theta^*\ x = 0$. This is justified by the fact that $h(\Theta^*\ x) \geq \frac{1}{2} \iff \Theta^*\ x \geq 0$.

iii. **The Poisson distribution:** The probability distribution with poisson rate λ is:

$$p(y|;\ \lambda) = \frac{\lambda^y\ e^{-\lambda}}{y!}$$

The outcome $y \in \mathbb{N}$ is usually a count of an event occurrence. We can rewrite the distribution in an equivalent form that makes it easier to identify as a member of the exponential family:

$$p(y;\ \lambda) = \frac{1}{y!}\ e^{[\ y\ ln(\lambda)\ -\ \lambda\ ]}$$

Step 1: The target matrix $Y \in \mathbb{N}^{m\times 1}$. In other terms, each training example $i \in \{1, .., m\}$ has a non-negative integer output y associated with it.

Step 2: We identify the following quantities:

- The natural parameter $\eta\ =\ ln(\lambda) \in \mathbb{R}$ (and so $\lambda\ =\ e^{\eta}$)
- The sufficient statistic is $T(y) = y$. Matrix $T = Y \in \mathbb{N}^{m\times 1}$ (here, $p = 1$)
- The dispersion matrix is $\rho = I_p = I_1 = 1$
- The non-negative base measure is $b(y, \rho) = b(y) = \frac{1}{y!}$
- The log-partition function maps $\eta \in \mathbb{R}$ to:

$$a(\eta)\ =\ \lambda\ =\ e^{\eta}\ \in \mathbb{R}$$

- The coefficient matrix is a row vector $\Theta \in \mathbb{R}^{1\times(n+1)}$.

Step 3: The function $\frac{\partial a}{\partial \eta} : \mathbb{R} \to \mathbb{R}$ maps η to e^{η}

Step 4: If needed (e.g., in the case of NR algorithm), the function $\frac{\partial^2 a}{\partial \eta^2} : \mathbb{R} \to \mathbb{R}$, maps η to e^{η}

Step 5: The log-partition vector is given by:

$$A_\Theta = \begin{bmatrix} e^{\Theta x^{(1)}} \\ .. \\ e^{\Theta x^{(m)}} \end{bmatrix} \in \mathbb{R}^m$$

Step 6: The log-partition Gradient matrix is:

$$D_\Theta = [\ e^{\Theta x^{(1)}}\ ..\ e^{\Theta x^{(m)}}\ ]\ \in \mathbb{R}^{1\times m}$$

Step 7: If needed (e.g., in the case of NR algorithm), The second order diagonal matrix $(S_{11})_\Theta \in \mathbb{R}^{m\times m}$ is given by:

$$(S_{11})_\Theta = \begin{bmatrix} e^{\Theta x^{(1)}} & 0 & .. & 0 & 0 \\ 0 & e^{\Theta x^{(2)}} & .. & 0 & 0 \\ 0 & 0 & .. & 0 & 0 \\ 0 & 0 & .. & 0 & e^{\Theta x^{(m)}} \end{bmatrix}$$

Step 8: Compute the Cost Function $J_{x,\lambda}(\Theta)$, its Gradient $\bigtriangledown_\Theta\ (J_{x,\lambda})$ and its Hessian $H_\Theta\ (J_{x,\lambda})$.

Step 9: Calculate the optimal Θ^* using e.g., BGD, SGD, NR.

Step 10: Test the model on an adequate test set and then conduct predictions on new input using the mean function h that maps $\eta = \Theta^*\ x$ to:

$$h(\eta^*) = \frac{\partial a}{\partial \eta}(\Theta^*\ x) = e^{\Theta^*\ x}$$

Note that the range of h is $\mathbb{R}^+$, although the sufficient statistic is in $\mathbb{N}$. This is because the mean function outputs the expected value of the sufficient statistic. So one would still need to map the expected value outputed by h to an integer in $\mathbb{N}$.

iv. **The Geometric distribution:** The corresponding probability distribution is:

$$p(y;\ \phi)\ =\ (1-\phi)^{(y-1)}\ \phi$$

The outcome y is $\in \mathbb{N}^+$. The distribution calculates the probability of having a success after exactly y trials, where the probability of success is equal to ϕ. We rewrite it in an equivalent form that makes it easier to identify as a member of the exponential family:

$$p(y;\ \phi)\ =\ e^{[\ (y-1)\ ln(1-\phi)\ +\ ln(\phi)\ ]}\ =\ e^{[\ y\ ln(1-\phi)\ +\ ln(\frac{\phi}{1-\phi})\ ]}$$

Step 1: The target matrix $Y \in (\mathbb{N}^+)^{m\times 1}$. In other terms, each training example $i \in \{1, .., m\}$ has a positive integer output y associated with it.

Step 2: We identify the following quantities:

- The natural parameter $\eta \ = \ ln(1-\phi) \in \mathbb{R}$ (and so $\phi \ = \ 1-e^{\eta}$)
- The sufficient statistic is $T(y) = y$. Matrix $T = Y \in \mathbb{N}^{m\times 1}$ $(p = 1)$
- The dispersion matrix is $\rho = I_p = I_1 = 1$
- The non-negative base measure is $b(y, \rho) = b(y) = 1$
- The log-partition function maps $\eta \in \mathbb{R}$ to:

$$a(\eta) \ = \ -ln(\ \frac{\phi}{1-\phi}\ ) \ = \ ln(\frac{e^{\eta}}{1-e^{\eta}}) \ \in \mathbb{R}$$

- The coefficient matrix is a row vector $\Theta \in \mathbb{R}^{1\times(n+1)}$.

Step 3: The function $\frac{\partial a}{\partial \eta} : \mathbb{R} \to \mathbb{R}$ maps η to $\frac{1}{1-e^{\eta}}$

Step 4: If needed (e.g., in the case of NR algorithm), the function $\frac{\partial^2 a}{\partial \eta^2} : \mathbb{R} \to \mathbb{R}$, maps η to $\frac{e^{\eta}}{(1-e^{\eta})^2}$

Step 5: The log-partition vector is given by:

$$A_{\Theta} = \begin{bmatrix} ln(\ \frac{e^{\Theta x^{(1)}}}{(\ 1\ -\ e^{\Theta x^{(1)}}\ )}\ ) \\ .. \\ ln(\ \frac{e^{\Theta x^{(m)}}}{(\ 1\ -\ e^{\Theta x^{(m)}}\ )}\ ) \end{bmatrix} \in \mathbb{R}^m$$

Step 6: The log-partition Gradient matrix is:

$$D_{\Theta} \ = \ [\ \frac{1}{1\ -\ e^{\Theta x^{(1)}}} \ \ .. \ \ \frac{1}{1\ -\ e^{\Theta x^{(m)}}}\ ] \in \mathbb{R}^{1\times m}$$

Step 7: If needed (e.g., in the case of NR algorithm), the second order diagonal matrix $(S_{11})_{\Theta} \in \mathbb{R}^{m\times m}$ is given by:

$$(S_{11})_{\Theta} = \begin{bmatrix} \frac{e^{\Theta x^{(1)}}}{(1\ -\ e^{\Theta x^{(1)}})^2} & 0 & .. & 0 & 0 \\ 0 & \frac{e^{\Theta x^{(2)}}}{(1\ -\ e^{\Theta x^{(2)}})^2} & .. & 0 & 0 \\ 0 & 0 & .. & 0 & 0 \\ 0 & 0 & .. & 0 & \frac{e^{\Theta x^{(m)}}}{(1\ -\ e^{\Theta x^{(m)}})^2} \end{bmatrix}$$

Step 8: Compute the Cost Function $J_{x,\lambda}(\Theta)$, its Gradient $\bigtriangledown_{\Theta}\ (J_{x,\lambda})$ and its Hessian $H_{\Theta}\ (J_{x,\lambda})$.

Step 9: Calculate the optimal Θ^* using e.g., BGD, SGD, NR.

Step 10: Test the model on an adequate test set and then conduct predictions on new input using the mean function h that maps $\eta = \Theta^*\ x$ to:

$$h(\eta) = \frac{\partial a}{\partial \eta}(\Theta^*\ x) = \frac{1}{(1\ -\ e^{\Theta^*\ x})}$$

Note that the range of h is $\mathbb{R}$ although the sufficient statistic is in $\mathbb{N}^+$. This is because the mean function outputs the expected value of the sufficient statistic. So one would still need to map the expected value outputed by h to an integer in $\mathbb{N}^+$.

***v.* The Multinomial distribution:** The corresponding probability distribution is:

$$p(y;\ \phi_1, ..., \phi_{k-1}) \ = \ \Pi_{i=1}^{k-1}\ \phi_i^{1\{y=i\}}\ (\ 1 - \Sigma_{j=1}^{k-1}\ \phi_j\ )^{1\{y=k\}}$$

Here, $1\{y = i\}$ is a function of y that returns 1 if and only if $y = i$, and returns 0 otherwise. One can think of this distribution as the probability of y taking a value in the set $\{1, ..., k\}$, with:

$$\phi_1 \ = \ p(y = 1)$$

$$...$$

$$\phi_{k-1} \ = \ p(y = (k-1))$$

$$\phi_k \ = \ p(y = k) \ = \ 1 - \Sigma_{i=1}^{k-1}\ \phi_i.$$

We rewrite it in an equivalent form that makes it easier to identify as a member of the exponential family:

$$p(y;\ \phi_1, ..., \phi_{k-1}) \ = \ e^{\ ln\ [\ \Pi_{i=1}^{k-1}\ \phi_i^{1\{y=i\}}\ (\ 1-\Sigma_{j=1}^{k-1}\ \phi_j\ )^{1\{y=k\}}\ ]} \ =$$

$$e^{[\ \Sigma_{i=1}^{k-1}\ 1\{y=i\}\ ln(\phi_i)\ +\ 1\{y=k\}\ ln(\ 1-\Sigma_{j=1}^{k-1}\ \phi_j\ )\ ]} \ =$$

$$e^{[\ \Sigma_{i=1}^{k-1}\ 1\{y=i\}\ ln(\phi_i)\ +\ (\ 1-\Sigma_{i=1}^{k-1}\ 1\{y=i\}\ )\ ln(\ 1-\Sigma_{j=1}^{k-1}\ \phi_j\ )\ ]} \ =$$

$$e^{[\ \Sigma_{i=1}^{k-1}\ 1\{y=i\}\ ln(\frac{\phi_i}{(\ 1-\Sigma_{j=1}^{k-1}\ \phi_j\ )})\ +\ ln(\ 1-\Sigma_{j=1}^{k-1}\ \phi_j\ )\ ]}$$

Step 1: The target matrix $Y \in \{1, .., k\}^{m\times 1}$. In other terms, each training example $i \in \{1, .., m\}$ has an integer output $1 \leq y \leq k$ associated with it.

Step 2: We identify the following quantities:

- The natural parameter is given by:

$$\eta \equiv [\eta_1\ ...\ \eta_{k-1}]^T \ = \ [\ ln(\frac{\phi_1}{(\ 1 - \Sigma_{j=1}^{k-1}\ \phi_j)}\ )\ \ ..\ \ ln(\frac{\phi_{k-1}}{(\ 1 - \Sigma_{j=1}^{k-1}\ \phi_j)}\ )\ ]^T \ \in \mathbb{R}^{k-1}$$

- The sufficient statistic of y is:

$$T(y) = [\ 1\{y = 1\}\ \ ..\ \ 1\{y = k-1\}\ ]^T$$

As a result, the sufficient statistic matrix is given by:

$$T = \begin{bmatrix} 1\{y^{(1)} = 1\} & .. & 1\{y^{(1)} = (k-1)\} \\ .. & .. & .. \\ 1\{y^{(m)} = 1\} & .. & 1\{y^{(m)} = (k-1)\} \end{bmatrix} \in \{0,1\}^{m \times (k-1)}$$

Here, $p = (k-1)$.

- The dispersion matrix is $\rho = I_p = I_{k-1}$
- The non-negative base measure is $b(y, \rho) = b(y) = 1$
- The log-partition function maps $\eta \in \mathbb{R}$ to:

$$\begin{aligned} a(\eta) \ &= \ -ln(\ 1 - \Sigma_{j=1}^{k-1}\ \phi_j\ ) \ = \ ln(\ \frac{1}{1 - \Sigma_{j=1}^{k-1}\ \phi_j}\ ) \\ &= \ ln(\ \Sigma_{j=1}^{k-1}\ e^{\eta_j}\ +\ 1\ ) \ \in \mathbb{R} \end{aligned}$$

- The coefficient matrix is given by:

$$\Theta = \begin{bmatrix} \Theta_1^T \\ .. \\ \Theta_{(k-1)}^T \end{bmatrix} = \begin{bmatrix} \theta_{11} & .. & \theta_{1(n+1)} \\ .. & .. & .. \\ \theta_{(k-1)1} & .. & \theta_{(k-1)(n+1)} \end{bmatrix} \in \mathbb{R}^{(k-1)\times(n+1)}$$

Step 3: $\forall s \in \{1, .., k-1\}$, the function $\frac{\partial a}{\partial \eta_s} : \mathbb{R}^{k-1} \to \mathbb{R}$ corresponds to the following mapping:

$$\eta \ \equiv \ [\eta_1\ ...\ \eta_{(k-1)}]^T \ \to \ \frac{e^{\eta_s}}{1\ +\ \Sigma_{j=1}^{k-1}\ e^{\eta_j}}$$

Step 4: If needed (e.g., in the case of NR algorithm), $\forall s, t \in \{1, .., k-1\}$, the function $\frac{\partial^2 a}{\partial \eta_s \partial \eta_t} : \mathbb{R}^{k-1} \to \mathbb{R}$ can be computed as follows:

$$\eta \ \equiv \ [\eta_1\ ...\ \eta_{k-1}]^T \ \to \ \frac{1}{(\ 1\ +\ \Sigma_{j=1}^{k-1}\ e^{\eta_j}\ )^2}\ [\ \delta_{st}\ e^{\eta_t}\ (\ 1\ +\ \Sigma_{j=1}^{k-1}\ e^{\eta_j}\ )\ -\ e^{[\ \eta_s\ +\ \eta_t\ ]}\ ]$$

where $\delta_{st} = 1$ if $s = t$ and 0 otherwise.

Step 5: The log-partition vector is given by:

$$A_\Theta = \begin{bmatrix} ln(\ 1\ +\ \Sigma_{j=1}^{k-1}\ e^{\Theta_j^T x^{(1)}}\ ) \\ .. \\ ln(\ 1\ +\ \Sigma_{j=1}^{k-1}\ e^{\Theta_j^T x^{(m)}}\ ) \end{bmatrix} \in \mathbb{R}^m$$

Step 6: The log-partition Gradient matrix is given by:

$$D_\Theta = \begin{bmatrix} \frac{e^{[\Theta_1^T x^{(1)}]}}{1 + \Sigma_{j=1}^{k-1} e^{[\Theta_j^T x^{(1)}]}} & .. & .. & \frac{e^{[\Theta_1^T x^{(m)}]}}{1 + \Sigma_{j=1}^{k-1} e^{[\Theta_j^T x^{(m)}]}} \\ .. & .. & .. & .. \\ .. & .. & .. & .. \\ \frac{e^{[\Theta_{k-1}^T x^{(1)}]}}{1 + \Sigma_{j=1}^{k-1} e^{[\Theta_j^T x^{(1)}]}} & .. & .. & \frac{e^{[\Theta_{k-1}^T x^{(m)}]}}{1 + \Sigma_{j=1}^{k-1} e^{[\Theta_{k-1}^T x^{(m)}]}} \end{bmatrix} \in \mathbb{R}^{(k-1)\times m}$$

Step 7: If needed (e.g., in the case of NR algorithm), $\forall u, v \in \{1, .., k-1\}$ the second order diagonal matrix $(S_{uv})_\Theta \in \mathbb{R}^{m\times m}$ is the diagonal matrix whose ii^{th} entry ($i \in \{1, ..., m\}$) is given by:

$$[(S_{uv})_\Theta]_{ii} = \frac{1}{(\ 1\ +\ \Sigma_{j=1}^{k-1}\ e^{[\Theta_j^T x^{(i)}]}\ )^2}\ [\ \delta_{uv}\ (\ 1 + \Sigma_{j=1}^{k-1}\ e^{\Theta_j^T x^{(i)}}\ )\ e^{\Theta_v^T x^{(i)}} - e^{[\ (\Theta_u^T + \Theta_v^T) x^{(i)}\ ]}]$$

Step 8: Compute the Cost Function $J_{x,\lambda}(\Theta)$, its Gradient $\bigtriangledown_\Theta\ (J_{x,\lambda})$ and its Hessian $H_\Theta\ (J_{x,\lambda})$.

Step 9: Calculate the optimal Θ^* using e.g., BGD, SGD, NR.

Step 10: Test the model on an adequate test set and then conduct predictions on new input using the mean function h that maps $\eta\ =\ \Theta^*\ x$ to:

$$h(\eta)\ =\ (\bigtriangledown_\eta\ a)^T\ (\Theta^*\ x)\ =\ \begin{bmatrix} \frac{e^{[\Theta_1^{*T}\ x]}}{1\ +\ \Sigma_{j=1}^{k-1}\ e^{[\Theta_j^{*T}\ x]}} \\ .. \\ \frac{e^{[\Theta_{k-1}^{*T}\ x]}}{1\ +\ \Sigma_{j=1}^{k-1}\ e^{[\Theta_j^{*T}\ x]}} \end{bmatrix}$$

Note that by definition, we have $h(\eta) = E[T(y)\ |\ x;\ \Theta]$. Substituting the previously derived expression for $T(y)$, we get:

$$\begin{bmatrix} E[\ 1\{y = 1\}\ |\ x;\ \Theta] \\ .. \\ E[\ 1\{y = k-1\}\ |\ x;\ \Theta] \end{bmatrix} = \begin{bmatrix} \frac{e^{[\Theta_1^{*T}\ x]}}{1\ +\ \Sigma_{j=1}^{k-1}\ e^{[\Theta_j^{*T}\ x]}} \\ .. \\ \frac{e^{[\Theta_{k-1}^{*T}\ x]}}{1\ +\ \Sigma_{j=1}^{k-1}\ e^{[\Theta_j^{*T}\ x]}} \end{bmatrix}$$

Observing that:

$$E[\ 1\{y = i\};\ \phi_1, .., \phi_{k-1}]\ =\ 1 \times \phi_i\ +\ 0 \times \Pi_{j=1, j\neq i}^{k-1}\ \phi_j\ =\ \phi_i$$

We get:

$$h(\Theta^*\ x) = \begin{bmatrix} \phi_1 \\ .. \\ \phi_{k-1} \end{bmatrix} \equiv \begin{bmatrix} p(y=1 \mid x;\ \Theta) \\ .. \\ p(y=(k-1) \mid x;\ \Theta) \end{bmatrix} = \begin{bmatrix} \frac{e^{[\Theta_1^{*T}\ x]}}{1\ +\ \Sigma_{j=1}^{k-1}\ e^{[\Theta_j^{*T}\ x]}} \\ .. \\ \frac{e^{[\Theta_{k-1}^{*T}\ x]}}{1\ +\ \Sigma_{j=1}^{k-1}\ e^{[\Theta_j^{*T}\ x]}} \end{bmatrix}$$

This is the familiar **softmax regression** model. We predict $y = i \in \{1,\ ..,\ k\}$ whenever $\phi_i\ =\ max_{j=1}^{k}\ \phi_j$ and where we define ϕ_k to be equal to $(\ 1 - \Sigma_{j=1}^{k-1}\ \phi_j\ )$. This classification highlights the presence of **decision boundaries** that delimit k distinct zones where the i^{th} zone correspond to the set of x vectors for which $\phi_i = max_{j=1}^{k}\ \phi_j$.

Note that logistic regression is a special case of softmax regression when $k = 2$.

7 The multivariate Gaussian distribution case

The multivariate Gaussian distribution is given by:

$$p(y;\ \mu, \Sigma)\ =\ \frac{1}{\sqrt{(2\pi)^r|\Sigma|}}\ e^{-\frac{1}{2}\ (y-\mu)^T\ \Sigma^{-1}\ (y-\mu)}$$

where $y \in \mathbb{R}^r, \mu \in \mathbb{R}^r$ and Σ is a symmetric positive definite matrix in $\mathbb{S}_{++}^r$.

Note that the symmetry and positive definiteness of Σ imply the symmetry and positive definiteness of Σ^{-1} as we now demonstrate.

- **Proof that Σ^{-1} is symmetric**: We claim that for any invertible matrix A, it holds that $(A^T)^{-1}\ =\ (A^{-1})^T$. Indeed, letting I denote the identity matrix, we have the following chain of equalities:

$$A^{-1}\ A\ =\ I\ =\ I^T\ =\ (\ A^{-1}\ A\ )^T\ =\ A^T\ (A^{-1})^T$$

  It ensues that $(A^T)^{-1}\ =\ (A^{-1})^T$. By virtue of being positive definite, Σ is invertible and we get as a result that $(\Sigma^T)^{-1}\ =\ (\Sigma^{-1})^T$. Invoking the symmetric nature of Σ, we then conclude that $\Sigma^{-1}\ =\ (\Sigma^{-1})^T$. This shows that Σ^{-1} is symmetric.

- **Proof that Σ^{-1} is positive definite**: Being positive definite, Σ exclusively admits positive eigenvalues. But the eigenvalues of Σ^{-1} are the reciprocals of those of Σ and hence are also positive. This in turn implies that Σ^{-1} is positive definite.

We rewrite the probability distribution in an equivalent form that makes it easier to identify as a member of the exponential family with dispersion matrix:

$$p(y;\ \mu, \Sigma)\ =\ \frac{1}{\sqrt{(2\pi)^r |\Sigma|}}\ e^{-\frac{1}{2}\ (y^T \Sigma^{-1} y)}\ e^{[\ \frac{1}{2}\ (y^T\ \Sigma^{-1}\ \mu)\ +\ \frac{1}{2}\ (\mu^T\ \Sigma^{-1}\ y)\ -\ \frac{1}{2}\ (\mu^T\ \Sigma^{-1}\ \mu)\ ]}$$

Being a scalar, the quantity $y^T\ \Sigma^{-1}\ \mu$ is equal to its transpose $\mu^T\ (\Sigma^{-1})^T\ y$. And since Σ^{-1} is symmetric, it must be that $y^T\ \Sigma^{-1}\ \mu\ =\ \mu^T\ \Sigma^{-1}\ y$. Moreover, the determinant of Σ^{-1} satisfies $|\Sigma^{-1}| = \frac{1}{|\Sigma|}$. As a result, we write:

$$p(y;\ \mu, \Sigma^{-1})\ =\ \sqrt{\frac{|\Sigma^{-1}|}{(2\pi)^r}}\ e^{-\frac{1}{2}\ (y^T\ \Sigma^{-1}\ y)}\ e^{[\ (\mu^T\ \Sigma^{-1}\ y)\ -\ \frac{1}{2}\ (\mu^T\ \Sigma^{-1}\ \mu)\ ]}$$

Step 1: The target matrix $Y \in \mathbb{R}^{m \times r}$. In other terms, each training example $i \in \{1,\ ..,\ m\}$ has a multivariate output $y \in \mathbb{R}^r$ associated with it.

Step 2: We identify the following quantities:

- The natural parameter vector $\eta\ =\ \mu\ \in \mathbb{R}^p$.
- The sufficient statistic is $T(y)\ =\ y$. Matrix $T\ =\ Y\ \in \mathbb{R}^{m \times p}$. Note that the dimension p of the sufficient statistic is equal to the dimension r of y.
- The dispersion matrix is $\Lambda\ =\ \Sigma^{-1}$ which is **not** necessarily a positive multiple of the identity matrix I_p. The inverse of the covariance matrix Σ is also usually known as the **precision** matrix.
- The non-negative base measure is $b(y, \Lambda)\ =\ \sqrt{\frac{|\Lambda|}{(2\pi)^r}}\ e^{-\frac{1}{2}\ (y^T\ \Lambda\ y)}$.
- The log-partition function maps $(\eta, \Lambda)\ \in \mathbb{R}^p \times \mathbb{R}^{p \times p}$ to:

$$c(\eta, \Lambda)\ =\ \left(\frac{\mu^T}{\sqrt{2}}\right)\ \Lambda\ \left(\frac{\mu}{\sqrt{2}}\right)\ =\ \left(\frac{\eta^T}{\sqrt{2}}\right)\ \Lambda\ \left(\frac{\eta}{\sqrt{2}}\right)\ \in \mathbb{R}$$

  where $q : \mathbb{R}^p \to \mathbb{R}^p$ is the map that takes $\eta\ \equiv\ [\eta_1\ ..\ \eta_p]^T$ to $q(\eta)\ =\ \frac{\eta}{\sqrt{2}}$

  We can also compute the derivative of q with respect to η :

$$(\mathcal{D}_\eta\ q)\ =\ \begin{bmatrix} \frac{\partial q_1}{\partial \eta_1} & .. & \frac{\partial q_1}{\partial \eta_p} \\ .. & .. & .. \\ \frac{\partial q_p}{\partial \eta_1} & .. & \frac{\partial q_p}{\partial \eta_p} \end{bmatrix} (\eta)\ = \frac{1}{\sqrt{2}}\ I_p$$

- The coefficient matrix is $\Theta \in \mathbb{R}^{p \times (n+1)}$.

Step 3: We claim that the Gradient of c with respect to η is the map from $\mathbb{R}^p$ to $\mathbb{R}^p$ that takes $\eta \equiv [\eta_1\ ..\ \eta_p]^T$ to $(\bigtriangledown_\eta\ c)^T = \Lambda\ \eta$

To see why, note that if $A \in \mathbb{R}^{p \times p}$ is a given matrix and $f : \mathbb{R}^p \rightarrow \mathbb{R}^p$ maps vector $x = [x_1 \; .. \; x_p]^T$ to $f(x) = x^T \; A \; x$, then:

$\{ \; f(x) = \Sigma_{j=1}^{p} \; \Sigma_{k=1}^{p} \; x_j \; A_{jk} \; x_k$. As a result, $\forall i \in \{1, .., p\}$:

$$\frac{\partial f}{\partial x_i} \; = \; \Sigma_{j=1}^{p} \; \Sigma_{k=1}^{p} \; \{ \; \delta_{ij} \; (A_{jk} \; x_k) + \; (x_j \; A_{jk}) \; \delta ik \; \} \; =$$

$$\Sigma_{j=1}^{p} \; \Sigma_{k=1}^{p} \; \{ \; A_{ik} \; x_k + x_j \; A_{ji} \; \} \; = \; \Sigma_{j=1}^{p} \; \{ \; (A_{ij} \; + A_{ji}) \; x_j \; \}$$

$\{$ And so: $(\bigtriangledown_x \; f)^T \; = \; [\; \frac{\partial f}{\partial x_1} \; .. \; \frac{\partial c}{\partial \eta_p} \;]^T \; = \; (A + A^T) \; x$

$\{$ We then conclude that: $(\bigtriangledown_\eta \; c)^T \; = \; \frac{1}{2} \; (\Lambda + \Lambda^T) \; \eta \; = \; \Lambda \; \eta$

Step 4: We now turn to computing the cost function $J_{x,\lambda}^{(LG)}(\Theta, \Lambda)$ derived from conducting maximum likelihood estimation using the multivariate Gaussian distribution. After substituting the relevant parameters in equation (18), we get:

$$J_{x,\lambda}^{(LG)}(\Theta, \Lambda) \; = \; \frac{1}{m} \; \Sigma_{i=1}^{m} \; w_x^{(i)} \; \{ \; \frac{1}{2} \; (x^{(i)})^T \; \Theta^T \; \Lambda \; \Theta \; x^{(i)} \; - \; (x^{(i)})^T \; \Theta^T \; \Lambda \; y^{(i)} \; + \; \frac{1}{2} \; (y^{(i)})^T \; \Lambda \; y^{(i)} \; + \; \frac{1}{2} \; ln(\; (2\pi)^p \; \left|\Lambda^{-1}\right| \;) \; \} \; + \; \frac{\lambda}{2} \; \Sigma_{j=1}^{p} \; [\; \Theta_j^T \; \Theta_j \; + \Lambda_j^T \; \Lambda_j \;]$$

We rewrite this more concisely using matrix notation as follows:

$$\boxed{J_{x,\lambda}^{(LG)}(\Theta, \Lambda) \; = \; \frac{1}{m} \; Tr\{ \; [\; \frac{1}{2} \; X \; \Theta^T \Lambda \; \Theta \; X^T \; - \; X \; \Theta^T \Lambda \; Y^T \; + \; \frac{1}{2} \; Y \; \Lambda \; Y^T \; + \; \frac{1}{2} \; ln(\; (2\pi)^p \; \left|\Lambda^{-1}\right| \;) \; I_p \;] \; W_x \; \} \; + \frac{\lambda}{2} \; [\; Tr(\Theta \; \Theta^T) \; + \; Tr(\Lambda \; \Lambda^T) \;]} \tag{32}$$

We have seen earlier that the long form general Cost Function is convex in Θ. Moreover, we showed that if $b(y, \Lambda)$ is convex in Λ, then so will the Cost Function. We now demonstrate that for the case of the multivariate Gaussian, $b(y, \Lambda)$ is indeed convex in Λ. To do so, we prove the equivalent claim that $-ln(\; b(y, \Lambda) \;)$ is convex in Λ. To see why, note that:

$$-ln(\; b(y, \Lambda) \;) \; = \; \frac{1}{2} \; Y^T \; \Lambda \; Y \; - \; \frac{1}{2} \; ln(\; |\Lambda| \;) \; + \; \frac{1}{2} \; ln(\; (2 \; \pi)^r \;)$$

Note that the first term is linear in Λ and hence convex in Λ. The last term is independent of Λ. The second term is a negative multiple of the natural logarithm of the determinant of a positive definite matrix. To establish the convexity of this quantity, we first find the Gradient of $-\frac{1}{2} \; ln(\; |\Lambda| \;)$ which turns out to be equal to

$-\frac{1}{2}\ \Lambda^{-1}$ (for a proof, the reader can refer to e.g., section 4.A.1 of [1]). We then compute the Hessian which turns out to be equal to $\frac{1}{2}\ \Lambda^{-2}$. This shows that the Hessian is positive definite which implies that $-\frac{1}{2}\ ln(\ |\Lambda|\ )$ is convex in Λ.

Step 5: We now turn to computing the Cost Function's Gradient as specified by equation (19). In order to do so, we first prove some identities involving the derivative of a Trace.

- **Gradient of** $Tr\left(\ \frac{1}{2}\ X\ \Theta^T\ \Lambda\ \Theta\ X^T\ W_x\ \right)$**:**

  In what follows, the quantity δ_{ij} evaluates to 1 if $i = j$ and 0 otherwise. First of all, note that:

$$Tr\left(\ \frac{1}{2}\ X\ \Theta^T\Lambda\ \Theta\ X^T\ W_x\ \right)\ =\ \sum_{i,j,k,l,m,n}\ X_{ij}\ \theta_{kj}\ \rho_{kl}\ \theta_{lm}\ X_{nm}\ (W_x)_{ni}$$

  $\forall u \in \{1,..,p\}$ and $j \in \{1,..,(n+1)\}$, we can express the $(uj)^{th}$ component of its derivative with respect to Θ as follows:

$$\left[\ \bigtriangledown_\Theta\ \ Tr\left(\ \frac{1}{2}\ X\ \Theta^T\ \Lambda\ \Theta\ X^T\ W_x\ \right)\ \right]_{uv}\ =$$

$$\frac{1}{2}\ \frac{\partial}{\partial\theta_{uv}}\ \sum_{i,j,k,l,m,n}\ X_{ij}\ \theta_{kj}\ \rho_{kl}\ \theta_{lm}\ X_{nm}\ (W_x)_{ni}\ =$$

$$\frac{1}{2}\ \sum_{i,j,k,l,m,n}\ [\ X_{ij}\ \delta_{uk}\ \delta_{vj}\ \rho_{kl}\ \theta_{lm}\ X_{nm}\ (W_x)_{ni}$$
$$+\ \ X_{ij}\ \theta_{kj}\ \rho_{kl}\ \delta_{ul}\ \delta_{vm}\ X_{nm}\ (W_x)_{ni}\ ]\ =$$

$$\frac{1}{2}\ \sum_{i,j,k,l,m,n}\ [\ X_{iv}\ \rho_{ul}\ \theta_{lm}\ X_{nm}\ (W_x)_{ni}$$
$$+\ \ X_{ij}\ \theta_{kj}\ \rho_{ku}\ X_{nv}\ (W_x)_{ni}\ ]\ =$$

$$\frac{1}{2}\ \sum_{i,j,k,l,m,n}\ [\ \rho_{ul}\ \theta_{lm}\ X_{nm}\ (W_x)_{ni}\ X_{iv}$$
$$+\ \ \rho_{ku}\ \theta_{kj}\ X_{ij}\ (W_x)_{ni}\ X_{nv}\ ]\ =$$

$$\frac{1}{2}\ \left(\ \Lambda\ \Theta\ X^T\ W_x\ X\ \right)_{uv}\ +\ \frac{1}{2}\ \left(\ \Lambda^T\ \Theta\ X^T\ W_x^T\ X\ \right)_{uv}\ =$$

$$\left(\ \Lambda\ \Theta\ X^T\ W_x\ X\ \right)_{uv}$$

(since Λ and W_x are symmetric.)

As a result, we conclude that:

$$\boxed{\bigtriangledown_\Theta \;\; Tr\left(\frac{1}{2}\, X\; \Theta^T\; \Lambda\; \Theta\; X^T\; W_x \right) \;=\; \Lambda\; \Theta\; X^T\; W_x\; X} \tag{33}$$

Similarly, we can follow the same procedure to find that:

$$\boxed{\bigtriangledown_\Lambda \;\; Tr\left(\frac{1}{2}\, X\; \Theta^T\; \Lambda\; \Theta\; X^T\; W_x \right) \;=\; \frac{1}{2}\; (\; \Theta\; X^T\; W_x\; X\; \Theta^T\;)} \tag{34}$$

- **Gradient of** $Tr\,(\; X\; \Theta^T \Lambda\; Y^T\; W_x\;)$**:**

We first notice that:

$$Tr\,(\; X\; \Theta^T\; \Lambda\; Y^T\; W_x\;) \;=\; \sum_{i,j,k,l,m} X_{ij}\; \theta_{kj}\; \rho_{kl}\; Y_{ml}\; (W_x)_{mi}$$

Since Λ is symmetric, we can equivalently write this equality as:

$$\frac{1}{2} \sum_{i,j,k,l,m} (\; X_{ij}\; \theta_{kj}\; \rho_{kl}\; Y_{ml}\; (W_x)_{mi} \;+\; X_{ij}\; \theta_{kj}\; \rho_{lk}\; Y_{ml}\; (W_x)_{mi}\;)$$

We then apply the same logic as before and demonstrate that:

$$\boxed{\bigtriangledown_\Theta \;\; Tr\,(\; X\; \Theta^T \Lambda\; Y\; W_x\;) \;=\; \Lambda\; Y^T\; W_x\; X} \tag{35}$$

And that:

$$\boxed{\bigtriangledown_\Lambda \;\; Tr\,(\; X\; \Theta^T \Lambda\; Y\; W_x\;) \;=\; \frac{1}{2}\; (\; \Theta\; X^T\; W_x\; Y \;+\; Y^T\; W_x\; X\; \Theta^T\;)} \tag{36}$$

- **Gradient of** $Tr\,(\; Y\; \Lambda\; Y^T\; W_x\;)$**:**

Given the lack of dependency on Θ, it is obvious that:

$$\boxed{\bigtriangledown_\Theta \;\; Tr(\; Y\; \Lambda\; Y^T\; W_x\;) \;=\; 0} \tag{37}$$

Furthermore, observing that W_x is symmetric and applying the same methodology that we previously used allow us to readily show that:

$$\boxed{\bigtriangledown_{\Lambda} \;\; Tr(\; Y \; \Lambda \; Y^T \; W_x \;) \;=\; \frac{1}{2} \; Y^T \; W_x \; Y} \tag{38}$$

- **Gradient of** $Tr\left(\; \frac{1}{2} \; ln(\; (2\pi)^p \; |\Lambda^{-1}|\;) \; I_p \; W_x \;\right)$**:**

  Here too, independence of Θ justifies that:

$$\boxed{\bigtriangledown_{\Theta} \;\; Tr\left(\; \frac{1}{2} \; ln(\; (2\pi)^p \; \left|\Lambda^{-1}\right|\;) \; I_p \; W_x \;\right) \;=\; 0} \tag{39}$$

  On the other hand, we have:

$$\bigtriangledown_{\Lambda} \;\; Tr\left(\; \frac{1}{2} \; ln(\; (2\pi)^p \; \left|\Lambda^{-1}\right|\;) \; I_p \; W_x \;\right) \;=$$

$$\bigtriangledown_{\Lambda} \;\; \left(\; \frac{1}{2} \; ln(\; (2\pi)^p \; \left|\Lambda^{-1}\right|\;) \; Tr(\; W_x \;) \;\right) \;=$$

$$Tr(\; W_x \;) \;\times\; \bigtriangledown_{\Lambda} \;\; \left(\; \frac{1}{2} \; ln(\; (2\pi)^p \; \left|\Lambda^{-1}\right|\;) \;\right) \;=$$

$$Tr(\; W_x \;) \;\times\; \bigtriangledown_{\Lambda} \;\; \left(\; \frac{1}{2} \; ln(\; \left|\Lambda^{-1}\right|\;) \;\right) \;=\; -\frac{1}{2} \; Tr\;(W_x) \;\times\; \bigtriangledown_{\Lambda} \;(\; ln(\; |\Lambda|\;)\;)$$

  We stated earlier that $\bigtriangledown_{\Lambda} \;(\; ln(\; |\Lambda|\;)\;) \;=\; \Lambda^{-1}$ (see e.g., section 4.A.1 of [1]). As a result, we conclude that:

$$\boxed{\bigtriangledown_{\Lambda} \;\; Tr\left(\; \frac{1}{2} \; ln(\; (2\pi)^p \; \left|\Lambda^{-1}\right|\;) \; I_p \; W_x \;\right) \;=\; -\frac{1}{2} \; Tr(\; W_x \;) \; \Lambda^{-1}} \tag{40}$$

- **Gradient of** $Tr(\; \Theta \; \Theta^T \;)$ **and** $Tr(\; \Lambda \; \Lambda^T \;)$**:**

  Clearly:

$$\boxed{\bigtriangledown_{\Lambda} \;\; (\; Tr(\; \Theta \; \Theta^T \;)\;) \;=\; \bigtriangledown_{\Theta} \;\; (\; Tr(\; \Lambda \; \Lambda^T \;)\;) \;=\; 0} \tag{41}$$

  Furthermore, we can apply the same calculation as before to conclude that:

$$\boxed{\bigtriangledown_{\Theta} \;\; (\; Tr(\; \Theta \; \Theta^T \;)\;) \;=\; 2 \; \Theta} \tag{42}$$

$$\boxed{\bigtriangledown_{\Lambda} \;\; (\; Tr(\; \Lambda \; \Lambda^T \;)\;) \;=\; 2 \; \Lambda} \tag{43}$$

The Gradient of the Cost Function can now be readily computed as:

$$\boxed{\bigtriangledown_{\Theta}\ (J_{x,\lambda}^{(LG)})\ (\Theta,\Lambda)\ =\ \frac{1}{m}\ (\ \Lambda\ \Theta\ X^T\ W_x\ X\ -\ \Lambda\ Y^T\ W_x\ X\ )\ +\ \lambda\ \Theta} \tag{44}$$

$$\boxed{\begin{array}{c}\bigtriangledown_{\Lambda}\ (J_{x,\lambda}^{(LG)})\ (\Theta,\Lambda)\ =\ \frac{1}{2m}\ (\ \Theta\ X^T\ W_x\ X\ \Theta^T\ -\ \Theta\ X^T\ W_x\ Y \\ \\ -\ Y^T\ W_x\ X\ \Theta^T\ +\ Y^T\ W_x\ Y\ -\ Tr\ (W_x)\ \Lambda^{-1}\ )\ +\ \lambda\ \Lambda\end{array}} \tag{45}$$

Step 6: With the Cost function and its gradient thus derived, we can use numerical algorithms to find the optimal Θ^* and Λ^* that minimize $J_{x,\lambda}^{(LG)}$. In what follows we illustrate how this can be done using BGD. Recall that R denotes the learning rate matrix that was previously introduced in section 5, *iter* is an iteration count variable and *max-iter* is the maximum number of allowed iterations before testing for convergence:

$\Theta^{(curr)}\ \leftarrow\ Initialize()$
$\Lambda^{(curr)}\ \leftarrow\ Initialize()$
(They could be initialized to e.g., $0^{p\times(n+1)}$ and $0^{p\times p}$ respectively).

For $(iter < max\text{-}iter)$
{
$\Theta^{(new)} = \Theta^{(curr)} - R\ \bigtriangledown_{\Theta^{(curr)}}\ (J_{x,\lambda}^{(LG)})\ (\Theta^{(curr)},\ \Lambda^{(curr)})$
$\Lambda^{(new)} = \Lambda^{(curr)} - R\ \bigtriangledown_{\Lambda^{(curr)}}\ (J_{x,\lambda}^{(LG)})\ (\Theta^{(curr)},\ \Lambda^{(curr)})$

$\Theta^{(curr)} \leftarrow \Theta^{(new)}$
$\Lambda^{(curr)} \leftarrow \Lambda^{(new)}$

Update all the Θ and Λ-dependent quantities including:

i.) $J_{x,\lambda}^{(LG)}\ (\Theta,\Lambda)$
ii.) $\bigtriangledown_{\Theta}\ (J_{x,\lambda}^{(LG)})$
iii.) $\bigtriangledown_{\Lambda}\ (J_{x,\lambda}^{(LG)})$
}

It is important to note that the entries of matrices $\Theta^{(new)}$ and $\Lambda^{(new)}$ must get simultaneously updated before they can be used in the next iteration.

Step 7: Test the model on an adequate test set and then conduct predictions on new input using the mean function h that maps $(\eta,\Lambda^*)\ =\ (\Theta^* x,\Lambda^*)$ to:

$$h(\eta, \Lambda) \;=\; \Lambda^{-1}\,(\bigtriangledown_{\eta}\, c)^{T} \;=\; \Lambda^{-1}\,\Lambda\,\eta \;=\; \eta \;=\; \Theta^{*}\,x$$

It turns out that one can calculate the optimal coefficient matrix Θ^{*} and the optimal dispersion matrix Λ^{*} in closed form whenever the Cost Function is expressed in its long basic form $J^{(LB)}$. In other terms, whenever we assume that the weight matrix W_x is independent of x and equal to the $(m \times m)$ identity matrix, and that the regularization matrix is independent of λ and equal to the $((n+1) \times (n+1))$ 0 matrix. In this case, equation (44) yields:

$$\bigtriangledown_{\Theta}\,(J^{(LB)}_{x,\lambda})\,(\Theta, \Lambda) \;=\; \frac{1}{m}\,(\;\Lambda\,\Theta\,X^{T}\,X \;-\; \Lambda\,Y^{T}\,X\;)$$

$$\begin{aligned}\bigtriangledown_{\Lambda}\,(J^{(LB)}_{x,\lambda})\,(\Theta, \Lambda) \;=\; \frac{1}{2m}\,(\;&\Theta\,X^{T}\,X\,\Theta^{T} \;-\; \Theta\,X^{T}\,Y \\ &-\; Y^{T}\,X\,\Theta^{T} \;+\; Y^{T}\,Y \;-\; m\,\Lambda^{-1}\;)\end{aligned}$$

Recall that the Cost Function is convex in Θ, and in Λ. We can therefore find Θ^{*} and Λ^{*} by setting the Gradient with respect to Θ and to Λ to 0. By doing so, we get:

$$\bigtriangledown_{\Theta}\,(J^{(LB)}_{x,\lambda}) \;=\; 0 \;\;\Longleftrightarrow\;\; \frac{1}{m}\,\Lambda\,(\;\Theta^{*}\,X^{T}\,X \;-\; Y^{T}\,X\;) \;=\; 0$$

Since Λ is positive definite, this is equivalent to $(\;\Theta^{*}\,X^{T}\,X \;-\; Y^{T}\,X\;) = 0$. We then retrieve the familiar **normal equation** we obtained in the univariate case:

$$\Theta^{*} = Y^{T}\,X\,(X^{T}\,X)^{-1}$$

Similarly, by setting $\bigtriangledown_{\Lambda}\,(J^{(LB)}_{x,\lambda}) \;=\; 0$ we find:

$$(\Lambda^{*})^{-1} \;=\; \frac{1}{m}\,(\Theta\,X^{T} \;-\; Y^{T})\,(\Theta\,X^{T} \;-\; Y^{T})^{T}$$

8 Python implementation

This section provides a python script that implements the GLM Supervised Learning class using matrix notation. We limit ourselves to cases where the dispersion matrix is a positive scalar multiple of the identity matrix. We emphasize that the code is meant for educational purposes and we recommend using tested packages (e.g., Scikit-Learn) to run specific predictive models.

```
import sys;
import math;
import numpy as np;


# THE GLM CLASS DEFINES AN INSTANCE OF A PREDICTION PROBLEM.
# IN ORDER TO DEFINE AN INSTANCE, ONE SHOULD PROVIDE THE FOLLOWING:
# ------------------------------------------------------------------------
# ------X_train (m by (n+1)): Training set input (including all-1 column)
# ---------------------------- m is number of training examples and n is
# ---------------------------- number of features.
# ------------------------------------------------------------------------
# ------Y_train (m by r): Output matrix for the training set
# ---------------------------- m is number of training examples and r is
# ---------------------------- the output's dimension.
# ------------------------------------------------------------------------
# ------reg_model: String defining the probability distribution to be
# ---------------------------- used for regression / classification. We
# ---------------------------- include the following (but can be expanded
# ---------------------------- further): "normal", "binomial", "poisson",
# ---------------------------- "geometric", "multinomial".
# ------------------------------------------------------------------------
# ------p: Refers to the dimension of the sufficient stastic. For most
# ---------------------------- examples used here, it is equal to 1. For
# ---------------------------- multinomial classification it is equal to
# ---------------------------- one less than the number of classes.
# ------------------------------------------------------------------------
# ------regu_lambda: Refers to the regularization parameter lambda. it is
# ---------------------------- a scalar that can be set to 0.
# ------------------------------------------------------------------------
# ------weight_flag: Should be set to 1 if one wishes to conduct weighted
# ---------------------------- regression/ classification. It relies on
# ---------------------------- the familiar bell-shaped function. In this
# ---------------------------- case, the data point where prediction is to
# ---------------------------- be made defines the weights to be used. As
# ---------------------------- a result, that specific data point must be
# ---------------------------- included as an input to the fitting process
# ---------------------------- as we later describe in the "fit" method.
# ---------------------------- If no weights are needed, set flag to 0.
# ------------------------------------------------------------------------
# ------weight_tau: Holds the bandwidth value tau of the weight function.
# ---------------------------- It is set to 1 by default and is only used
# ---------------------------- when the weight_flag is set to 1.
# ------------------------------------------------------------------------

class Glm:
    def __init__(self, X_train, Y_train, reg_model, p, regu_lambda,
                                                weight_flag, weight_tau = 1):
        self.X_train = X_train
        self.Y_train = Y_train
        self.reg_model = reg_model
        self.p = p
        self.regu_lambda = regu_lambda
        self.weight_flag = weight_flag
        self.weight_tau = weight_tau


    def display_X_train(self):
        print "\n The training set input matrix is:\n", self.X_train


    def display_Y_train(self):
        print "\n The training set output is:\n", self.Y_train


    def display_reg_model(self):
        print "\n The model is a: ", self.reg_model, " distribution"


    def display_regu_lambda(self):
        print "\n The regularization parameter is: ", self.regu_lambda
```

```
    def display_weight_flag(self):
        if (self.weight_flag == 1):
            print "\n The current prediction instance is weighted"

        elif (self.weight_flag == 0):
            print "\n The current prediction instance is not  weighted"

        else: print "\n Flag value not recognized. Please insert 1 or 0"


    def display_weight_tau(self):
        if (self.weight_flag == 1):
            print "\n The weight function bandwidth is ", self.weight_tau

        else: print "\n The current prediction is not weighted."


# THE GLM CLASS INCLUDES 3 GETTER METHODS. THESE ARE:
# ------get_m(): Returns total number of training examples.
# ------------------------------------------------------------------------
# ------get_n(): Returns total number of input features. We subtract 1
# ------------------------- to exclude X_train's all-1 column.
# ------------------------------------------------------------------------
# ------get_p(): Returns dimension of the sufficient statistic.
# ------------------------------------------------------------------------

    def get_m(self):
        return self.X_train.shape[0]

    def get_n(self):
        return self.X_train.shape[1] - 1

    def get_p(self):
        return self.p


# THE GLM CLASS LEVERAGES MATRIX NOTATION. THE FOLLOWING METHODS HELP
# GENERATE THE RELEVANT MATRICIAL QUANTITIES.
# ------weight_bell_func(x, training_i): Returns the weight specific
# ------------------------- to training example i, as dictated by input
# ------------------------- x (which is an (n+1) by 1 vector). The
# ------------------------- The weight we use is the bell-shaped one
# ------------------------- with bandwidth tau.
# ------------------------------------------------------------------------
# ------generate_W(x): Generates the weight matrix W associated with
# ------------------------- input x. When the weighted regression flag
# ------------------------- is 0, it is the (m by m) identity matrix.
# ------------------------------------------------------------------------
# ------generate_L(): Generates the ((n+1) by (n+1)) regularization
# ------------------------- matrix L. It is equal to "lambda" times the
# ------------------------- ((n+1) by (n+1)) identity matrix.
# ------------------------------------------------------------------------
# ------generate_T(): Generates the (m by p) sufficient statistic matrix
# ------------------------- T. It depends on the probabilistic model.
# ------------------------- Note that for the multinomial model, we
# ------------------------- label classes from 0 to (p-1).
# ------------------------------------------------------------------------

    def weight_bell_func(self, x, training_i):
        return math.exp(-np.dot((self.X_train[training_i,::] - x.T),
                                (self.X_train[training_i,::] - x.T).T)/
                                        float(2*(self.weight_tau**2)))


    def generate_W(self, x):
        m = self.get_m()
        W = np.eye(m, dtype=np.float64)

        if (self.weight_flag == 1):
            for i in range(m):
                W[i,i] = self.weight_bell_func(x, i)

        return W


    def generate_L(self):
        n = self.get_n()
        L = self.regu_lambda * np.eye(n+1, dtype=np.float64)

        return L
```

```
    def generate_T(self):
        p = self.get_p()
        m = self.get_m()
        T = np.zeros((m,p), dtype=np.float64)

        if (self.reg_model in ["normal", "binomial", "poisson",
                                                    "geometric"]):
            T = self.Y_train

        elif (self.reg_model == "multinomial"):
            for i in range(m):
                for j in range(p):
                    if(int(self.Y_train[i,0]) == j):
                        T[i,j] = 1

        else:
            print "No valid model has been inserted. System exiting now"
            sys.exit()

        return T


# TO GENERATE THE REMAINING RELEVANT MATRICES, THE GLM CLASS MAKES USE OF
# QUANTITIES BUILT OFF THE LOG-PARTITION FUNCTION AND ITS DERIVATIVES.
# THE LOG-PARTITION FUNCTION IS PROBABILITY DISTRIBUTION-DEPENDENT.
# ------a_func(eta): Returns the value of the relevant log-partition
# ---------------------------- function evaluated at eta. For most of the
# ---------------------------- encoded distributions, the first derivative
# ---------------------------- is a function from R to R. For the case of
# ---------------------------- the multinomial, it is a map from R^p to R.
# ------------------------------------------------------------------------
# ------a_first_deriv(eta, index): Returns the value of the partial
# ---------------------------- derivative of the log-partition function
# ---------------------------- with respect to the index's variable and
# ---------------------------- evaluated at eta. For most of the encoded
# ---------------------------- distributions, the first derivative is a
# ---------------------------- function from R to R. For the case of the
# ---------------------------- multinomial, it is a map from R^p to R.
# ------------------------------------------------------------------------
# ------a_second_deriv(eta, index1, index2): Returns the value of the 2nd
# ---------------------------- partial derivative of the log-partition
# ---------------------------- function with respect to the 2 indexes'
# ---------------------------- variables index1 and index2 evaluated at
# ---------------------------- eta. For most of the encoded distributions,
# ---------------------------- the second derivative is a function from R
# ---------------------------- to R. For the case of the multinomial, it
# ---------------------------- is a function from R^p to R.
# ------------------------------------------------------------------------

    def a_func(self, eta):
        p = self.get_p()
        if (self.reg_model == "normal"):
            return eta**2/float(2)

        elif (self.reg_model == "binomial"):
            return math.log(1+math.exp(eta))

        elif (self.reg_model == "poisson"):
            return math.exp(eta)

        elif (self.reg_model == "geometric"):
            return math.log(math.exp(eta)/float(1-math.exp(eta)))

        elif (self.reg_model == "multinomial"):
            val = 0

            for j in range(p):
                val += math.exp(eta[j])

            return math.log(val + 1)

        else:
            print "No valid model has been inserted. System exiting now"
            sys.exit()
```

```
    def a_first_deriv(self, eta, index):
        p = self.get_p()
        if (self.reg_model == "normal"):
            return eta

        elif (self.reg_model == "binomial"):
            return math.exp(eta)/float(1+math.exp(eta))

        elif (self.reg_model == "poisson"):
            return math.exp(eta)

        elif (self.reg_model == "geometric"):
            return 1/float(1-math.exp(eta))

        elif (self.reg_model == "multinomial"):
            denom = 1

            for j in range(p):
                denom += math.exp(eta[j])

            return math.exp(eta[index])/float(denom)

        else:
            print "No valid model has been inserted. System exiting now"
            sys.exit()


    def a_second_deriv(self, eta, index1, index2):
        p = self.get_p()
        if (self.reg_model == "normal"):
            return 1

        elif (self.reg_model == "binomial"):
            return math.exp(eta)/float((1+math.exp(eta))**2)

        elif (self.reg_model == "poisson"):
            return math.exp(eta)

        elif (self.reg_model == "geometric"):
            return math.exp(eta)/float((1-math.exp(eta))**2)

        elif (self.reg_model == "multinomial"):
            denom = 1

            for j in range(p):
                denom += math.exp(eta[j])

            val = -math.exp(eta[index1]+eta[index2])/float(denom**2)

            if (index1 == index2):
                val += math.exp(eta[index1])/float(denom)

            return val

        else:
            print "No valid model has been inserted. System exiting now"
            sys.exit()



# THE REMAINING RELEVANT MATRICES NEEDED TO COMPLETE THE OPTIMIZATION
# EXERCISE ARE GIVEN BELOW:
# ------generate_A(theta): Returns the (m by 1) vector whose entries
# -------------------------- are relevant values of the log-partition
# -------------------------- function. "A" is theta-dependent.
# ---------------------------------------------------------------------
# ------generate_D(theta): Returns the "D" matrix. It is theta-dependent.
# ---------------------------------------------------------------------
# ------generate_S_uk(theta, u, k): Returns the S_uk block. It is used
# -------------------------- to compute the Cost Function's Hessian.
# ---------------------------------------------------------------------

    def generate_A(self, theta):
        m = self.get_m()
        A = np.zeros((m,1), dtype = np.float64)

        for i in range (m):
            A[i] = self.a_func(np.dot(self.X_train[i,::],theta.T).T)

        return A
```

```
    def generate_D(self, theta):
        p = self.get_p()
        m = self.get_m()
        D = np.zeros((p,m),dtype = np.float64)

        for i in range(p):
            for j in range(m):
                D[i,j]= self.a_first_deriv(
                            np.dot(self.X_train[j,::],theta.T).T, i)

        return D


    def generate_S_uk(self, theta, u, k):
        m = self.get_m()
        S_uk = np.zeros((m,m),dtype = np.float64)

        for i in range(m):
            S_uk[i,i] = self.a_second_deriv(
                            np.dot(self.X_train[i,::],theta.T).T, u, k)
        return S_uk;


# THE FOLLOWING METHODS COMPUTE THE COST FUNCTION, GRADIENT, AND HESSIAN.
# ------cost_Grad_Hess(self, theta, x): Takes theta and x as input, and
# --------------------------- outputs 3 quantities:
# --------------------------- 1) Cost Function evaluated at theta and x.
# --------------------------- 2) Gradient evaluated at theta and x.
# ---------------------------    (It is a p by (n+1) matrix)
# --------------------------- 3) Hessian at theta and x.
# ---------------------------    (It is a p*(n+1) by p*(n+1) matrix)
# ------------------------------------------------------------------------
# ------stoch_Grad(x, i, theta): Used in stochastic gradient descent.
# --------------------------- Input i refers to the relevant training
# --------------------------- example while theta and x are as before.
# ------------------------------------------------------------------------

    def cost_Grad_Hess(self, theta, x):
        m = self.get_m()
        n = self.get_n()
        p = self.get_p()
        O = np.ones((m,1), dtype = np.float64)
        W = self.generate_W (x)
        L = self.generate_L()
        T = self.generate_T()
        A = self.generate_A(theta)
        D = self.generate_D(theta)

        cost = float(((O.T).dot(W).dot(A) -
            np.trace(self.X_train.dot(theta.T).dot(T.T).dot(W)))/float(m)
                + self.regu_lambda*np.trace(theta.dot(theta.T))/float(2))

        gradient = (((D - T.T).dot(W).dot(self.X_train))/float(m) +
                                                        theta.dot(L))

        hessian = np.zeros((p*(n+1), p*(n+1)), dtype = np.float64)
        H_uk = np.zeros((p,p), dtype = np.float64)

        for u in range(p):
            for k in range(p):
                S_uk = self.generate_S_uk(theta, u, k);
                H_uk = ((self.X_train.T).dot(S_uk).dot(W).dot(
                                            self.X_train))/float(m)

                if (u == k):
                    H_uk += L

                hessian[u*(n+1): (u+1)*(n+1), k*(n+1):(k+1)*(n+1)] = H_uk

        return [cost, gradient, hessian]
```

```
    def stoch_Grad(self, x, i, theta):
        m = self.get_m()
        n = self.get_n()
        p = self.get_p()
        T = self.generate_T()

        modified_Grad = np.zeros((p,n+1), dtype = np.float64)

        for j in range(p):
            for k in range(n+1):
                temp = (self.a_first_deriv(np.dot(
                    self.X_train[i,::], theta.T).T, j)*self.X_train[i,k]-
                                                    self.X_train[i,k]*T[i,j])
                if (self.weight_flag == 1):
                    temp = temp * self.weight_bell_func(x, i)

                modified_Grad[j,k] = (temp/float(m) +
                                        self.regu_lambda*theta[j,k])

        return modified_Grad


# THE FOLLOWING METHODS ARE HELPERS FOR THE VARIOUS OPTIMIZATION
# ALGORITHMS INCLUDING BATCH GRADIENT DESCENT (BGD), STOCHASTIC
# GRADIENT DESCENT (SGD) AND NEWTON RAPHSON (NR):
# ------initialize_Theta(): Sets theta to the (p by(n+1)) zero matrix.
# ---------------------------------------------------------------------
# ------vectorize(matrix): Takes a matrix, stacks all its rows in column
# -------------------------- fashion and outputs the resulting vector.
# ---------------------------------------------------------------------
# ------matricize(vector, mat_rows): Takes a vector and the number of
# -------------------------- rows for the matrix to be created. It then
# -------------------------- transforms the vector into a corresponding
# -------------------------- matrix (dimensionality-allowing).
# ---------------------------------------------------------------------

    def initialize_Theta(self):
        n = self.get_n()
        p = self.get_p()

        return np.zeros((p,n+1), dtype = np.float64)


    def vectorize(self, matrix):
        return matrix.flatten().T


    def matricize(self, vector, mat_rows):
        if ((vector.shape[0] % mat_rows) == 0):
            return np.reshape(vector, (mat_rows, vector.shape[0]/mat_rows))

        else:
            print "Matrix dimensions not compatible with vector length."
            print "System exiting now"
            sys.exit()


# WE IMPLEMENT NUMERICAL OPTIMIZATION ALGORITHMS (BGD, SGD, NR):
# ------BGD(x, p, n, R, theta): Runs one iteration of BGD.
# -------------------------- R is A (p*(n+1) by p*(n+1)) diag matrix
# --------------------------   equal to the learning rate "alpha"
# --------------------------   multiplied by the identity matrix.
# -------------------------- theta is the current theta to be updated.
# -------------------------- x is the base point being fed to the
# --------------------------   predictor (it only matters in the case of
# --------------------------   weighted regression or classifiction).
# -------------------------- n is the number of input features.
# -------------------------- p is the sufficient statistic's dimension.
# -------------------------- The method returns 4 values including:
# -------------------------- 1) Updated theta after a single iteration.
# -------------------------- 2) Value of the cost function evaluated at
# --------------------------    the updated theta.
# -------------------------- 3) Value of the gradient evaluated at the
# --------------------------    updated theta.
# -------------------------- 4) Value of the hessian evaluated at the
# --------------------------    updated theta.
# ---------------------------------------------------------------------
# ------SGD(x, p, n, m, R, theta): Runs one iteration of SGD. Inputs and
# -------------------------- outputs carry the same meaning as BGD. Each
# -------------------------- iteration runs through m sub-iterations.
# ---------------------------------------------------------------------
```

```
# ------NR(x, p, n, theta): Runs one iteration of NR. The learning rate
# ---------------------------- here is automatically calculated and
# ---------------------------- dynamically  changed for optimality. Except
# ---------------------------- for R, all inputs / outputs are as before.
# ------------------------------------------------------------------------

    def BGD(self, x, p, n, R, theta):
        gradient = self.cost_Grad_Hess(theta, x)[1]
        theta -= np.dot(R, gradient)

        [cost, gradient, hessian] = self.cost_Grad_Hess(theta, x)

        return theta, cost, gradient, hessian


    def SGD(self, x, p, n, m, R, theta):
        for i in range(m):
            theta = theta - np.dot(R, self.stoch_Grad(x, i, theta))

        [cost, gradient, hessian] = self.cost_Grad_Hess(theta, x)

        return theta, cost, gradient, hessian


    '''NOTE ON USING NR FOR MULTINOMIAL CASE:
    There are convergence problems whenever Newton's method is used on a
    multi-class regression problem with more than 2 classes and is hence
    not recommended. A modified Hessian could be used as articulated in
    http://people.stat.sc.edu/gregorkb/Tutorials/MultLogReg_Algs.pdf
    '''
    def NR(self, x, p, n, theta):
        vectorized_theta = self.vectorize(theta)
        [gradient, hessian] = self.cost_Grad_Hess(theta, x)[1:3]
        vectorized_gradient = self.vectorize(gradient)

        theta = self.matricize(vectorized_theta - np.dot(
                        np.linalg.pinv(hessian), vectorized_gradient), p)

        [cost, gradient, hessian] = self.cost_Grad_Hess(theta, x)

        return theta, cost, gradient, hessian


# THE 3 METHODS BELOW COMPUTE THE OPTIMAL THETA AND CONDUCT PREDICTIONS.
# ------find_Optimal_Theta(algorithm, x, iter_max, alpha = 0.01):
# ---------------------------- The algorithm is a string indicating "BGD",
# ----------------------------    "SGD", or "NR".
# ---------------------------- x is the input on which prediction will be
# ----------------------------    conducted. It is encoded as a column
# ----------------------------    vector. Note that theta's dependence on
# ----------------------------    x is only applicable in the case of a
# ----------------------------    weighted learning.
# ---------------------------- iter_max is the max number of iterations
# ----------------------------    for the algorithm.
# ---------------------------- alpha is the learning rate (relevant to
# ----------------------------    BGD and SGD only).
# ---------------------------- The method returns 2 values:
# ---------------------------- 1) The updated theta returned by the
# ----------------------------    algorithm after iter_max iterations.
# ---------------------------- 2) A list containing the value of
# ----------------------------    the cost function after each
# ----------------------------    iteration (useful  in case we want
# ----------------------------    to plot the cost value vs. iterations
# ----------------------------    and check for convergence).
# ------------------------------------------------------------------------
# ------predict(x, theta): Applies the corresponding predictor based
# ---------------------------- on the chosen probabilistic model x is the
# ---------------------------- input point represented as a column vector.
# ------------------------------------------------------------------------
# ------fit(algorithm, X_predict, iter_max, alpha = 0.01):
# ---------------------------- The algorithm is a string indicating "BGD",
# ----------------------------    "SGD", or "NR".
# ---------------------------- X_predicted is the input matrix (can be a
# ----------------------------    vector in case of single input or a
# ----------------------------    matrix in case prediction is conducted
# ----------------------------    on many input vectors. X_predicted is
# ----------------------------    augmented with the all-1 column and
# ----------------------------    has the same structure as X_train.
```

```
# ---------------------------- The method returns 3 values including:
# ---------------------------- 1) theta_optimal (it is a single value in
# ----------------------------    case of non-weighted learning, or a
# ----------------------------    list of values corresponding to each
# ----------------------------    input in case of weighted learning.
# ---------------------------- 2) A list containing the value of the cost
# ----------------------------    function after each iteration until
# ----------------------------    theta_optimal is calculated (note that
# ----------------------------    in case of non-weighted learning, we
# ----------------------------    get a single list of cost values. But
# ----------------------------    if weighted learning is applied, each
# ----------------------------    input will have a different
# ----------------------------    theta_optimal and hence get an
# ----------------------------    associated list of cost values).
# ---------------------------- 3) An array of the predicted values
# ----------------------------    corresponding to each input.
# ------------------------------------------------------------------------
```

```
    def find_Optimal_Theta(self, algorithm, x, iter_max, alpha = 0.01):
        p = self.get_p()
        n = self.get_n()
        m = self.get_m()
        R = np.eye(p,p, dtype = np.float64) * alpha;

        theta = self.initialize_Theta()

        cost = [0] * iter_max
        iter_count = 0

        if (algorithm == "BGD"):
            while(iter_count < iter_max):
                theta, cost[iter_count] = self.BGD(x, p, n, R,
                                                   theta)[0:2]
                iter_count += 1

        elif (algorithm == "SGD"):
            while(iter_count < iter_max):
                theta, cost[iter_count] = self.SGD(x, p, n, m, R,
                                                   theta)[0:2]
                iter_count += 1

        elif (algorithm == "NR"):
            while(iter_count < iter_max):
                theta, cost[iter_count] = self.NR(x, p, n, theta)[0:2]
                iter_count += 1

        else:
            print "No valid model has been inserted. System exiting now"
            sys.exit()

        return theta, cost


    def predict(self, x, theta):
        p = self.get_p()
        predicted = np.zeros((p,1),dtype = np.float64)

        for i in range(p):
            predicted[i,0] = self.a_first_deriv(np.dot(theta,x), i)

        if (self.reg_model in ["normal", "poisson", "geometric"]):
            return predicted[0,0];

        elif (self.reg_model == "binomial"):
            return predicted[0,0] > 0.5

        elif (self.reg_model == "multinomial"):
            max_val = predicted.max();
            sum_val = predicted.sum();
            max_index = np.argmax(predicted)

            if (max_val < (1-sum_val)):
                return p

            else:
                return max_index
```

```
def fit(self,algorithm, X_predict, iter_max, alpha = 0.01):
    l = X_predict.shape[0]
    Y_predict = np.zeros((l, 1), dtype = np.float64)

    if (self.weight_flag == 0):
        theta_optimal, cost = self.find_Optimal_Theta(
                    algorithm, X_predict.T[::,0], iter_max, alpha)

        for element in range(l):
            Y_predict[element,0] = self.predict(
                            X_predict.T[::,element], theta_optimal)

        return theta_optimal, cost, Y_predict

    elif (self.weight_flag == 1):
        theta_optimal = []
        cost = []

        for element in range(l):
            theta_new, cost_new = self.find_Optimal_Theta(
                algorithm, X_predict.T[::,element], iter_max, alpha)

            theta_optimal.append(theta_new);
            cost.append(cost_new)

            Y_predict[element,0] = self.predict(
                    X_predict.T[::,element], theta_optimal[element])

        return theta_optimal, cost, Y_predict
```

References

[1] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[2] Karl Gregory. Multinomial logistic regression algorithms, 2018.

[3] Bent Jorgensen. Exponential dispersion models. *Journal of the Royal Statistical Society*, 49(2):127–162, 1987.