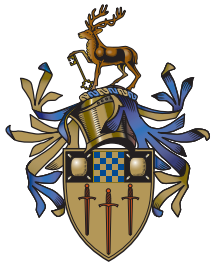
The MM Principle with a special case called the Expectation Maximization algorithm and its application

Ivelina Mladenova

BSc Thesis submitted to the University of Surrey

Supervisor: Dr Naratip Santitissadeekorn

Department of Mathematics
University of Surrey
Guildford GU2 7XH, United Kingdom



E-mail: im00321@surrey.ac.uk

Abstract

In many Statistics problems it is common to optimise a function such as the likelihood or the sum of squares. This report studies the MM method and the special case, the EM method for building surrogate functions through the use of inequalities. The MM stands for *majorise-minimisation* in a a minimisation problem, and *minorise-maximisation* in a maximisation problem. Meaning the surrogate function *minorises* the objective function and is maximised instead of the objective function. Maximising the surrogate function also maximises the objective function due to the descent property of the MM method.

The use of the MM method can be advantageous in optimisation problems because it can separate model parameters by turning products of variables into sums of variables; it can avoid matrix inversion for large datasets; it can turn non-differentiable problems into smooth problems. One notable advantage of the MM method its numerical stability which is guaranteed by the descent property. However, the MM method's biggest drawback is the price of iteration and slow convergent rate.

I explore the MM method through several examples where it is applied to both simple and complex problems. The simple examples in chapter 2 are finding the sample median, minimising the ordinary least squares and the least absolute deviation. Chapter 2 also contains the derivation of the EM algorithm for Gaussian Mixture models. Chapter 3 shows an application of the MM method to a complex problem. It was used to estimate the parameters of a self-exciting point process called the Hawkes process. It investigates a discrete case of the Hawkes process used with count data with no time stamps. The model was tested on COVID-19 data, specifically the average new COVID-19 cases per week in the UK between September 2020 and May 2021.

The examples all show the numerical stability of the MM method, and all the objective functions decreased with each iteration as expected due to the decent property. However, the algorithms required many iterations, although simple iterations to compute. The one dimensional model of the Hawkes process did not provide a good fit for the data, but further investigation can be done to model the data with a multi-dimensional Hawkes process.

With the growth of computational power, I believe the MM and EM methods can be a practical approach for optimisation problems.

Keywords and AMS Classification Codes: MM Principle, EM Algorithm, Constrained Optimisation, Majorization, Minorization,

Contents

1 Motivation

In Statistics lectures, we often get a simple linear model for finding the parameters using techniques like the Ordinary Least Squares (OLS) or the Maximum Likelihood Estimation (MLE). However, in the real world, models become far more sophisticated, and we need numerical methods to help find the optimal model parameters. Some well-known numerical methods to find the global optima include Newton-Raphson method [1], Gradient Descent [2] and Quasi-Newton methods [3].

This report looks at a slightly different parameter estimation method called the Maximization-Minimization (MM) method and its special case, the Expectation-Maximization (EM) algorithm. Kenneth Lange has been exploring the topic of convex optimisation using the MM method and the EM algorithm for over 30 years. In his books [4], [5], he mentions that the MM principle first appeared in the numerical analysis text by Ortega and Rheinboldt in 1970 [6], in the context of iterative solutions for systems of non-linear equations. There were two publications in 1977 contributing to our understanding of the EM algorithm and its applications. One publication was written by Dempster AP, Laird NM and Rubin DB [7] called the *Maximum likelihood from incomplete data via the EM algorithm* built on the MM theory by suggesting a two-step algorithm called the EM algorithm to compute Maximum Likelihood Estimates where there is "incomplete", or latent data. The EM algorithm is better known in Statistics, especially an application of the EM algorithm for finding optimal parameters for Gaussian Mixture Models in probabilistic classification problems.

In Chapter 2, I describe the idea behind the MM method, the rules the method obeys, and the properties it holds. I also discuss ways of constructing surrogate functions that one can optimise instead of the original objective function. To illustrate the way of using some of the inequalities for building surrogate functions, I explore several simple examples using the MM method for solving common problems such as finding the sample median, estimating parameters of linear models where one needs to optimise the functions: the ordinary least squares (OLS) and least absolute deviation (LAD). I also describe the EM method for building surrogate functions, which introduces the notion of latent or missing variables that are used to make the surrogate function easier

to maximise, or minimise. I study the derivation of a famous application of the EM algorithm: an unsupervised learning method for finding optimal parameters of a Gaussian mixture model.

Chapter 3 applies the MM method on a more complex model, which is a discrete case of the Hawkes process, suggested by my supervisor. The chapter starts with a general introduction to the Hawkes process. Then I present the discrete model of the Hawkes process with three parameters. The MM method is used to build a surrogate function that *majorises* the objective function. Due to the properties of the MM method, minimising the surrogate function results in the minimisation of the model's negative log-likelihood. By approximating one of the parameters to zero, the surrogate function is built such that one can separate the model parameters and find an update function for each one in an iterative way. Using Python, I generated synthetic data and analysed the behaviour of the model parameters using Principle Component Analysis (PCA). That helps to understand the relationship between parameters which made it easier to analyse the results from running the update functions. My supervisor suggested to examine the expectation and variance of the weak stationary, discrete Hawkes process as the time step go to infinity, and we see that they both converge to a constant value for different parameter values.

I build the functions that optimise the model parameters in Python and test them on synthetic data, with 5,000 data points. The iterative method works, and the negative log-likelihood decreases with each iteration. However, I also come across some drawbacks of the functions. For example having null data values throws off the parameter updates and the iterations stop. Another problem is that one parameter has an unbounded update function, making the method harder to test with different initial parameter values. The update functions in Python are then applied to the daily, new COVID-19 cases by Specimen date in the UK, averaged across the week, for 35 weeks. The data on new COVID-19 cases starts from the 9th of September, the day that children go back to school. The first run results in the decay rate parameter going beyond its bounded region of (0,1), which sends the rest of the parameters to zero, and the negative log-likelihood proliferates. After a few runs with different initial parameter values, the iterations stabilise and the benefits of the MM method are confirmed - the negative log-likelihood begins to decrease with each iteration.

In summary, in this report, I explore the MM method for parameter optimisation and highlight the benefits of using it by studying several simple examples, and exploring a more complex one. The code for the analysis is contained in the Appendix. However, I have also uploaded it on my GitHub account for ease of use and proofreading, which can be accessed here: *https://github.com/Ivelina0/Dissertation_code*.

2

Introduction to the MM and the EM methods

2.1. Introduction to the MM principle

The MM principle is not a concrete algorithm but an optimisation method that relies heavily on convexity arguments [4]. It is also not a replacement for the other iterative methods but can be more efficient when used correctly. The MM method is advantageous in high-dimensional problems and is known to easily deal with equality and inequality constraints; as well as turning non-differentiable problems into smooth problems. It is also quite useful in separating parameters in optimisation problems, and an example of this is shown in the next chapter.

The MM method is based on the notion of tangent majorization. Given a function $f(\theta)$ where the θ is the parameter vector that we want to optimise, we can construct a surrogate function $g(\theta|\theta^{(n)})$ which is said to majorise (or minimise) the objective function at θ_n provided that the following conditions are met:

$$f(\theta^{(n)}) = g(\theta^{(n)}|\theta^{(n)}), \tag{2.1}$$

$$f(\theta) \leqslant g(\theta|\theta^{(n)}),\ \theta \neq \theta^{(n)}. \tag{2.2}$$

The first condition 2.1 is the **tangency condition** and the second 2.2 - the **domination condition**. This means that in a minimisation problem the surrogate function lies above the surface $f(\theta)$ and is tangent to it at the point $\theta = \theta^{(n)}$. Note that n represents the number of iterates in a search for the optima. Fig. 2.2 shows a simple one-dimensional example of how a surrogate function minimises a piecewise linear function f(x) and the $cos(x)$ function for a domain defined as $[0, 2\pi]$.

The MM method lets us minimise the objective function $f(\theta)$ by constructing a majorising function $g(\theta|\theta^{(n)})$ and minimising it to produce the next iterate $\theta^{(n+1)}$. The MM algorithm drives the $f(\theta)$ downhill if we define a

minimum $\theta^{(N)}$ for some $N \geqslant n$, then using the inequality $g(\theta^{(N)}|\theta^{(n)}) \leqslant g(\theta^{(N-1)}|\theta^{(n)})$ gives the following result

$$\begin{aligned} f(\theta^{(N)}) &= g(\theta^{(N)}|\theta^{(n)}) + f(\theta^{(N)}) - g(\theta^{(N)}|\theta^{(n)}) \\ &\leqslant g(\theta^{(N-1)}|\theta^{(n)}) + f(\theta^{(N-1)}) - g(\theta^{(N-1)}|\theta^{(n)}) \\ &= f(\theta^{(N-1)}) \; for \; any \; N > n. \end{aligned} \tag{2.3}$$

Equation 2.3 shows the **decent property** of the MM algorithm that forces $f(\theta)$ downhill, depends only on the decreasing the surrogate function and not on minimising it. The descent property also provides the MM algorithm with renowned numerical stability. Note that when the objective function $f(\theta)$ is convex, we can use existent convex optimisation theories to show that $\theta^{(n)}$ converges to a global minimum of the objective function regardless of the initial point $\theta^{(0)}$. However, it is also important to note that the MM and the EM methods do not guarantee convergence to a global minimum when the objective function is not convex.

If the surrogate function majorises the objective function at an interior point $\theta^{(n)}$ of the domain of $f(\theta)$, then the difference $g(\theta|\theta^{(n)}) - f(\theta)$ is at the stationary point $\theta^{(n)}$ and the following identity holds,

$$g'(\theta^{(n)}|\theta^{(n)}) = f'(\theta^{(n)}). \tag{2.4}$$

In other words, the equality 2.4 means that the stationary point minorises both functions and therefore the tangent lines at that point must be the equal. To add to that, the second derivative of the inequality $g''(\theta^{(n)}|\theta^{(n)}) - f''(\theta^{(n)})$ is positive (semi-definite). The equality shows that the majorisation relation between functions is closed under the formation of sums, non-negative products, limits and composition with an monotonically increasing function.

Similarly, the MM method is used to solve a maximization problem by maximising a surrogate function $g(\theta|\theta^{(n)})$ that minorises the objective function $f(\theta)$.

2.2. Methods for constructing surrogate functions

This section goes through six methods for constructing majorising functions. The first two methods are built on the definition of convex functions. So we start by defining a convex function.

Definition 2.1. *(Convex function)*
A real-valued, continuous function $f(\theta)$ defined on an interval $[a, b]$ is convex provided that for any two points $\theta_1, \theta_2 \in [a, b]$ and any α, it meets the following:

$$f[\alpha\theta_1 + (1-\alpha)\theta_2] \leqslant \alpha f(\theta_1) + (1-\alpha) f(\theta_2) \; for \; \alpha \; \in (0, 1). \tag{2.5}$$

If the inequality is strictly less than $(<)$, for all θ_1 and θ_2, then $f(\theta)$ is called **strictly convex**.

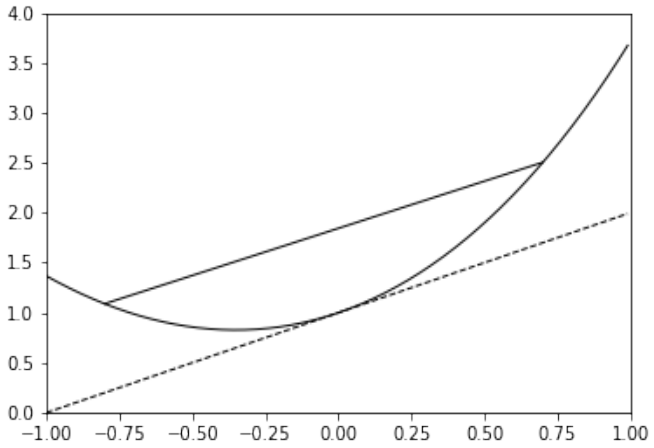


Figure 2.1: Convex function $f(\theta) = e^{\theta} + \theta^2$

By induction, one can prove that the inequality 2.5 holds for all θ_i and hence defining a convex function $f(\theta)$ such that:

$$f\left(\sum_{i}^{n} \alpha_i \theta_i\right) \leqslant \sum_{i}^{n} \alpha_i f(\theta_i), \tag{2.6}$$

where $\alpha \in (0, 1)$ and $\sum_i^m \alpha_i = 1$. The inequality 2.6 is called the **Jensen inequality**. Similarly, a concave function satisfies the reverse of the Jensen inequality. The Jensen inequality is crucial in majorising functions of the form $f[\alpha_1(\theta_1) + \alpha_2(\theta_2)]$, where the a_i for $i = 1, 2$ and the functions are positive for some underlying parameters θ_1, θ_2 [5]. Intuitively, the upper bound in Jensen's inequality is illustrated by the chord above the curve in fig. 2.1, where the chord represents the function evaluated at each parameter $\theta_i \in [a, b]$ multiplied by the weight α_i, whereas the curve below the chord is just the function evaluated at each $\alpha_i \theta_i$. Also note how the tangent line to the convex function lies below the function, and minimises it.

Following from the Jensen inequality, we derive the first three majorization methods. Suppose we have vectors $\mathbf{c}$, $\boldsymbol{\gamma}$ and $\boldsymbol{\gamma}^{(n)}$ where the components for each vector are positive, and $\boldsymbol{c}, \boldsymbol{\gamma}^{(n)}, \boldsymbol{\gamma} \in \mathbb{R}^m$. Using Jensen's inequality, we can create a convex function given the simple linear function $\mathbf{c}^{\top}\boldsymbol{\gamma}$, by taking $\theta_i = \boldsymbol{c}^{T}\boldsymbol{\gamma}^{(n)}\ \gamma_i/\gamma_i^{(n)}$ and $\alpha_i = c_i\gamma_i^{(n)}/\boldsymbol{c}^{T}\boldsymbol{\gamma}$ to build the following convex surrogate function:

$$f(\boldsymbol{c}^{T}\boldsymbol{\gamma}) \leqslant \sum_{i}^{m} \frac{c_i\gamma_i^{(n)}}{\boldsymbol{c}^{T}\boldsymbol{\gamma}^{(n)}} f\left(\boldsymbol{c}^{T}\boldsymbol{\gamma}^{(n)} \frac{\gamma_i}{\gamma_i^{(n)}}\right) = g(\boldsymbol{\gamma}|\boldsymbol{\gamma}^{(n)}). \tag{2.7}$$

Note that the surrogate function $g(\boldsymbol{\gamma}|\boldsymbol{\gamma}^{(n)})$ equals $f(\boldsymbol{c}^{T}\boldsymbol{\gamma})$ when $\boldsymbol{\gamma} = \boldsymbol{\gamma}^{(n)}$. The inequality in 2.7 is useful in constructing high-dimensional optimisation problems because it separates the parameters in the surrogate function therefore reduces optimisation over $\boldsymbol{\gamma}$ to a sequence of one dimensional optimizations over each component γ_i.

The inequality would still hold if the parameter vector $\boldsymbol{\theta}$ is replaced by a vector-valued function $u(\gamma)$ of γ.

A further extension of equation 2.7 that relaxes the positivity restrictions on the vectors $\mathbf{c}$, $\boldsymbol{\gamma}$ and $\boldsymbol{\gamma}^{(n)}$, is the following:

$$f(\boldsymbol{c}^T\boldsymbol{\gamma}) \leqslant \sum_i^m \alpha_i f\left(\frac{c_i}{\alpha_i}(\gamma_i - \gamma_i^{(n)}) + \boldsymbol{c}^T\boldsymbol{\gamma}^{(n)}\right) = g(\boldsymbol{\gamma}|\boldsymbol{\gamma}^{(n)}) \tag{2.8}$$

where $\alpha_i \geqslant 0$ $(i = 1, ..., m)$, $\sum_{i=1}^m \alpha_i = 1$, and $\alpha_i > 0$ whenever $c_i \neq 0$. The inequality was suggested by [8] to be used in medical imaging context, in 1995. In this example we can pick α_i as:

$$\alpha_i = \frac{|c_i|^p}{\sum_j |c_j|^p} \; for \; p \geqslant 0. \tag{2.9}$$

When $p = 0$ and $c_i = 0$ then $\alpha_i = 0$, and if $p = 0$ but c_i is a non-zero coefficient, where there are q non-zero coefficients, then represent $\alpha_i = 1/q$.

The third method uses the arithmetic-geometric mean inequality and is a special case of the Jensen inequality 2.6 when $\alpha_i = 1/m$ with the convex function e^θ. Suppose we take $f(\theta) = e^\theta$ and the weight $\alpha_i = 1/m$ then

$$\begin{aligned} exp\left(\frac{1}{m}\sum_{i=1}^m \theta_i\right) &\leqslant \frac{1}{m}\sum_{i=1}^m exp(\theta_i) \\ \left(\prod_{i=1}^m y_i\right)^{1/m} &\leqslant \frac{1}{m}\sum_{i=1}^m y_i \end{aligned} \tag{2.10}$$

where we can generalise the inequality by letting $y_i = e^{\theta_i}$, to obtain the standard form of the arithmetic-geometric mean inequality. Since the exponential function is strictly convex, the equality holds if and only of all y_i coincide.

In fact, the inequality is reversed for the log-function which is strictly concave, shown by

$$ln\sum_{i=1}^m \alpha_i\theta_i \geqslant \sum_{i=1}^m \alpha_i ln(\theta_i), \tag{2.11}$$

for constants $\alpha_i 0$ with $\sum_{i=1}^m \alpha_i = 1$. This equation is used in the derivation of the Expectation-maximization algorithm (EM) in Section 2.4.

The fourth majorization method uses first-order Taylor series approximation to construct a surrogate function of a differentiable **convex function** $f(\cdot)$ on the domain S. Suppose we have two vectors $\boldsymbol{\theta}$, $\boldsymbol{\theta}^{(n)} \in \mathbb{R}^m$, then we can construct the following linear majorisation function:

$$f(\boldsymbol{\theta}) \geqslant f(\boldsymbol{\theta}^{(n)}) + \nabla f(\boldsymbol{\theta}^{(n)})^T(\boldsymbol{\theta} - \boldsymbol{\theta}^{(n)}) = g(\boldsymbol{\theta}|\boldsymbol{\theta}^{(n)}). \tag{2.12}$$

Equality holds when $\boldsymbol{\theta} = \boldsymbol{\theta}^{(n)}$. If the function is convex on the domain $\mathbb{R}^m$, then the inequality is reversed. The inequality also holds if the vector θ is replaced by a vector-valued function $u(\theta)$. The method is also known as the supporting hyperplane method where the $g(\boldsymbol{\theta}|\boldsymbol{\theta}^{(n)})$ describes the hyperplane through $(\boldsymbol{\theta}^{(n)}, f(\boldsymbol{\theta}^{(n)}))$ with the normal $\nabla f(\boldsymbol{\theta}^{(n)})$.

An extension to this method can be shown by expressing the continuous convex function as the difference of two convex functions, we can use this trick to construct a surrogate function [9]. For example, let $f(\boldsymbol{\theta}) = g(\boldsymbol{\theta}) - h(\boldsymbol{\theta})$ where both $g(\boldsymbol{\theta})$ and $h(\boldsymbol{\theta})$ are convex, using Taylor series again gives

$$f(\boldsymbol{\theta}) \leqslant g(\boldsymbol{\theta}) - h(\boldsymbol{\theta}^{(n)}) - \nabla h(\boldsymbol{\theta}^{(n)})^T(\boldsymbol{\theta} - \boldsymbol{\theta}^{(n)}).$$

The differences of convex functions is a useful strategy in convex optimisation and has many applications in machine learning such as kernel selection [10].

In order to define the fifth method, we need to use the following proposition from [5].

Proposition 2.2. *Let $f(\boldsymbol{\theta})$ be a continuous, twice-differentiable function on the open convex set $S \in \mathbb{R}^n$. If its Hessian matrix $H(\boldsymbol{\theta}) = d^2 f(\boldsymbol{\theta})/d\theta^2$ is positive semi-definite for all $\boldsymbol{\theta}$, then $f(\boldsymbol{\theta})$ is convex. It is strictly convex if $H(\boldsymbol{\theta})$ is positive definite for all $\boldsymbol{\theta}$.*

Proof: Using the second-order Taylor expansion

$$\begin{aligned} f(\boldsymbol{y}) = f(\boldsymbol{\theta}) &+ \frac{df}{d\boldsymbol{\theta}}(\boldsymbol{\theta})(\boldsymbol{y} - \boldsymbol{\theta}) \\ &+ (\boldsymbol{y} - \boldsymbol{\theta})^* \int_0^1 \frac{d^2}{d\boldsymbol{\theta}^2} f\Big(\boldsymbol{\theta} + t(\boldsymbol{y} - \boldsymbol{\theta})\Big)(1-t)dt(\boldsymbol{y} - \boldsymbol{\theta}) \end{aligned}$$

for $\boldsymbol{y} \neq \boldsymbol{\theta}$ demonstrates that

$$f(\boldsymbol{y}) \geqslant f(\boldsymbol{\theta}) + \frac{df}{d\boldsymbol{\theta}}(\boldsymbol{\theta})(\boldsymbol{y} - \boldsymbol{\theta}).$$

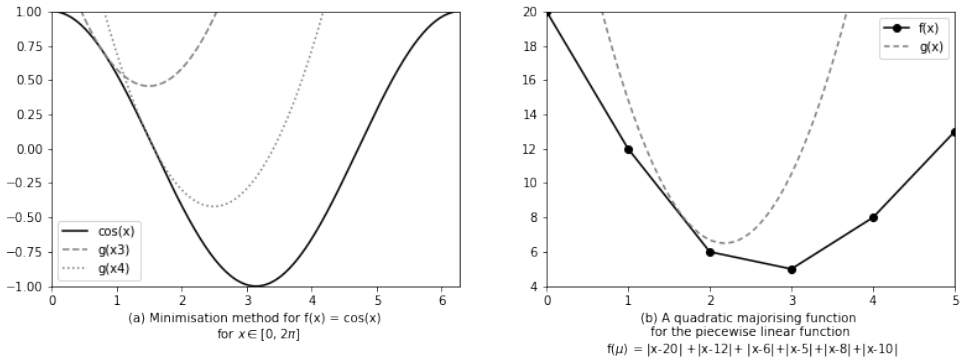
The inequality is strict when $d^2 f(\boldsymbol{\theta})/d\boldsymbol{\theta}^2$ is positive definite. □

The fifth method also uses the 2.2 and the quadratic upper bound principle [11]. Suppose a convex, quadratic function $f(\boldsymbol{\theta}) = 1/2\ \boldsymbol{\theta}^T A\boldsymbol{\theta} + b^T\boldsymbol{\theta} + c$ is twice differentiable, $\boldsymbol{y} = \boldsymbol{\theta}^{(n)}$ and there exists a positive definite matrix $\mathbf{B}$ satisfying $\mathbf{B} \geqslant d^2 f(\boldsymbol{\theta})/d\boldsymbol{\theta}^2$ and $\mathbf{B} \geqslant 0$ where $\mathbf{B} - d^2 f(\boldsymbol{\theta})/d\boldsymbol{\theta}^2$ is also positive definite for all $\boldsymbol{\theta}$. Then using the second-order Taylor series approximation of $f(\boldsymbol{\theta})$ to construct the following:

$$\begin{aligned} f(\boldsymbol{\theta}) &= f(\boldsymbol{\theta}^{(n)}) + \frac{df}{d\boldsymbol{\theta}}(\boldsymbol{\theta}^{(n)})(\boldsymbol{\theta} - \boldsymbol{\theta}^{(n)}) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^{(n)})^T \frac{d^2 f(\boldsymbol{\alpha})}{d\boldsymbol{\theta}^2}(\boldsymbol{\theta} - \boldsymbol{\theta}^{(n)}) \\ &\leqslant f(\boldsymbol{\theta}^{(n)}) + \frac{df}{d\boldsymbol{\theta}}(\boldsymbol{\theta}^{(n)})(\boldsymbol{\theta} - \boldsymbol{\theta}^{(n)}) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}^{(n)})^T \mathbf{B}(\boldsymbol{\theta} - \boldsymbol{\theta}^{(n)}) \\ &= g(\boldsymbol{\theta}|\boldsymbol{\theta}^{(n)}). \end{aligned} \tag{2.13}$$

Here $\boldsymbol{\alpha}$ is an intermediate point on the line segment from $\boldsymbol{\theta}$ to $\boldsymbol{\theta}^{(n)}$. This method is often used to construct a quadratic surrogate function that can avoid the inversion of the Hessian matrix in Newton's method which can be computationally expensive.

The sixth and final majorization method is derived from the Cauchy-Schwartz inequality for the Euclidean norm. Suppose there exists a convex function $f(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|$ that satisfies the triangle inequality and the homogeneous condition $\|\nu\boldsymbol{\theta}\| = |\nu| \cdot \|\boldsymbol{\theta}\|$, then we can re-write the function as $f(\boldsymbol{\theta}) = \sqrt{\sum_i \theta_i^2}$, take the derivative



(a) Minimisation method for f(x) = cos(x) for $x \in [0, 2\pi]$

(b) A quadratic majorising function for the piecewise linear function f(μ) = |x-20| + |x-12| + |x-6| + |x-5| + |x-8| + |x-10|

Figure 2.2: Simple examples

$\nabla f(\boldsymbol{\theta}) = |\boldsymbol{\theta}|/\|\boldsymbol{\theta}\|$ and use equation 2.12 to get the following Cauchy-Schwartz inequality

$$\begin{aligned} \|\boldsymbol{\theta}\| &\geqslant \|\boldsymbol{\theta}^{(n)}\| + \frac{(\boldsymbol{\theta} - \boldsymbol{\theta}^{(n)})^T \boldsymbol{\theta}^{(n)}}{\|\boldsymbol{\theta}^{(n)}\|} \\ &= \frac{\boldsymbol{\theta}^T \boldsymbol{\theta}^{(n)}}{\|\boldsymbol{\theta}^{(n)}\|} \\ &= g\big(\boldsymbol{\theta}|\boldsymbol{\theta}^{(n)}\big), \end{aligned} \tag{2.14}$$

using the fact that $\|\boldsymbol{\theta}\| = \boldsymbol{\theta}^T\boldsymbol{\theta}/\|\boldsymbol{\theta}\|$ to reduce the surrogate function The Cauchy-Schwartz inequality was used in deriving MM algorithms for multi-dimensional scaling by [12], [13].

2.3. Applications of the MM method

2.3.1. Calculation of sample median

To understand how these inequalities work, I study a few simple and well understood examples. The first example is a simple one-dimensional example from [14], [15]. Suppose we need to calculate the sample median $\theta_{1/2}$ of a random variable X. In fact, it is known that the median minimises the expectation $f(\theta) = E(|X - \theta|)$ by [16]. See Appendix A.1 for proof.

This example shows an iteratively reweighted least squares method for finding the sample median and sample quantiles of m real numbers $x_1, \ldots, x_m$ without using a sorting algorithm. Therefore we minimize the non-differentiable objective function

$$f(\theta_{1/2}) = \sum_{i=1}^{m} |x_i - \theta_{1/2}|. \tag{2.15}$$

By constructing a quadratic function for a single point using the arithmetic-geometric mean inequality defined

in 2.10, where $y_i = (x_i - \theta_{1/2})^2$ and $1/m = 1/2$ gives

$$v_i(\theta_{1/2}|\theta_{1/2}^{(n)}) = \frac{(x_i - \theta_{1/2})^2}{2|x_i - \theta_{1/2}^{(n)}|} + \frac{1}{2}|x_i - \theta_{1/2}^{(n)}|. \tag{2.16}$$

where the surrogate majorises the $|x_i - \theta_{1/2}|$ at the point $\theta_{1/2}^{(n)}$. Hence, the full surrogate function is defined as $g(\theta_{1/2}|\theta_{1/2}^{(n)}) = \sum_{i=1}^{m} v_i(\theta_{1/2}|\theta_{1/2}^{(n)})$ which majorises the objective function $f(\theta_{1/2})$.

Note also that $\theta_{1/2}^{(n)}$ is the nth iterate of the sample median. Differentiating $g(\theta_{1/2}|\theta_{1/2}^{(n)})$ with respect to $\theta_{1/2}$ and set it to 0 gives

$$\frac{d}{d\theta_{1/2}} g(\theta_{1/2}|\theta_{1/2}^{(n)}) = \sum_{i=1}^{m} \frac{1}{2} \frac{(-2)(x_i - \theta_{1/2})}{|x_i - \theta_{1/2}^{(n)}|} = 0.$$

$$\sum_{i=1}^{m} \frac{-x_i}{|x_i - \theta_{1/2}^{(n)}|} + \sum_{i=1}^{m} \frac{\theta_{1/2}}{|x_i - \theta_{1/2}^{(n)}|} = 0$$

Re-arranging, to get the next iterate

$$\theta_{1/2}^{n+1} = \frac{\sum_{i=1}^{m} \omega_i^{(n)} x_i}{\sum_{i=1}^{m} \omega_i^{(n)}}. \tag{2.17}$$

where $\omega_i^{(n)} = |x_i - \theta_{1/2}^{(n)}|^{-1}$. This algorithm works except if the weight $\omega_i^{(n)} = \infty$ which could occur when $x_i = \theta_{1/2}^{(n)}$.

2.3.2. Calculation of sample quantiles

The iterative sample median algorithm can be generalised and applied to the broader problem of sample quantiles. Assume that $(x_i - \theta_q)$ is well defined for each i and $q \in (0, 1)$, then we can extend this algorithm for an arbitrary sample quantile θ_q. Fig.2.2 shows an example of the function $f(\theta)$ and its majorizer $g(\theta|\theta_{1/2}^{(n)})$ for a sample size of n = 6. Note that a general definition of the *q-th* quantile, e.g.$q = 0.75th$ quantile, of X is given as

$$p(X \leqslant x) \geqslant q \text{ and } p(X \geqslant x) \geqslant 1 - q, \tag{2.18}$$

for some data points $X = x$. Using the result from [17] which states that a sample q quantile θ_q of $x_1, \ldots, x_m$ minimizes the function

$$f(\theta) = \sum_{i=1}^{m} \eta_q(x_i - \theta), \tag{2.19}$$

and if one sets $c = x_i - \theta$ then the following result is shown

$$\eta_q(c) = |c| \left[q \mathbb{1}_{\{c \geqslant 0\}} + (1 - q) \mathbb{1}_{\{c < 0\}} \right] = qc - c \mathbb{1}_{\{c < 0\}}. \tag{2.20}$$

In this case the q-th quantile minimises the objective function and similarly to the sample median example, we can construct a majorising function by defining an unique quadratic curve tangent to the graph of $\eta_q(x_i - \theta)$ at the points $\theta = \pm\theta^{(n)}$, for each i. The function, proposed by [17],

$$\nu_q(x_i - \theta|x_i - \theta_q^{(n)}) = \frac{1}{4} \left[\frac{(x_i - \theta)^2}{|x_i - \theta_q^{(n)}|} + (4q - 2)(x_i - \theta) + |x_i - \theta_q^{(n)}| \right] \tag{2.21}$$

is a majorising function for each data point x_i and summing over all values gives the surrogate function $g(\theta|\theta_q^{(n)})$. Differentiating the surrogate function with respect to θ and setting it to 0 yields the following update rule

$$\theta^{n+1} = \frac{m(2q-1) + \sum_{i=1}^{m} \omega_i^{(n)} x_i}{\sum_{i=1}^{m} \omega_i^{(n)}} \tag{2.22}$$

where the weight $\omega_i^{(n)} = |x_i - \theta_q^{(n)}|^{-1}$ depends on $\theta_q^{(n)}$. Similarly to the sample median example, this algorithm for finding the sample quantile would not work if the weight is undefined, namely when $x_i = \theta_q^{(n)}$. Kenneth Lange and David Hunter, [17], provide a solution for the undefined weight and extends the idea to a broader technique of quantile regression which was suggested by [18]. This is fascinating because it avoids using sorting algorithms and only depends on using arithmetic and iteration.

Example 2.3. *Suppose that you want to find the median value of the following data set* $x = [1, 3, 4, 8, 10, 11, 15]$. *In this simple case, the real median of the data is 8. One can find the median iteratively using the MM method with the update rule from 2.17. We get the following results for the median, starting at an initial guess value of* $\theta = 6$.

Iteration no.	$\theta^{(n)}$	$f(\theta^{(n)})$	Iteration no.	$\theta^{(n)}$	$f(\theta^{(n)})$
1	6.4775	29.5225	5	7.9005	28.0995
2	6.9419	29.0581	6	7.9867	28.0133
3	7.3578	28.6422	7	7.9997	28.0003
4	7.6883	28.3117	8	7.9999	28.0

Table 2.1: The objective function with each iteration

Table 2.1 shows that after 9 iterations, the value of $\theta^{(9)}$ *converges to 8 which is the real median of the data set. Notice also that the objective function is stable and decreasing. The advantages of using this method in order to find the median of a data set are: the simplicity, the ease of implementation and the numerical stability of the objective function [14].*

In the paper Hunter compares the MM algorithm to the interior point method. The conclusion from the paper is that the MM method is computationally competitive with the interior point method on non-linear quantile regression problems. On the other hand, the MM algorithm does not converge as fast given a large number of parameters and it does not always converge to the global minima. This method can sometimes converge to the local minima and so it is advised to run the MM algorithm several times starting at different initial values of θ.

2.3.3. Linear regression with Ordinary Least Squares (OLS)

Ordinary least squares a very common concept in Statistics, which makes it another simple case to use the MM method on, and get a better understanding of how it works.

The least squares method finds the optimal parameters by minimizing the sum of the squares of the residuals. The regression model is linear if the model includes a linear combination of $n+1$ parameters, where the general form for the multiple linear model is

$$Y_i = \theta_0 + \sum_{j=1}^{n} \theta_j x_{ij} + \epsilon_i, \tag{2.23}$$

where $\epsilon_i \sim N(0, \sigma^2)$ is i.i.d normal random variable, $\theta_0, \theta_1, \cdots, \theta_n$ are the parameters, or coefficients we want to estimate and x_i are the m data points. The outcome Y_i is also a random variable with $E(Y_i) = \theta_0 + \sum_{j=1}^{n} \theta_j x_{ij}$ and $Var(Y_i) = \sigma^2$. The is generally modelled by the equation

$$f(x) = \theta_0 + \sum_{j=1}^{n} \theta_j x_j. \tag{2.24}$$

The parameter estimates for the regression model are fitted using a known estimation method - the least squares method, where if X is a full rank $m \times n+1$ matrix (with 1s in the first column). The columns of the matrix are all linearly independent, $\boldsymbol{\theta}$ is a $n+1 \times 1$ parameter vector, $\mathbf{y}$ is a $m \times 1$ vector of the observed values and there exists a $m \times 1$ error vector $\mathbf{r}$ such that the linear regression model can be rewritten as $\boldsymbol{y} = X\boldsymbol{\theta} + \boldsymbol{r}$.

To find the estimates $\hat{\theta}_i$ for $i = 0, 1, 2, \cdots, n$ for the parameters $\theta_0, \theta_1, \cdots, \theta_n$, we find the coefficients that minimize the residual sum of squares

$$\begin{aligned} RSS(\theta) &= \sum_{i=1}^{m} (r_i)^2 \\ &= \sum_{i=1}^{m} (y_i - f(x_i))^2 \\ &= \sum_{i=1}^{m} (y_i - \sum_{j=0}^{n} \theta_j x_{ij})^2. \end{aligned} \tag{2.25}$$

The residual sum-of-squares in matrix becomes

$$RSS(\theta) = (y - X\theta)^T (y - X\theta) \tag{2.26}$$

Differentiating the RSS with respect to θ and setting the derivative to zero, gives an unique solution $\hat{\theta} = (X^t X)^{-1} X^T y$. However, we often do not get a full-rank matrix X or the number features $n+1$ can exceed the number of training cases, or data points m.

In that case, we can use the MM method to avoid matrix inversion. Since the $RSS(\theta) = \sum_{i=1}^{m} (y_i - \sum_{j=0}^{n} \theta_j x_{ij})^2$

function is convex we majorise it by using the inequality defined in 2.8. The majorisation function is defined as

$$\begin{aligned}\sum_{i=1}^{m}\left(y_i-\sum_{j=0}^{n}\theta_i x_{ij}\right)^2 &\leqslant \sum_{i=1}^{m}\sum_{j=0}^{n}\alpha_{ij}\left[y_i-\frac{x_{ij}}{\alpha_{ij}}(\theta_j-\theta_j^{(n)})-x_i^T\theta^{(n)}\right]^2 \\ &= g(\theta|\theta^{(n)})\end{aligned} \tag{2.27}$$

with equality when $\theta=\theta^{(n)}$ and

$$\alpha_{ij}=\frac{|x_{ij}|^p}{\sum_k|x_{ik}|^p}\ for\ p\in[0,1]. \tag{2.28}$$

To minimize the $g(\theta|\theta^{(n)})$, set $\alpha_{ij}=1$ and differentiate 2.27 with respect to θ_j and set to zero to get

$$\sum_{i=1}^{m}-2x_{ij}\left[y_i-\frac{x_{ij}}{\alpha_{ij}}(\theta_j-\theta_j^{(n)})-x_i^T\theta^{(n)}\right]=0 \tag{2.29}$$

which can be re-arranged to get the update

$$\theta_j^{n+1}=\theta_j^m+\frac{\sum_{i=1}^{m}x_{ij}(y_i-x_i^T\theta^{(n)})}{\sum_{i=1}^{n}\frac{x_{ij}^2}{\alpha_{ij}}}\ for\ p\geqslant 0. \tag{2.30}$$

Choosing a value of p for α_{ij} doesn't always have to be 1. And alternating different values of p as iterations proceed can accelerate convergence. This method is useful when trying to estimate only a few parameters θ as it avoids matrix inversion such as the one in fig. 2.3a.

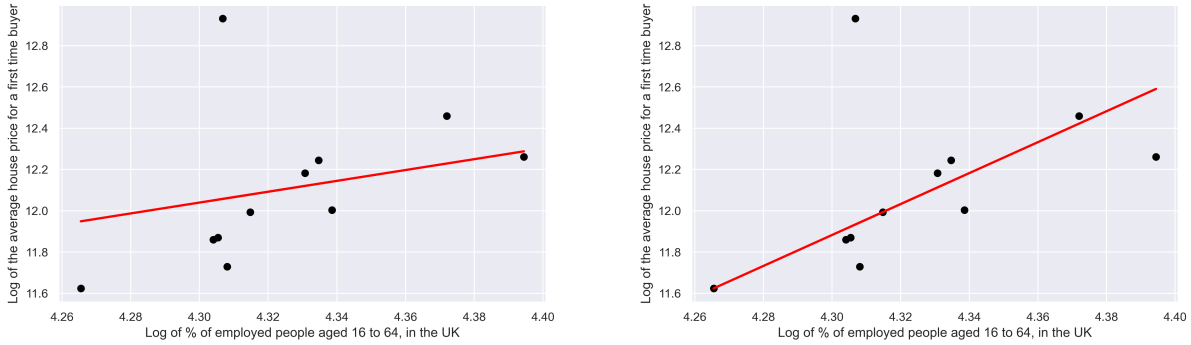
Fig.2.3a and table 2.2 show a simple linear regression example using the MM method. The data consists of the (log of) UK employment percentage by region, plotted against the (log of) the average price of a house for first time buyers by region between June 2019 and December 2019. The parameters converged to $\hat{\theta_0}=0.7264$ and $\hat{\theta_1}=2.6310$ after 48 iterations with initial values $\theta_0^{(0)}=-2,\theta_1^{(0)}=2$. Table 2.2 shows that the algorithm converges after 48 iterations. See Appendix A.6. for code. Data taken from Office for National Statistics - Employment and labour market and Office for National Statistics - People, population and community .

Compared to the MM method, using the Scikit-learn package to fit a linear model to the data gives the parameters $\theta_0=4.7254$ and $\theta_1=-8.3320$. So the algorithm doesn't always converge to the global optima. To fix this, it is advised to run the algorithm several times with different initial values and evaluate and compare the residual sum of squares at the end of each run.

The MM method in the OLS example avoids matrix inversion and instead finds the optimal parameters using an iterative approach to minimising the surrogate function. It also guarantees a decrease/increase of the objective function, in this case the it minimises the sum of square errors. The method is excellent for problems involving just a few parameters instead of using the single- step solution such as matrix inversion, but becomes more computationally expensive when one needs to estimate a large number of parameters. Also, it may not converge faster than other methods such as cyclic coordinate descent [4].

Iteration	$\theta^{(n)}$	$f(\theta^{(n)})$	Iteration	$\theta^{(n)}$	$f(\theta^{(n)})$
1	$[-1.5041, 2.1147]$	220.318	40	$[0.7259, 2.6307]$	1.1457
5	$[-0.2725, 2.3995]$	45.1579	45	$[0.7263, 2.6309]$	1.1457
10	$[0.3608, 2.5460]$	7.0621	46	$[0.7264, 2.6309]$	1.1457
20	$[0.6780, 2.6194]$	1.2526	47	$[0.7264, 2.6309]$	1.1457
30	$[0.7204, 2.6294]$	1.1476	48	$[0.7264, 2.6310]$	1.1457

Table 2.2: MM iterations and the objective function for OLS



(a) OLS Regression line

(b) LAD Regression line

Figure 2.3: Comparing OLS to LAD for fitting data with initial values $\theta_0 = -2, \theta_1 = 2$ and $\theta_0 = 0, \theta_1 = 0$, respectfully.
UK data on employment against the average price of first time home buyers from June 2019 to December 2019, using LAD Regression

The advantage of using the least squares method for finding the optimal model parameters are that it is the most efficient linear regression estimator if the model assumptions such as linearity, constant variance and no outliers in the data hold [19]. However, using the least squares method is not always robust as shown in fig. 2.3a where the data for London seems to be an outlier. Therefore the regression line is shifted upward to minimize the large squared error created by the outlier point.

2.3.4. Linear regression with Least Absolute Deviation (LAD)

One way to make the linear regression method more robust to outliers is instead to use the least absolute error (or deviation). The method replaces the sum of squared errors $\sum_{i=1}^{m}(y_i - x_{ij}^T\theta)^2$ by

$$\nu(\theta) = \sum_{i=1}^{m} |y_i - x_{ij}^T\theta| = \sum_{i=1}^{n} |r_i(\theta)|. \tag{2.31}$$

where the ith residuals $r_i(\theta) = y_i - x_{ij}^T\theta^{(n)}$ are treated equally and larger errors are not exaggerated. The asymptotic theory of Least absolute error regression was developed by [18], also referred to as the Least Absolute Deviation (LAD).

The absolute deviation function is a non-differetiable function. To construct a surrogate function, use the concave function $\sqrt{v}$ and the inequality to get

$$\sqrt{v} \leqslant \sqrt{v^{(n)}} + \frac{v - v^{(n)}}{2\sqrt{v^{(n)}}}. \tag{2.32}$$

Setting $\sqrt{\nu(\theta)} = \sum_{i=1}^{m}\sqrt{r_i^2(\theta)}$ gives

$$\begin{aligned} \nu(\theta) &\leqslant \nu(\theta^{(n)}) + \frac{1}{2}\sum_{i=1}^{m}\frac{r_i^2(\theta) - r_i^2(\theta^{(n)})}{\sqrt{r_i^2(\theta^{(n)})}} \\ &= g(\theta|\theta^{(n)}). \end{aligned} \tag{2.33}$$

Now minimise the surrogate function by differentiating it with respect to $\hat{\beta}$ and setting it to zero to get the update rule

$$\boldsymbol{\theta}^{(n+1)} = \left[X^T W(\theta^{(n+1)})X\right]^{-1} X^T W(\theta^{(n+1)})\boldsymbol{y}. \tag{2.34}$$

where $W(\theta^{(n)})$ is a diagonal matrix with ith diagonal entry $\omega_i(\theta^{(n)}) = |r_i(\theta^{(n)})|^{-1} = |y_i - x_{ij}^T\theta^{(n)}|^{-1}$.

Comparing the LAD method to the finding the sample quantile one, we notice that the LAD method is an example of quantile regression for $q = 1/2$, also called L_1 regression. The method of L_1 regression has been studied by [20], [18] and [21]. But just like the sample quantile example, the LAD one becomes undefined when the weight $\omega_i(\theta^{(n)}) = \infty$. A simple fix is further discussed by [21], [22].

Fig. 2.3b and table 2.3, show the outcome of the fitted least absolute deviation model. Using the MM method, the objective function converges after 50 iterations giving the parameters $\theta_0 = -20.3597$ and $\theta_1 = 7.4982$.

Using the MM method aided in constructing a majorising surrogate function for a non-differentiable function, which we could differentiate in order to find the next iterate and consequently minimise both the surrogate and the objective functions. As before, the disadvantage of using this method is that it can be computationally expensive to do for a large number of parameters and there is also the cost of many iterations.

Iteration	$\theta^{(n)}$	$f(\theta^{(n)})$	Iteration	$\theta^{(n)}$	$f(\theta^{(n)})$
1	$[-8.9471, 4.8658]$	2.3388	10	$[-19.5970, 7.3194]$	2.0253
2	$[-14.3137, 6.0985]$	2.1246	20	$[-20.0097, 7.4161]$	2.0245
3	$[-16.9842, 6.7142]$	2.0619	30	$[-20.2515, 7.4728]$	2.0241
4	$[-18.5728, 7.0817]$	2.0376	40	$[-20.3404, 7.4936]$	2.0239
5	$[-19.2833, 7.2460]$	2.0267	50	$[-20.3597, 7.4982]$	2.0238

Table 2.3: Iterations LAD

2.4. Introduction to the EM principle

The Expectation-Maximization (EM) method is a special case of the MM method, first described by Dempster, Laird and Rubin in their 1977 paper [7]. The algorithm has proved to be a useful as it takes into account missing or hidden data. The data can be missing due to failure to record certain observations, or it is missing because it is simply not possible to record all data points available. Data could also theoretically be introduced as missing for making the maximum likelihood estimation manageable to compute. This section studies the EM algorithm as a numerical method for finding maximum likelihood estimates, iteratively, given some missing data.

To briefly describe the notion of the method, we can think of the E-step as the step which fills in the missing data. It is estimated by using the conditional expectation for the given observed data and some current estimate of the model parameters. Similarly to the MM method, the M-step then maximizes the simpler surrogate function instead of the likelihood function.

As with the MM method, the EM method is numerically stable and for a concave function it increases the likelihood of the observed data. Therefore, we avoid overshooting or undershooting the parameters at each iteration, as with other algorithms such as the Newton Raphson method. Another advantage of the EM method is that it can handle parameter constrains since it is built into the maximization step. On the other hand, just like the MM method, the EM algorithm converges at an extremely slow rate, in the neighbourhood of the maximum point, and might not always converge to the global maximum if the likelihood isn't concave. The global maximum can be reached by using method of moments as a way of estimating the sub-optimal starting point, or picking a set of random starting points.

This section explores the derivation of the EM method and shows that it respects the **tangency** and **decent** properties. Suppose there are two sets of random variables: $y = (y_1, \cdots, y_n)^T$ that consists of the observed, incomplete variables, and $z = (z_1, \cdots, z_n)^T$ that consists of unobserved, hidden variables also called latent variables.

The aim is to maximise incomplete data likelihood by maximizing the it's log-likelihood or in other words

$$arg\max_{\theta} p(y|\theta) = arg\max_{\theta}[log\big(p(y|\theta)\big)], \tag{2.35}$$

as it is often easier to work with 'log-likelihoods', since it is a strictly increasing function, hence the value of θ which maximizes $p(y|\theta)$ also maximizes $log\big(p(y|\theta)\big)$. Note that the probability $p(y|\theta)$ may also be written in terms of the hidden variables z as a marginal likelihood given by

$$p(y|\theta) = \sum_{z} p(y, z|\theta). \tag{2.36}$$

It is usually easier to choose (y, z) such that the maximum log-likelihood estimation, $log\big(p(y, z|\theta)\big)$, for the complete data becomes easier to find. The marginal log-likelihood $log\big(p(y|\theta)\big)$ is typically harder to optimize. Therefore, to maximise the expected complete data log-likelihood $log\big(p(y, z|\theta)\big)$, use the conditional probability

$$p(y|\theta) = \frac{p(y, z|\theta)}{p(z|y, \theta)}. \tag{2.37}$$

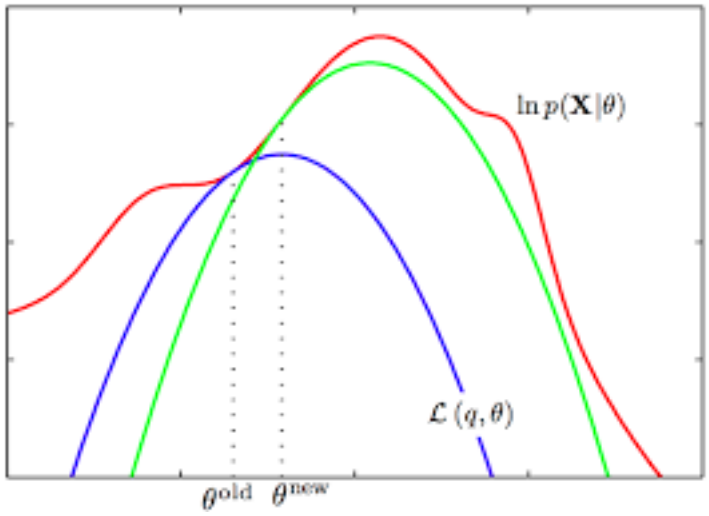
Using the results in 2.36, 2.37 and setting $L(\theta) = log\big(p(y|\theta)\big)$, gives

$$\begin{aligned} L(\theta) &= log\left(\sum_{z} p(y, z|\theta)\right) \\ &= log\left(\sum_{z} p(z|y, \theta^{(n)}) \frac{p(y, z|\theta)}{p(z|y, \theta^{(n)})}\right) \\ &= log\left(E_{z|y,\theta^{(n)}}\left[\frac{p(y, z|\theta)}{p(z|y, \theta^{(n)})}\right]\right) \\ &\geqslant E_{z|y,\theta^{(n)}}\left[log\left(\frac{p(y, z|\theta)}{p(z|y, \theta^{(n)})}\right)\right] \\ &= \sum_{z} p(z|y, \theta^{(n)})\ log\left(\frac{p(y, z|\theta)}{p(z|y, \theta^{(n)})}\right) \\ &= K\left(p_z, \theta\right) \end{aligned} \tag{2.38}$$

where we used Jensen's inequality 2.6 and more specifically the inequality of the concave log function in 2.11, in line four of the above derivation [23]. We call the marginal likelihood $log\Big(p(y|\theta)\Big)$ the **evidence** and the surrogate $K(p_z, \theta)$ is called the Evidence Lower BOund, or **ELBO**. In fig. 2.4, one can see how the marginal likelihood is minorised by the ELBO functions, and how once maximised they provide the next $\theta^{(n+1)}$ iterate. The inequality also satisfies the **descent** property given that the $log(\cdot)$ function is concave, then $log\big(p(y|\theta)\big) \geqslant K(p_z, \theta)$ for $\forall\ \theta$ and it is tangent to the log likelihood at $log\big(p(y|\theta^{(n)})\big) = K(p_z, \theta^{(n)})$.

One can use the the maximum likelihood estimator (MLE) method to find θ,

$$\hat{\theta}_{MLE} = arg\max_{\theta} log\big(p(y|\theta)\big). \tag{2.39}$$



Source: David Rosenberg, Expectation Maximization Algorithm, New York University Lecture Slides, pg.13, June 15, 2015

Figure 2.4: The red curve is the likelihood $log\big(p(y|\theta)\big)$ we want to maximise, while the green and blue curves show two lower bounds, or $L(q,\theta) = K(p_z,\theta)$ for parameters $\theta^{(old)}$, and $\theta^{(new)}$

But for the EM algorithm, we maximise the lower bound to find the optimal value of θ

$$\hat{\theta}_{EM} = arg\max_{\theta}\left[\max_{p_z} K(p_z,\theta)\right]. \tag{2.40}$$

A more precise form of the ELBO function is given by

$$\begin{aligned} K(p_z,\theta) &= \sum_z p(z|y,\theta^{(n)})\ log\big(p(y,z|\theta)\big) - \sum_z p(z|y,\theta^{(n)})\ log\big(p(z|y,\theta^{(n)})\big) \\ &= E_{z|y,\theta^{(n)}}\left[log\big(p(y,z|\theta)\big)\right] - E_{z|y,\theta^{(n)}}\left[log\big(p(z|y,\theta^{(n)})\big)\right] \\ &= Q(\theta|\theta^{(n)}) + R(\theta^{(n)}|\theta^{(n)}), \end{aligned} \tag{2.41}$$

where $Q(\theta|\theta^{(n)}) = E_{z|y,\theta^{(n)}}\left[log\big(p(y,z|\theta)\big)\right]$ is the new surrogate function and $R(\theta^{(n)}|\theta^{(n)}) = -E_{z|y,\theta^{(n)}}\left[log\big(p(z|y,\theta^{(n)})\big)\right]$ is a term representing the difference between the incomplete-data likelihood and the expectation of the completed-data likelihood [23]. Note that maximizing $log\big(p(y|\theta)\big)$ is equivalent to maximizing $Q(\theta|\theta^{(n)})$ since $R(\theta^{(n)}|\theta^{(n)})$ does not depend on θ and we can discard it. Finally, we have

$$\begin{aligned} \theta^{(n)} &= arg\max_{\theta}\left[\max_{p_z} K(p_z,\theta)\right] \\ &= arg\max_{\theta}\left[E_{z|y,\theta^{(n)}}\left[log\big(p(y,z|\theta)\big)\right]\right] \\ &= arg\max_{\theta}\left[Q(\theta|\theta^{(n)})\right] \end{aligned} \tag{2.42}$$

In other words, to create the surrogate function $Q(\theta|\theta^{(n)})$, take the expectation of the complete data log-likelihood given some data y and the current estimate of $\theta = \theta^{(n)}$. Then maximise the surrogate which yields a new parameter estimate $\theta^{(n+1)}$. Repeat the two steps until convergence of the parameter value.

The two properties of the MM algorithm, the descent property and the tangency condition are also conserved [4]. In other words,

$$L(\theta) \geqslant Q(L(\theta)|\theta^{(n)})$$

for all $\theta \neq \theta^{(n)}$. The tangency condition is also obeyed at $\theta = \theta^{(n)}$, where $L(\theta^{(n)}) = Q(L(\theta^{(n)})|\theta^{(n)})$.

The descent property means that with each iteration step, the log-likelihood increases. Given the log-likelihood of the observed data $L(\theta)$, and the fact that the $log(\cdot)$ function is concave gives

$$log\Big(p(y|\theta^{(n+1)})\Big) \geqslant log\Big(p(y|\theta^{(n)})\Big). \tag{2.43}$$

where the descent property of the MM method becomes an ascent one for the EM algorithm.

Formalising the EM algorithm, first pick an arbitrary small ϵ, and an initial value $\theta^{(0)}$.

Algorithm 1: EM algorithm

Initialization of $\theta^{(0)}$;
Initialization of ϵ;
Initialization of $Q(\theta^{(0)}|\theta^{(-1)}) = -\infty$;
while $Q(\theta^{(n+1)}|\theta^{(n)}) - Q(\theta^{(n)}|\theta^{(n-1)}) \geqslant \epsilon$ **do**
  E-step: Compute $Q(\theta|\theta^{(n)})$
  M-step: Compute $\theta^* = arg\max_\theta Q(\theta|\theta^{(n)})$
  Set $\theta^* = \theta^{(n+1)}$
end

Then loop through the two steps, the E-step and M-step, until the difference between the optimal values of two surrogate functions becomes less than ϵ. Once that happens, set the final and optimal value of the surrogate $Q(\theta|\theta^{(n+1)})$ and the log-likelihood $L(\theta)$ to be equal to $\theta^{(n+1)}$, for the (n+1)th iterate.

The EM algorithm is widely used in estimating parameters for models with and without missing variables. In this section, I have only studied the general algorithm but there are many applications of it such as the famous soft clustering EM method [24], Analysis of PET and SPECT Data [25], Allele Frequency estimation [26], analysis of data on HIV/AIDS and other infectious diseases [27] and other.

2.5. An application of the EM method: GMM

Using the EM method for finding Gaussian Mixture Models (GMM) is one of the most popular applications of the EM algorithm. It is an unsupervised learning method, similar to the K-means clustering algorithm because

it groups a set of data points into **K** coherent clusters. However, the main difference being that the K-means algorithm makes hard choices in cluster assignments while the EM algorithm for finding the optimal GMM parameters is a probabilistic model which uses soft clustering approach for distributing the points in **K** different clusters.

Consider a generative model for some data points $x_1, x_2, \cdots, x_m$ which have $z = \{1, 2, \cdots, k\}$ clusters and a probability density for each cluster. Suppose that the data points have been generated from a mixture of Gaussian distributions given that they belong in cluster z, or $x_i|z \sim N(X_i|\mu_z, \sigma_z)$, where each cluster has a cluster mean $\mu_1, \cdots, \mu_k$ and a cluster covariance matrix $\Sigma_1, \cdots, \Sigma_k$. Find the joint probability that a single point is generated from a Gaussian distribution with μ_z and Σ_z, and is from a cluster $z = \{1, 2, \cdots, k\}$ by

$$\begin{aligned} p(x_i, z|\mu_z, \Sigma_z) &= p(z|\mu_z, \Sigma_z)p(x_i|z, \mu_z, \Sigma_z) \\ &= \pi_z N(x_i|\mu_z, \Sigma_z), \end{aligned} \tag{2.44}$$

where π_z is the probability of choosing cluster z and the general, multidimensional normal distribution formula for a single point x is

$$N(\mu, \Sigma) = \frac{1}{(2\pi)^{\frac{m}{2}}|\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right). \tag{2.45}$$

If the model parameters: π_z and $\theta_z = (\mu_z, \Sigma_z)$, are known for each cluster, we could easily evaluate the joint density function $p(x, z|\theta_z)$, but in real life we don't know them. We know that each cluster must have a mean and covariance matrix, but we introduced the cluster assignment variable z that we call a hidden, or latent variable.

In order to find the parameters, assume that each data point is a realisation of the following marginal distribution

$$\begin{aligned} p(x|\theta_z) &= \sum_{z=1}^{k} p(x, z|\theta_z) \\ &= \sum_{z=1}^{k} \pi_z N(x|\theta_z), \end{aligned} \tag{2.46}$$

which is also known as the Gaussian Mixture Model (GMM) for $0 < \pi_z < 1$ and $\sum_{z=1}^{k} \pi_z = 1$. Note that $p(x|\theta_z)$ is a convex linear combination of the k Normal probability densities. In fact, we define a general mixture model with a probability density as follows.

Definition 2.4. *A probability density $p(x|\theta)$ is a mixture distribution or a mixture model, if we can write it as a convex combination of probability densities, i.e.*

$$p(x|\theta_i) = \sum_{i=1}^{k} \omega_i p_i(x|\theta_i),$$

where $\omega_i \geqslant 0$, $\sum_{i=1}^{k} \omega_i = 1$, each p_i is a probability density with parameter θ_i.

Here are two simple examples of GMMs. One example, a one-dimensional GMM with three clusters, shown by the black curve in fig.2.5 where the mixture distribution for the vector $\boldsymbol{\theta}$ containing all the parameters is

$$p(x|\boldsymbol{\theta}) = 0.3N(-3, 2) + 0.3N(0, 1) + 0.4N(8, 3).$$

The other example, shown in fig.2.6, shows the contour lines of two two-dimensional clusters with parameter means $\boldsymbol{\mu}_1 = [1, 5]^T$, $\boldsymbol{\mu}_2 = [3, 1]^T$ and standard deviations $\boldsymbol{\Sigma}_1 = [7, 0.5]^T$ and $\boldsymbol{\Sigma}_2 = [3, 1.5]^T$.

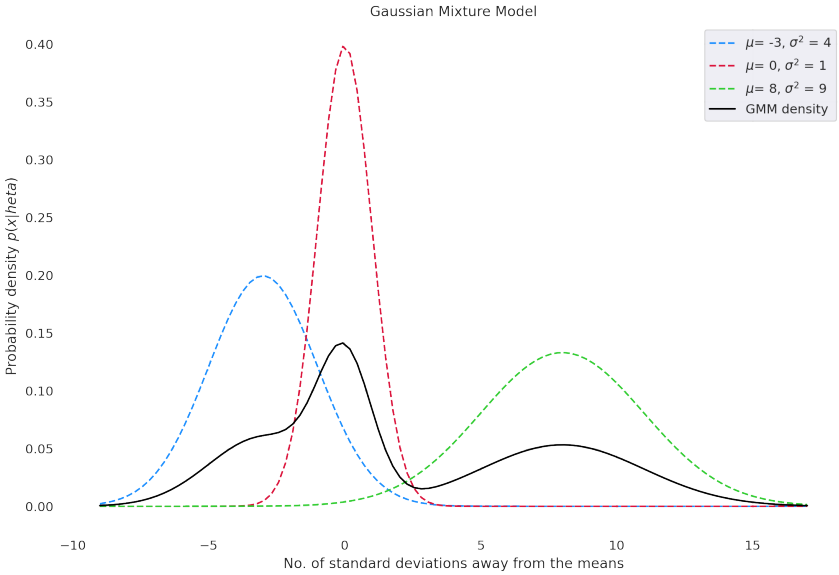


Figure 2.5: One-dimensional GMM for k=3

A common way to solve for the Gaussian model parameters $\theta_z = (\mu_z, \Sigma_z)$ of each cluster is using the Maximum Likelihood Estimation (MLE) method. The likelihood function for all the m data points is

$$\begin{aligned} p(x_1, \cdots, x_m|\boldsymbol{\theta}) &= \prod_{i=1}^{m} p(x_i|\boldsymbol{\theta}) \\ &= \prod_{i=1}^{m} \sum_{z=1}^{k} \pi_k N(x_i|\theta_k). \end{aligned} \tag{2.47}$$

Taking the log of the likelihood would help reduce the multiplicative terms to summation terms. That is,

$$\begin{aligned} \mathcal{L}(\boldsymbol{\theta}) &= log\big(p(x_1, \cdots, x_m|\boldsymbol{\theta})\big) \\ &= \sum_{i=1}^{m} log\left(\sum_{z=1}^{k} \pi_k N(x_i|\theta_k)\right). \end{aligned} \tag{2.48}$$

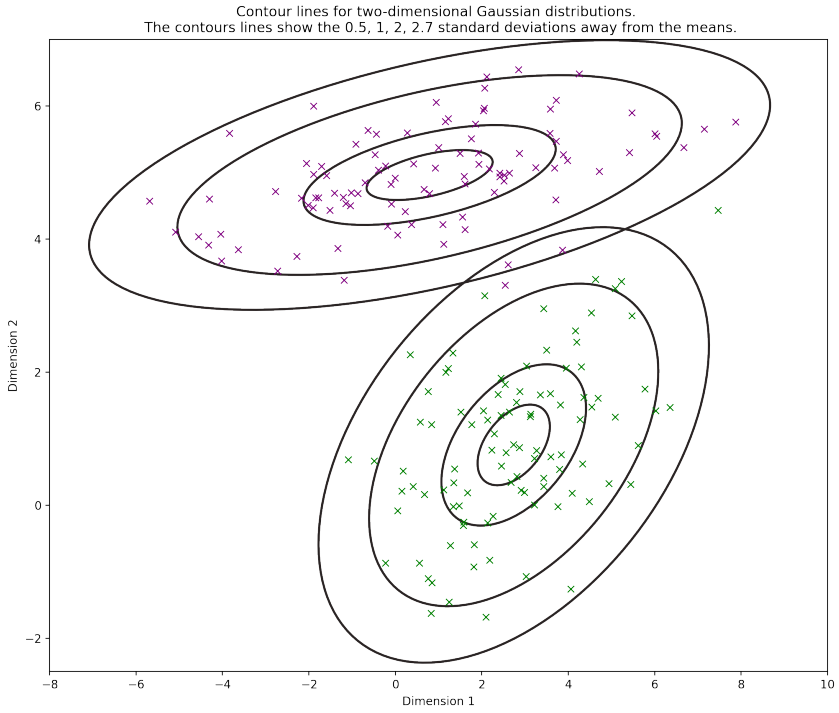


Figure 2.6: Multidimensional GMM for k=2, with centered at (1,5),(3,1) and standard deviations (7,0.5),(3,1.5)

The Expectation-Maximization (EM) method can be used to divide the parameters and iteratively update each one. Note that each parameter update will be dependent on the other parameters at the previous iteration step.

To find the update rules for the parameters $\pi_z, \theta_z = (\mu_z, \Sigma_z)$, differentiate the log-likelihood with respect to each parameter and set the derivative to zero. The derivative of the GMM mean for a single cluster is

$$\begin{aligned} \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \mu_z} &= \sum_{i=1}^{m} \frac{\partial log p\left(x_1, \cdots, x_m|\boldsymbol{\theta}\right)}{\partial \mu_z} \\ &= \sum_{i=1}^{m} \frac{1}{p\left(x_1, \cdots, x_m|\boldsymbol{\theta}\right)} \frac{\partial p\left(x_1, \cdots, x_m|\boldsymbol{\theta}\right)}{\partial \mu_z} = 0 \end{aligned} \tag{2.49}$$

Note that $p\left(x_1, \cdots, x_m|\boldsymbol{\theta}\right)$ is the likelihood function 2.47, which can written as

$$\sum_{z=1}^{k} \pi_z N(x_1, \cdots, x_m|\theta_z) = \sum_{z=1}^{k} \pi_z \frac{1}{(2\pi)^{\frac{m}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\sum_{i=1}^{m} \frac{1}{2}(x_i - \mu_z)^T \Sigma_z^{-1}(x_i - \mu_z)\right). \tag{2.50}$$

Differentiating a single cluster z with respect to the mean μ_z gives

$$\begin{aligned}\frac{\partial N\left(x_1,\cdots,x_m|\theta_z\right)}{\partial\mu_z} &= N\left(x_1,\cdots,x_m|\theta_z\right)\left(-\frac{1}{2}\right)\frac{\partial}{\partial\mu_z}\sum_{i=1}^{m}\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}\left(\mathrm{x}_i-\mu_z\right)\\ &= \sum_{i=1}^{m}\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}N\left(x_1,\cdots,x_m|\theta_z\right)\end{aligned} \tag{2.51}$$

Putting it all together would give us the following equation for the likelihood differential with respect to μ_z

$$\begin{aligned}\frac{\partial p\left(x_1,\cdots,x_m|\boldsymbol{\theta}\right)}{\partial\mu_z} &= \sum_{z=1}^{k}\pi_z\frac{\partial N\left(x_1,\cdots,x_m|\theta_z\right)}{\partial\mu_z}\\ &= \pi_z\frac{\partial N\left(x_1,\cdots,x_m|\theta_z\right)}{\partial\mu_z}\\ &= \pi_z\sum_{i=1}^{m}\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}N\left(x_1,\cdots,x_m|\theta_z\right).\end{aligned} \tag{2.52}$$

Since $p\left(x_1,\cdots,x_m|\boldsymbol{\theta}\right)=\sum_{z=1}^{k}\pi_z N(x_1,\cdots,x_m|\theta_z)$, then the derivative for the log-likelihood becomes

$$\begin{aligned}\sum_{i=1}^{m}\frac{1}{p\left(x_1,\cdots,x_m|\boldsymbol{\theta}\right)}\frac{\partial p\left(x_1,\cdots,x_m|\boldsymbol{\theta}\right)}{\partial\mu_z} &= \sum_{i=1}^{m}\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}\frac{\pi_z N\left(x_1,\cdots,x_m|\theta_z\right)}{\sum_{k=1}^{z}\pi_z N(x_1,\cdots,x_m|\theta_z)}\\ &= \sum_{i=1}^{m}r_{iz}\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}=0,\end{aligned} \tag{2.53}$$

where the r_{iz} is called the responsibility and it is the mathematical definition of what we called 'soft assignments'. It is simply showing the percentage that the $z-th$ component is probabilistically responsible for the data points. Re-arrange the above to obtain the update rule for a single cluster mean μ_z. Since Σ_z^{-1} is invertible, it can't be zero so $\sum_{i=1}^{m}r_{iz}\left(\mathrm{x}_i-\mu_z\right)^T=0$, which gives the update for the mean parameter

$$\begin{aligned}\mu_z^{(n+1)} &= \frac{\sum_{i=1}^{m}r_{iz}\mathrm{x}_i}{\sum_{i=1}^{m}r_{iz}}\\ &= \frac{1}{N_z}\sum_{i=1}^{m}r_{iz}\mathrm{x}_i,\end{aligned} \tag{2.54}$$

where $N_z=\sum_{i=1}^{m}r_{iz}$ is the 'number' of data points "soft assigned" to cluster z.

We can derive the update rule the covariance matrix Σ_z for a single cluster z, in a similar way. First, differentiate with respect to the log-likelihood to get that

$$\begin{aligned}\frac{\partial\mathcal{L}(\boldsymbol{\theta})}{\partial\Sigma_z} &= \sum_{i=1}^{m}\frac{\partial log p\left(x_1,\cdots,x_m|\boldsymbol{\theta}\right)}{\partial\Sigma_z}\\ &= \sum_{i=1}^{m}\frac{1}{p\left(x_1,\cdots,x_m|\boldsymbol{\theta}\right)}\frac{\partial p\left(x_1,\cdots,x_m|\boldsymbol{\theta}\right)}{\partial\Sigma_z}=0\end{aligned} \tag{2.55}$$

Again, differentiating the likelihood function in 2.50 for a single cluster z with respect to the covariance matrix Σ_z gives

$$\begin{aligned}\frac{\partial N\left(x_1,\cdots,x_m|\theta_z\right)}{\partial\Sigma_z} &= \frac{\partial}{\partial\Sigma_z}\left((2\pi)^{-\frac{m}{2}}|\Sigma_z|^{-\frac{1}{2}}\exp\left(-\sum_{i=1}^{m}\frac{1}{2}\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}\left(\mathrm{x}_i-\mu_z\right)\right)\right)\\ &= (2\pi)^{-\frac{m}{2}}\left[\frac{\partial}{\partial\Sigma_z}|\Sigma_z|^{-\frac{1}{2}}\exp\left(-\frac{1}{2}\sum_{i=1}^{m}\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}\left(\mathrm{x}_i-\mu_z\right)\right)\right]\\ &+ (2\pi)^{-\frac{m}{2}}\left[|\Sigma_z|^{-\frac{1}{2}}\frac{\partial}{\partial\Sigma_z}\exp\left(-\sum_{i=1}^{m}\frac{1}{2}\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}\left(\mathrm{x}_i-\mu_z\right)\right)\right]\\ &= N\left(x_1,\cdots,x_m|\theta_z\right)\left[-\frac{1}{2}\left(\Sigma_z^{-1}-\Sigma_z^{-1}\left(\mathrm{x}_i-\mu_z\right)\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}\right)\right].\end{aligned} \tag{2.56}$$

Putting it all together for all clusters gives the following likelihood equation

$$\begin{aligned}\frac{\partial p\left(x_1,\cdots,x_m|\boldsymbol{\theta}\right)}{\partial\Sigma_z} &= \sum_{z=1}^{k}\pi_z\frac{\partial N\left(x_1,\cdots,x_m|\theta_z\right)}{\partial\Sigma_z}\\ &= \pi_z\frac{\partial N\left(x_1,\cdots,x_m|\theta_z\right)}{\partial\Sigma_z}\\ &= \pi_z N\left(x_1,\cdots,x_m|\theta_z\right)\left[-\frac{1}{2}\left(\Sigma_z^{-1}-\Sigma_z^{-1}\left(\mathrm{x}_i-\mu_z\right)\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}\right)\right].\end{aligned} \tag{2.57}$$

Substituting the above back to 2.55, using the likelihood function and the responsibility variable defined in 2.53 gives

$$\begin{aligned}\frac{\partial N\left(x_1,\cdots,x_m|\theta_z\right)}{\partial\Sigma_z} &= \sum_{i=1}^{m}r_{ik}\left[-\frac{1}{2}\left(\Sigma_z^{-1}-\Sigma_z^{-1}\left(\mathrm{x}_i-\mu_z\right)\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}\right)\right]\\ &= -\frac{1}{2}\sum_{i=1}^{m}r_{iz}\left(\Sigma_z^{-1}-\Sigma_z^{-1}\left(\mathrm{x}_i-\mu_z\right)\left(\mathrm{x}_i-\mu_z\right)^T\Sigma_z^{-1}\right)\\ &= -\frac{1}{2}\Sigma_z^{-1}\sum_{i=1}^{m}r_{ik}+\frac{1}{2}\Sigma_z^{-1}\left(\sum_{i=1}^{m}r_{izk}\left(\mathrm{x}_i-\mu_z\right)\left(\mathrm{x}_i-\mu_z\right)^T\right)\Sigma_z^{-1},\end{aligned} \tag{2.58}$$

where we can set $\sum_{i=1}^{m}r_{iz}=N_z$ again, being the number of points that are 'soft assigned' to cluster z. Rearranging gives the covariance matrix update rule

$$\begin{aligned}N_z\Sigma_z^{-1} &= \Sigma_z^{-1}\left(\sum_{i=1}^{m}r_{iz}\left(\mathrm{x}_i-\mu_z\right)\left(\mathrm{x}_i-\mu_z\right)^T\right)\Sigma_z^{-1}\\ \Sigma_z^{(n+1)} &= \frac{1}{N_z}\sum_{i=1}^{m}r_{iz}\left(\mathrm{x}_i-\mu_z\right)\left(\mathrm{x}_i-\mu_z\right)^T.\end{aligned} \tag{2.59}$$

Finally, we need to find an update rule for π_z, the probability of choosing cluster z, also known as the weight of a Gaussian model in the Gaussian mixture. For this we use the Lagrange multiplier method, since there is the

constraint $\sum_{z=1}^{k} \pi_z = 1$. We get a new function made up of the log-likelihood and a Lagrange multiplier,

$$\begin{aligned}\Phi(\boldsymbol{\theta}, \pi_z) &= \mathcal{L}(\boldsymbol{\theta}) + \lambda \left(\sum_{z=1}^{k} \pi_z - 1 \right) \\ &= \sum_{j=1}^{m} log \sum_{z=1}^{k} \pi_z N(x_j|\theta_z) + \lambda \left(\sum_{z=1}^{k} \pi_z - 1 \right).\end{aligned} \tag{2.60}$$

Differentiating the new function $\Phi(\boldsymbol{\theta}, \pi_z)$, for a single cluster, with respect to π_z gives

$$\begin{aligned}\frac{\partial \Phi(\boldsymbol{\theta}, \pi_z)}{\partial \pi_z} &= \frac{1}{\pi_z} \sum_{i=1}^{m} \frac{N\left(x_1, \cdots, x_m|\theta_z\right)}{\sum_{k=1}^{z} \pi_z N(x_1, \cdots, x_m|\theta_z)} + \lambda \\ &= \frac{1}{\pi_z} \sum_{i=1}^{m} r_{iz} + \lambda \\ &= \frac{N_z}{\pi_z} + \lambda\end{aligned} \tag{2.61}$$

Differentiating the function with respect to the Lagrange multiplier gives

$$\frac{\partial \Phi(\boldsymbol{\theta}, \pi_z)}{\partial \lambda} = \sum_{z=1}^{k} \pi_z - 1. \tag{2.62}$$

Setting the differential equations to zero gives that the $\pi_z = -N_z/\lambda$ and the constraint $\sum_{z=1}^{k} \pi_z = 1$. Therefore,

$$\begin{aligned}\sum_{z=1}^{k} -\frac{N_z}{\lambda} &= 1 \\ \pi_z^{(n+1)} &= \frac{N_z}{m} = \frac{\sum_{i=1}^{m} r_{iz}}{m}.\end{aligned} \tag{2.63}$$

The update rules for each parameter value in the GMM are derived, and so we construct the EM algorithm to update them iteratively. The algorithm goes as follows

Algorithm 2: EM algorithm for constucting GMM

Initialization of $\mu_z^{(0)}$;
Initialization of $\Sigma_z^{(0)}$;
Initialization of $\pi_z^{(0)}$;
Initialization of ϵ;
Set $\theta_z^{(n)} = \left(\mu_z^{(n)}, \Sigma_z^{(n)}, \pi_z^{(n)}\right)$
while $\mathcal{L}(\theta_z^{(n+1)}) - \mathcal{L}(\theta_z^{(n)}) \geqslant \epsilon$ **do**
    E-step: Compute $r_{iz} = \frac{\pi_z^{(n)} N\left(x_1, \cdots, x_m|\theta_z^{(n)}\right)}{\sum_{k=1}^{z} \pi_z^{(n)} N(x_1, \cdots, x_m|\theta_z^{(n)})}$;
    M-step: Compute $\mu_z^{(n+1)}, \Sigma_z^{(n+1)}, \pi_z^{(n+1)}$;
    Set $\theta^{(n)} = \theta^{(n+1)}$
end

It is common practice to initiate the starting parameters using the same method as in k-means clustering where we either choose k random data points or other. In order to find the optimal initial admixture proportion π_z, one can run the algorithm for a range of random initialisations, then pick the one with the highest likelihood. In his dissertation [28], Zhengyu Hu summarises methods for GMM parameter initialisation for larger classification problems.

In this case, the EM method has successfully introduced a latent variable as well as allowed us to split the model parameters in order to individually update each one of them. Since the MM properties hold, the log-likelihood function will increase with each iteration. However, it increases rapidly in the first iterations, and increases slowly as the number of iterations increase [4]. This means that it can be computationally expensive and quite slow to compute the GMM parameters for a large number of clusters. In addition, the EM algorithm for GMM does not guarantee convergence to the global maxima so one will need to run the algorithm for a variety of starting points. Note that, the EM algorithm for finding GMM parameters is well-known and widely used in the data science community, so I have not included any coded examples in this section, but have only focused on the derivation.

3

Applying the MM method to a Hawkes process

3.1. Preliminary Model Definitions

3.1.1. The Self-Exciting Point Process

This section looks at a discrete case of a self-exciting point process called the Hawkes process. We estimate the model's parameters using the MM method, test the update rules on generated data, and then apply the update functions to real-world data, to model the rate of new COVID-19 cases in the UK.

We will start by defining a point process. Using the definitions from [29], a **point process** is a random process used to model events that occur at random intervals relative to the time axis or the space axis. Hence it lives on the non-negative real line. We will be looking at a temporal point process with events that are observed over time as a time series. Note that the events must be pointlike, discrete entities, i.e. the event occupies a small area of the domain, a concept heavily used in Physics. In other words, we can represent a temporal point process as a simple list of events and their relevant time occurrences.

To formalise this, let X be a **point process** is defined on a bounded set S and $\{x_1, x_2, \cdots, x_n\}$ be an ordered set of points. Let $N_X(A) = \#\{x_i \in A\}$ represent the random variable that finds the number of points of X in the finite set $A \subseteq S$, where the symbol $\#$ represents the number of points in the set. For the temporal point process we have that $A \subseteq S \subseteq \mathbb{R}$, where $\mathbb{R}$ is the real line representing time.

A **self-exciting point process** is a process has the following property $Cov[N_X(A)N_X(B)] > 0$ for any two sets A and B. This means that in a self-exciting point process, an occurrence of a single point or event prompts the occurrence of other points/events. Such models are often used in seismology, the study of events of earthquakes and seismic waves.

The temporal point process is a stochastic process where the events are scattered in time and occur in time points $\{T_i\}$ where $0 \leqslant T_1 \leqslant T_2 \leqslant \cdots \leqslant T_k$. The point process, has a counting process and an interval process.

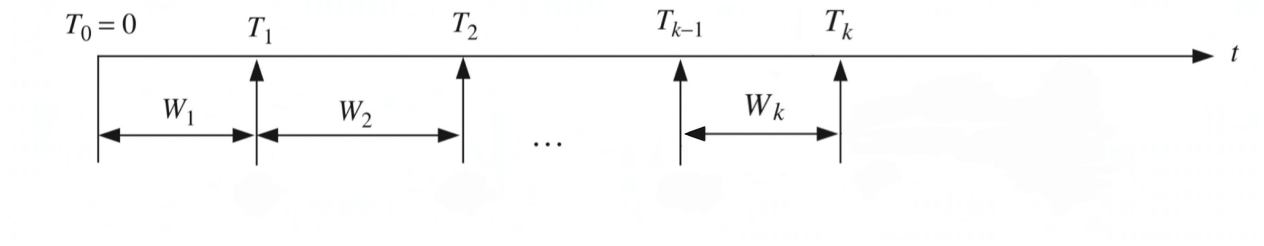


Figure 3.1: Relationship between T and W.

The former deals with the number of points/events in a fixed interval, and the latter finds the time intervals between the following points/events. Therefore we represent a point process X as

- The counting process $N_X(0, t)$ which finds the number of events between the time interval (0,t). So we write it as $N_X(0, t) = \{0 < T_i \leqslant t\}$.
- The interval process denoted by $\{W_i\}$ where $W_i = T_i - T_{i-1} > 0$, represents the distance between two consecutive points.

Fig 3.1 shows the relationship between the time process $\{T_i\}$ and the interval process $\{W_i\}$. The model for the Hawkes process that will be explored in the later section, focuses on the counting process of events for identically distributed time intervals W_i.

3.1.2. Poisson Process

The most straightforward class of point processes is the Poisson process, often used in queuing theory to model random events such as the arrival of customers in a queue at a store or the arrival of phone calls at a call centre. There are two cases of the Poisson process, the **homogeneous or stationary Poisson point process** and the **inhomogeneous Poisson point process**. The homogeneous Poisson process is characterised via the Poisson distribution, where the constant parameter, known as the rate or intensity, is the average number of events in some time period. On the other hand, the inhomogeneous Poisson point process is a Poisson parameter set as some time-dependent function in a bounded set on which the Poisson process is defined.

We only need the simple Poisson point process characterised by the Poisson distribution. Define the Poisson probability mass function as the distribution of a random Poisson variable N such that the probability that N equals some number of event occurrences n is given by

$$P\big(N = n\big) = \frac{\lambda^n e^{-\lambda}}{n!} \; for \; \lambda > 0, \;\; n = 0, 1, 2, \cdots \tag{3.1}$$

where $n!$ is the factorial of n, λ is the parameter which determines the shape of the distribution and is also known to be the expected or average value of N events. We can also interpret the average number of events λ

as the number of events τ during some period of time W, i.e. $\lambda = \tau W$ is comprised of τ number of events per unit of time W. So the p.m.f becomes

$$P\big(\tau \text{ events in time interval } W\big) = \frac{(\tau W)^n e^{-\tau W}}{n!} \text{ for } \tau W > 0, \ n = 0, 1, 2, \cdots \tag{3.2}$$

Note that the Poisson distribution has the property of complete independence where each event is an independent and identically distributed (i.i.d) random variable. In other words, each event is independent of the other. The Poisson process is defined as a completely random process.

3.1.3. The Hawkes process

One problem with just adopting the Poisson distribution for modelling events such as the pandemic cases per week for example, is that the process has a memoryless property, i.e. a future event depends only on the constant rate λ and relevant information about the number of event occurrences n. In other words, the events arrive independently of each other at a constant rate (for the homogeneous Poisson point process). Hence the Poisson point process doesn't have the excitation property described earlier, as events do not depend on the past.

However, for modelling events such as the number of COVID-19 cases per time unit such as a week, we know that the occurrence of new cases per day, or week depends on the number of affected people in the last few days or weeks. Therefore, the likelihood of observing an event in the future increases with the arrival of an event now. So we introduce a simple Hawkes process.

The Hawkes process is a counting process where one can use it to model a sequence of arrivals or counts over time. The Hawkes process has been used to model the arrival of financial trade orders [30] and limit order books [31], social media retweet cascades [29], criminal fatalities per week and earthquake occurrences, [32]. The defining feature of the Hawkes process is that it is a self-exciting point process which means that each arrival or each new count 'excites' the next arrivals. Unlike the Poisson process, it depends on history and has a long term memory. Therefore each new arrival increases the rate of future arrivals for some time period.

There are many articles about modelling discrete, inter-dependent events over continuous time using the Hawkes process. However, we observe events in time intervals and not in millisecond instances. Therefore this chapter studies a model with equally spaced out time intervals W between new arrivals. The assumption is useful when the data has no time stamps because we still observe the jumps as in the continuous case without knowing the exact timestamp of the jump, only the time interval. So the continuous-time Hawkes process can be useless in modelling data that comes from equally spaced time intervals. For example, if we look at the COVID-19 data, we can determine the reported case rates per week, but the actual date of infection is unknown.

3.2. The discrete Hawkes process

Assume that $W_i = 1$ for all $i \in \mathbb{Z}^+$. Suppose we have the initial, base parameter $\mu = \lambda_0$ for the first time period. Then we draw a random variable $\Delta N_i \sim Poi(\lambda_i)$ which represents the number of events or counts in the first time interval. Suppose we observe $\{\Delta N_1, \Delta N_2, \cdots, \Delta N_N\}$ number of counts per unit of time. Then the following iterative method describes the discrete time Hawkes process.

We draw a random variable $\Delta N_0 \sim Poi(\lambda_0)$. The new data point updates the Poisson parameter λ_1, by multiplying the data point at the previous step by an infectious rate α. This would create the 'excitation' for the process. We also add a decay rate γ. The decay rate keeps the 'memory' of the previous time steps by multiplying the 'older' data points by γ with each time step. Below you can see the updates for each λ_k for the first three-time intervals,

$$\begin{aligned}
&\lambda_1 = \mu + \alpha\Delta N_0 \ \rightarrow \ \Delta N_1 \sim Poi(\lambda_1) \\
&\lambda_2 = \mu + \gamma\alpha\Delta N_0 + \alpha\Delta N_1 \ \rightarrow \ \Delta N_2 \sim Poi(\lambda_2) \\
&\lambda_3 = \mu + \gamma^2\alpha\Delta N_0 + \gamma\alpha\Delta N_1 + \alpha\Delta N_2 \ \rightarrow \ \Delta N_3 \sim Poi(\lambda_3) \\
&\cdots .
\end{aligned} \tag{3.3}$$

The parameter restrictions are as follows $\mu > 0,\ \alpha > 0$ *and* $0 < \gamma < 1$. Therefore, the model can be summarised by the following equation,

$$\lambda_k = \mu + \sum_{l=1}^{k-1} \alpha\gamma^{k-l-1}\Delta N_l. \tag{3.4}$$

Note that since there are no time stamps so we assume the time intervals for this model are equally spaced and equal to 1.

In fig. 3.2, I have generated some count data for the equally spaced time intervals. The data points N_l are drawn from the updated λ_k at each time step, are the number of counts per time step, and $\Delta N_l \in \mathbb{N}$ and $\{\lambda_k \in \mathbb{R} | \lambda_k > 0\}$. In other words the data drawn from the Poisson distribution are natural numbers and the λ_k are positive real numbers.

In this example, the synthetic data is generated for the model parameters $\mu = 8, \gamma = 0.1$ *and* $\alpha = 0.25$ using the model 3.4. We see that the $\lambda_0 = 8$ since the parameter $\mu = 8$, then a random variable $\Delta N_1 \sim Poi(\lambda_0)$ is drawn, using a Poisson random number generator, see code in Appendix. The new data point updates the λ variable and the iteration continues for 100 points. The data points are plotted using the bar plot and the deterministic parameter values of λ is plotted using a step plot.

3.3. The likelihood and negative log-likelihood function

Suppose we observe some sample data point or events $\{\Delta N_0, \Delta N_1, \Delta N_2, \cdots, \Delta N_N\}$ for some identically distributed time intervals $W_1 = W_2 = \cdots = W_N = 1$ and some given values of the unknown parameters μ, α *and* γ.

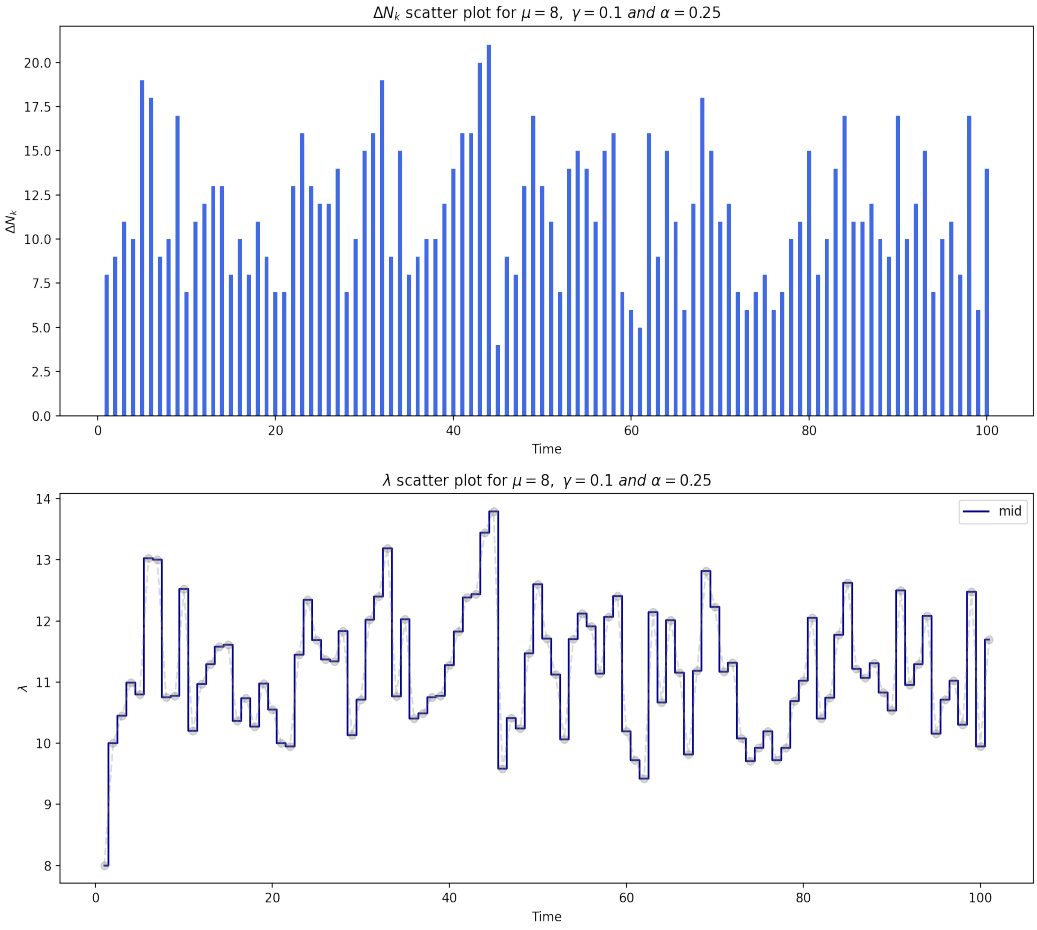


Figure 3.2: An example point process realisations for $\{W_k\} = 1$ for all $k \in \mathbb{Z}^+$ and the counts ΔN_k for the corresponding parameter values λ_k

Using 3.2, the probability that a random variable N is equal to a single data point is given by the Poisson p.m.f

$$P(N = \Delta N_i) = \frac{\lambda_i^{\Delta N_i} e^{-\lambda_i}}{\Delta N_i!}, \tag{3.5}$$

Computing the likelihood function $L(\theta)$ to estimate the model parameters $\theta = (\mu, \alpha, \gamma)$ for which the observed data points are most likely, by

$$L(\theta) = \frac{(\lambda_1^{\Delta N_1} \lambda_2^{\Delta N_2} \cdots \lambda_N^{\Delta N_N})(e^{-\lambda_1} e^{-\lambda_2} \cdots e^{-\lambda_N})}{\Delta N_1! \Delta N_2! \cdots \Delta N_N!}. \tag{3.6}$$

We find the optimal values of our model parameters that fit our data such that $P(\Delta N_i|\mu, \alpha, \gamma)$, using the MLE method because taking a log of the likelihood can split up the product of large number of probabilities which can help numerically since instead we sum the log of small probabilities.

In this case, the negative log-likelihood is used to make it easier to use the MM method to find a surrogate function. Therefore the surrogate would need to be minimised. Consequently, the maximisation problem becomes a minimisation by taking the negative log-likelihood of the likelihood function

$$-log(L) = -\sum_{k=1}^{N} \Delta N_k log(\lambda_k) + \sum_{k=1}^{N} \lambda_k. \tag{3.7}$$

Note that the value of $\sum_{k=1}^{N} log(\Delta N_k!)$ is the same for different values of our parameters and so it is discarded since it only adds an extra step for the numerical estimation of the three unknown parameters. Therefore, the optimisation problem becomes

$$min \left\{ -log\{L(\boldsymbol{\theta})\} \right\} = min \left\{ -\sum_{k=1}^{N} \Delta N_k log(\lambda_k) + \sum_{k=1}^{N} \lambda_k \right\}, \tag{3.8}$$

where $\boldsymbol{\theta} = (\mu, \alpha, \gamma)$.

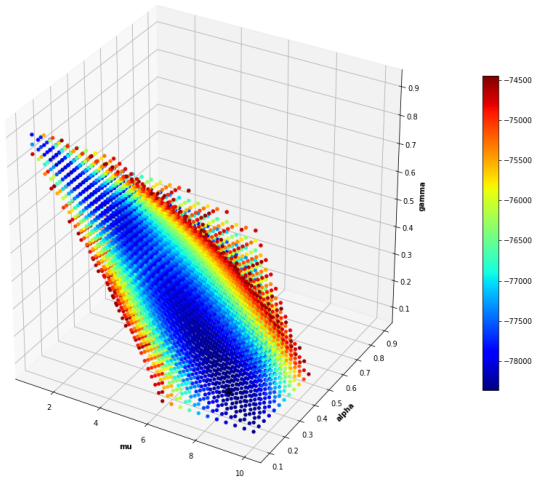
Fig. 3.3, shows initial analysis of the parameter values for data generated by $\mu = 8$, $\gamma = 0.1$ *and* $\alpha = 0.25$. I sampled a range of 25 equally spaced points for each parameter where $1 \geqslant \mu \geqslant 10$, $0.1 \geqslant \alpha \geqslant 0.9$ and $0.1 \geqslant \gamma \geqslant 0.9$, according to the assumptions about the parameters. Note that the parameter $0 \leqslant \gamma \leqslant 1$ so I picked a range within that but I could have also used $0.1 \geqslant \gamma \geqslant 0.99$ or somthing else. In this case, we can re-arrange the model equation for λ_k to give a iterative method of finding λ_{k+1} without having to sum all the data points at each new value of λ

$$\lambda_{k+1} = (\lambda_k - \mu)\gamma + \mu + \alpha \Delta N_k. \tag{3.9}$$

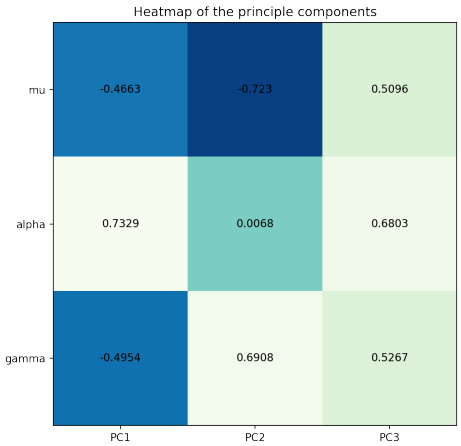
This is the model equation we use to find the λ_k values given N data points ΔN_k. Finding the λ_k for different parameter value combinations and some data points, we can use this to find the negative log-likelihood values using equation 3.7.

Using the λ_k values, we plot the three parameters, that take on a range of 25 values each, using a 3D plot and shade each parameter combination or point on the axis, according to how small or large the value of

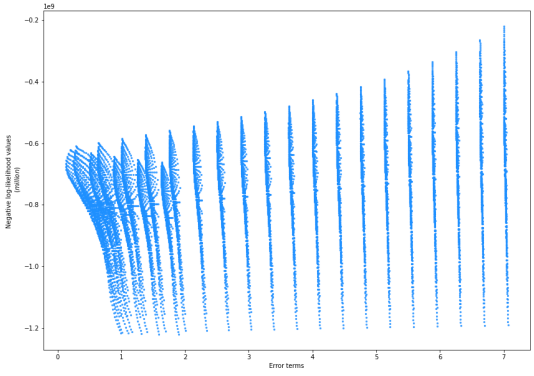
the negative log-likelihood is. The colour bar on the side of the plot helps identify the points that take on the smallest negative log-likelihood values.



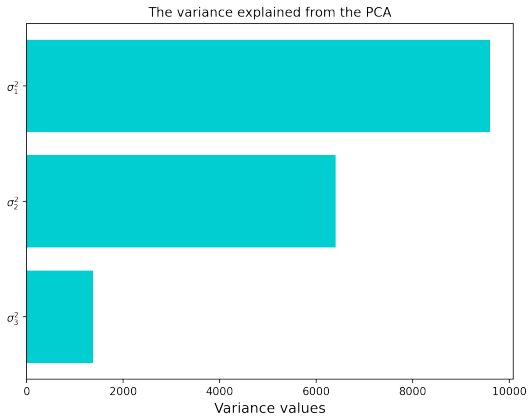
(a) 3D plot of 0.95 times the smallest negative log-likelihood values



(b) PCA heatmap of the parameter values



(c) Negative log-likelihood values against the error terms



(d) PCA variances

Figure 3.3: Analysing the synthetic data properties

In fig. 3.3a, the colour bar shows a range of values for the negative log-likelihood: from 0.95 times the minimum value until the minimum negative log-likelihood, call this *Lmin*. Therefore, the minimum negative log-likelihood is $Lmin = -78,364$, so the upper bound of the colour bar is $Lmin * 0.95 = -74,446$. The range of values 'filter' the parameter combinations to give the 3D shape of the points that provide the values

of the negative log-likelihood within the region from $-74,446$ to $-78,364$. The 3D plot shows a rainbow-like pattern forming. In the middle of the shape, the values in blue have a very close range of negative log-likelihood values: from $-77,500$ to $-78,000$. In general, we won't be able to find the exact value of the 'real' negative log-likelihood nor the exact parameter combination, but we can get close to it. The parameter update function could converge to any point in the blue region, also providing good estimates for the data since the points have a very small negative log-likelihood.

The appendix A.13. shows the 3D plot for the full range of negative log-likelihood values: from $288,093$ to $-78,363$. Since, the points on the 3D plot represent 0.95 times the smallest negative log-likelihood value, those points have a relatively small negative log-likelihood range; therefore any point in the plot 3.3a might be a good parameter estimate for the data. Therefore, we can analyse the 3D shape of the parameter values for the smallest negative log-likelihood using Principle component analysis (PCA).

Fig. 3.3b shows the principal components with values for each parameter. I did the analysis using two different methods - using the Scikit-learn and the Numpy packages to make sure it is correct, shown in A.15. and A.16.. The first principal component shows a correlation between μ and γ, and a negative correlation between the two with α. In other words, an increase in the values of μ or γ would decrease the value of α. This makes sense as some initial analysis in A.11. suggests that the values of α and γ should add to less than 1, i.e. $\alpha + \gamma < 1$ or the process will 'explode' and therefore will not be weak stationary. Therefore, an increase in α would also decrease γ and μ.

From the first principal component, we notice that the value of $\gamma = -0.4954$ is similar to the value of $\mu = -0.4663$, whereas $\alpha = 0.7329$ is positive, going in the opposite direction to μ and α. Conducting the analysis with other values of μ, α and γ gives a similar relationship between the parameters.

The second principal component shows the relationship between μ and γ again. This time the small value of $\alpha = 0.0068$ means that it is not correlated in this direction compared to the other two parameters. We can see that $\mu = -0.723$ and $\gamma = 0.6908$, are inversely proportional to each other. This means that an increase in γ will lead to a decrease in μ.

Fig. 3.3d shows that we can disregard the last principal as it accounts for only 7.9% of the total variance. Whereas the first component accounts for 55.2% and the second: 36.9% of the total variance, giving a total pf 92% being accounted for by the first two principal components.

In summary, the analysis suggests that there is firstly a strong negative relationship between α with γ and μ, and given how we choose α, there is also an inverse relationship between γ and μ.

Having used PCA to analyse several other examples with different parameter values, there is also another result that the principal components provided: the first principle component in several examples suggests there is a strong inverse relationship between α and γ and a minimal, negligible correlation between α and μ; the

second principal component suggests a strong positive correlation for μ and negligible, negative ones for μ and α; the third principle component suggests a strong positive correlation between α and γ, and a small positive one with μ. The total variance accounted for, for the first two components, in the other examples, did not reach 90%, however. In summary, the multiple runs suggest that α and γ always have an inverse relationship, once chosen, the two parameters have a small, inverse relationship with μ. In other words, once the two parameters are fixed, the model variance increases and decreases with the choice of μ.

Fig. 3.3c shows the negative log-likelihood values for each parameter combination, plotted against the normed difference between the actual parameter values I used to generate the data and all the values used to plot the 3D figure. The plot shows that the negative log-likelihood decreases with the parameter error. But it also suggests that some more significant parameter errors still provide a small negative log-likelihood. As we have seen in the 3D plot, there seems to be a range of values that can give a reasonable estimate for the model parameters. Therefore, the update functions for the parameters can converge to values that are not the exact same values used to generate the data, but the new values could also model the data well.

3.4. The Expectation and Variance of the infinite Hawkes process

The Hawkes process is a weak stationary process, so the count data changes over time, but on average the mean and variance stay the same, in the long run. This section derives the expectation and variance for λ_k as $k \to \infty$. The mean and variance help to describe the properties of the weak stationary Hawkes process without specifying a distribution for the process.

I have used an iterative approach to find the expectation of λ_k as $k \to \infty$ in the Appendix A.19.. However, my supervisor suggested a different way of finding the long term mean and variance.

For this method we use the law of total expectation where given two random variables X and Y, the expectation of X is given by

$$\mathbb{E}\left(X\right) = \mathbb{E}\left(\mathbb{E}\left(X|Y\right)\right), \tag{3.10}$$

and the variance is given by the law of total variance,

$$\mathrm{Var}\big(X\big) = \mathbb{E}\left(\mathrm{Var}\big(X|Y\big)\right) + \mathrm{Var}\big(\mathbb{E}\left(X|Y\right)\big). \tag{3.11}$$

The derivation also requires the use of the limit definition of a derivative, given by

$$\frac{dy}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}. \tag{3.12}$$

We assume that λ_k is a function of time, so we can re-write the time step $k = t$, and assume the time is not a constant unit of time. We take $\gamma = (1-\beta)$ for ease of use and re-write the expression in 3.9 for $\lambda(t)$

$$\lambda(t+\delta t) = \mu + (1-\beta\delta t)(\lambda(t) - \mu) + \alpha N_t, \quad N_t \sim \mathrm{Poi}(\lambda(t)\delta t). \tag{3.13}$$

Define the function for the expectation for $\lambda(t+\delta t)$ to be

$$\begin{aligned} M(t+\delta t) &= \mathbb{E}[\lambda(t+\delta t)] \\ &= \mathbb{E}[\mathbb{E}[\lambda(t+\delta t) \mid \lambda(t)]] \ \textit{using 3.10} \\ &= \mathbb{E}[\mu\beta\delta t + (1+(k-\beta)\delta t)\lambda(t)] \ \textit{using 3.13} \\ &= \mu\beta\delta t + (1+(\alpha-\beta)\delta t)M(t) \ \textit{expand and simplify}. \end{aligned} \tag{3.14}$$

The variance function can be found in a similar way. That is,

$$\begin{aligned} V(t+\delta t) &= \mathrm{Var}[\lambda(t+\delta t)] \\ &= \mathbb{E}\left[\mathrm{Var}[\lambda(t+\delta t)|\lambda(t)] + \mathrm{Var}[\mathbb{E}[\lambda(t+\delta t) \mid \lambda(t)]]\right] \ \textit{using 3.11} \\ &= \mathbb{E}[\alpha^2\mathrm{Var}[N(t)]] + \mathrm{Var}[\mu\beta\delta t + (1+(\alpha-\beta)\delta t)\,\lambda(t)] \ \textit{using 3.13} \\ &= \mathbb{E}[\alpha^2\lambda\delta t] + (1+(\alpha-\beta)\delta t)^2\,\mathrm{Var}[\lambda(t)] \ \textit{expand and simplify} \\ &= \delta t\alpha^2 M(t) + (1+(\alpha-\beta)\delta t)^2\,V(t). \end{aligned} \tag{3.15}$$

Using 3.12 re-arange the two functions for the expectation and variance to the following first order differential equations as $\delta t \to 0$,

$$\begin{aligned} M^{'} &= \mu\beta + (\alpha-\beta)M \\ V^{'} &= 2(\alpha-\beta)V + \alpha^2 M. \end{aligned} \tag{3.16}$$

My supervisor plugged the equations into the Mathematica software to get the following solutions

$$M(t) = \frac{\mu\beta}{\beta-\alpha} + \frac{(\mu + m_0(\alpha-\beta))}{\alpha-1}e^{(\alpha-\beta)t} \tag{3.17}$$

$$V(t) = \frac{1}{2(\alpha-\beta)^2}\left(\mu\beta\alpha^2 - 2\left(\mu + m_0(\alpha-\beta)\right)\alpha^2 e^{(\alpha-\beta)t} + \left(\mu\beta\alpha^2 + 2v_0(\alpha-\beta)^2 + 2\alpha^2(k-\beta)m_0\right)e^{2(\alpha-\beta)\epsilon}\right), \tag{3.18}$$

for some initial values $m_0 := M(0), v_0 := V(0)$. Taking the limit as time goes to infinity, the expectation and variance converge to

$$\begin{aligned} M(t) &= \frac{\mu\beta}{\beta-\alpha} \\ V(t) &= \frac{\mu\beta\alpha^2}{2(\alpha-\beta)^2}. \end{aligned} \tag{3.19}$$

Fig 3.4 shows the rate at which the expectation converges as time goes to infinity (in this case t $\to$ 100). Similarly, fig. 3.5 shows the rate at which the variance converges as time goes to infinity. Just after around 20 time steps the expectation and variance have converged to the expected constants, meaning we have the two model properties without describing the distribution of the process.

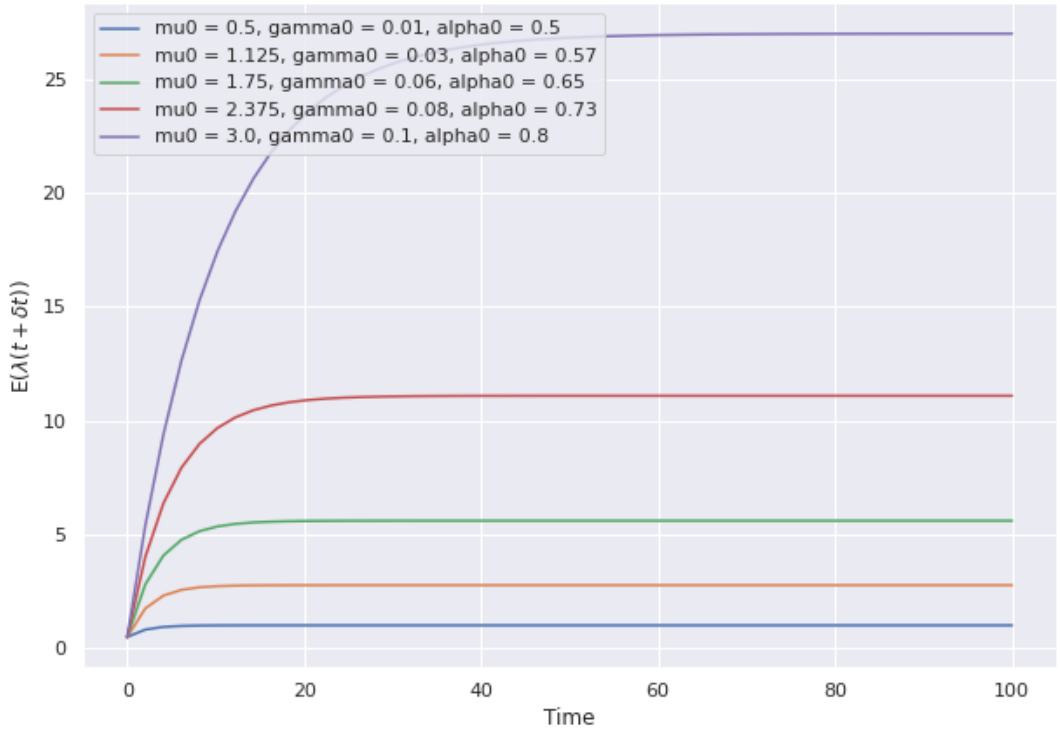


Figure 3.4: $\lim_{k\to\infty} \mathbb{E}(\lambda_k)$ ODE

3.5. Constructing the surrogate function

In order to find a maximum likelihood estimates for our parameters μ, α, γ, we need to find a surrogate function $Q_k(\theta|\theta^n)$, i.e. an upper bound for the negative log-likelihood function. Using θ to represent the three model parameters μ, α, γ, the surrogate function needs to satisfy the two properties mentioned in Chapter 2,

$$-log\big(L(\theta)\big) = Q_k(\theta^n|\theta^n) \tag{3.20}$$

$$-log\big(L(\theta)\big) \leqslant Q_k(\theta|\theta^n) \; for \; any \; \theta, \tag{3.21}$$

where $-log\big(L(\theta)\big) = -\sum_{k=1}^{N} \Delta N_k log(\lambda_k) + \sum_{k=1}^{N} \lambda_k$ from equation 3.7.

Since the model consists of a log-function, the Jensen's inequality 2.6 is used to find an upper bound for each $-log\big(\lambda_k\big)$ that can maximise the negative log-likelihood function. Applying Jensen's inequality for the strictly

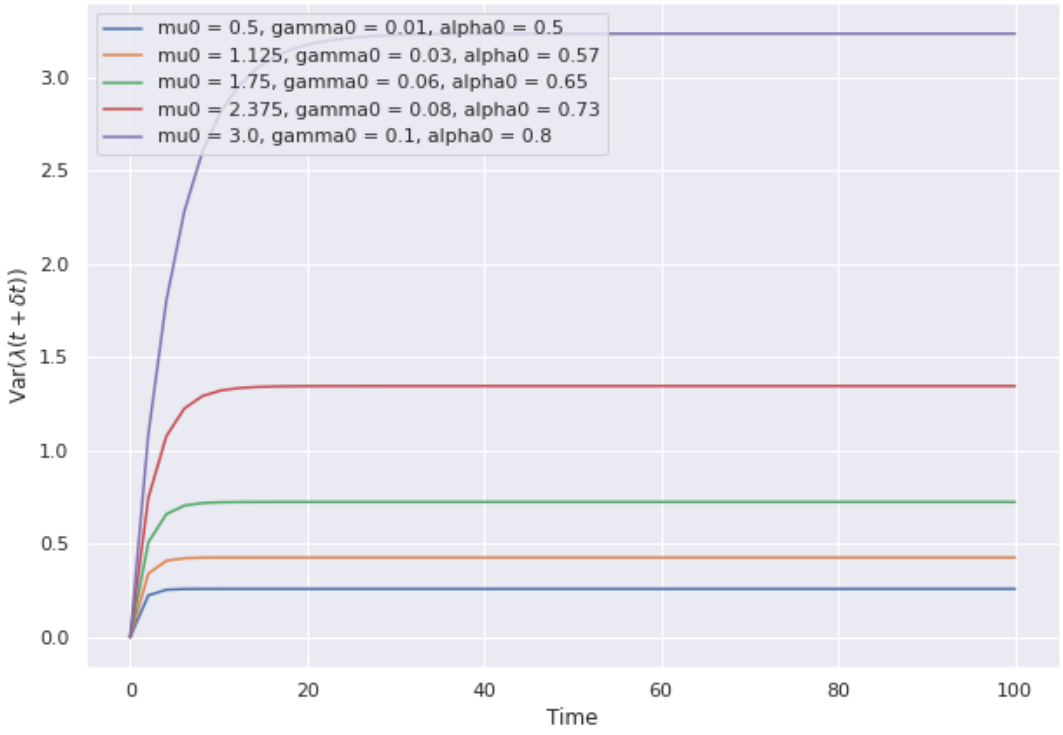


Figure 3.5: $\lim_{k\to\infty} \mathrm{Var}(\lambda_k)$ ODE

convex $-log(\lambda_k)$ *on* $(0,\infty)$ gives the following inequality

$$\begin{aligned} -log(\lambda_k) &= -log\left(\mu + \sum_{l=1}^{k-1} \phi_{kl}\right) \\ &\leqslant -\frac{\mu^{(n)}}{\lambda_k^{(n)}} log\left(\frac{\mu^{(n)}}{\lambda_k^{(n)}}\ \mu\right) - \sum_{l=1}^{k-1} \frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}} log\left(\frac{\lambda_k^{(n)}}{\phi_{kl}^{(n)}}\ \phi_{kl}\right) \\ &= -Q_k\left(\theta|\theta^{(n)}\right), \end{aligned} \tag{3.22}$$

where $\phi_{kl} = \alpha\gamma^{k-l-1}\Delta N_l$. Therefore $\phi_{kl}^{(n)}$ is the n-th iterate of the parameters for the data points ΔN_l. Note

that $\mu^{(n)}$ *and* $\lambda_k^{(n)}$ are also the n-th iterates of each variable. This gives the surrogate function

$$\begin{aligned} Q\Big(\theta|\theta^{(n)}\Big) &= -\sum_{k=1}^{N} \Delta N_k Q_k\Big(\theta|\theta^{(n)}\Big) + \sum_{k=1}^{N} \lambda_k \\ &\geqslant -\sum_{k=1}^{N} \Delta N_k log(\lambda_k) + \sum_{k=1}^{N} \lambda_k \\ &= -log\big(L(\theta)\big) \; for \; any \; \theta. \end{aligned} \tag{3.23}$$

By Jensen's inequality, the domination condition 3.22 is satisfied and we can easily check the inequality 3.23 is also satisfied at the n-th iterate, i.e. $Q(\theta^{(n)}|\theta^{(n)}) = -log\big(L(\theta)\big)$ *at* $\theta = \theta^{(n)}$.

We have used Jensen's inequality to successfully separate the parameter μ. In the first sum of 3.7 we can use the rule of logs to separate the parameters $log(\phi_{kl}) = log(\alpha\gamma^{k-l-1}\Delta N_l) = log(\alpha) + log(\gamma^{k-l-1}) + log(\Delta N_l)$ when updating them. However, the parameters α *and* γ are still conditional on each other in $\sum_{k=1}^{N} \lambda_k$. Expanding and re-arranging the function gives

$$\begin{aligned} \sum_{k=1}^{N} \lambda_k = N\mu &+ (1 + \gamma + \gamma^2 + \cdots + \gamma^{N-2})\alpha\Delta N_1 + \\ &+ (1 + \gamma + \gamma^2 + \cdots + \gamma^{N-3})\alpha\Delta N_2 \\ &+ (1 + \gamma + \gamma^2 + \cdots + \gamma^{N-4})\alpha\Delta N_3 \\ &+ \cdots \\ &+ (1 + \gamma)\alpha\Delta N_{N-2} \\ &+ \alpha\Delta N_{N-1}. \end{aligned} \tag{3.24}$$

In this case, to separate the parameters α and γ, assume that the parameter γ is small. Suppose $\gamma \approx 0$, then the sum of λ_k can also be approximated by

$$\sum_{k=1}^{N} \lambda_k \approx N\mu + (1 + \gamma)\ \alpha\ \big[\Delta N_1 + \cdots + \Delta N_{N-2}\big] + \alpha\Delta N_{N-1}, \tag{3.25}$$

where the values of γ^n for $n > 2$ become insignificant. The approximation helps to split the two parameters as well as using the AM-GM inequality in 2.10. Define $\omega = 1 + \gamma$, such that $1 < \omega < 2$ and use the AM-GM inequality to get the upper bound

$$\omega\alpha \leqslant \frac{\alpha^{(n)}}{2\omega^{(n)}}\ \omega^2 + \frac{\omega^{(n)}}{2\alpha^{(n)}}\ \alpha^2. \tag{3.26}$$

The surrogate function that 'sits above' the negative log-likelihood function becomes

$$Q\Big(\theta|\theta^{(n)}\Big) = -\sum_{k=1}^{N} \Delta N_k Q_k\Big(\theta|\theta^{(n)}\Big) + N\mu + \left(\frac{\alpha^{(n)}}{2\omega^{(n)}}\ \omega^2 + \frac{\omega^{(n)}}{2\alpha^{(n)}}\ \alpha^2\right)\Delta N_s + \alpha\Delta N_{N-1}. \tag{3.27}$$

where $\Delta N_s = \Delta N_1 + \cdots + \Delta N_{N-2}$ and $\omega^{(n)} = 1 + \gamma^{(n)}$.

Minimising the surrogate function is advantageous because all the parameters are separated. So one can find the optimal value for each parameter iteratively by differentiating the function $Q(\theta|\theta^{(n)})$ with respect to each parameter, setting it to zero and finding the update rules for each parameter.

3.6. Estimating the model parameters

To find the optimal parameters, differentiate the surrogate function with respect to each parameter and set it to zero. So the optimal μ is found by differentiating the $Q\big(\theta|\theta^{(n)}\big)$ function with respect to μ

$$\frac{\partial Q\big(\theta|\theta^{(n)}\big)}{\partial \mu} = -\sum_{k=1}^{N} \frac{\partial Q_k}{\partial \mu} \Delta N_k + N \tag{3.28}$$

Differentiating only the function $Q_k\big(\theta|\theta^{(n)}\big)$ gives

$$\begin{aligned} -\frac{\partial Q_k\big(\theta|\theta^{(n)}\big)}{\partial \mu} &= -\frac{\mu^{(n)}}{\lambda_k^{(n)}} \frac{\mu^{(n)}}{\lambda_k^{(n)}} \frac{1}{\mu} \frac{\lambda_k^{(n)}}{\mu^{(n)}} \\ &= -\frac{\mu^{(n)}}{\lambda_k^{(n)}} \frac{1}{\mu} \end{aligned} \tag{3.29}$$

Differentiating the whole surrogate function $Q\Big(\theta|\theta^{(n)}\Big)$ gives

$$\frac{\partial Q\big(\theta|\theta^{(n)}\big)}{\partial \mu} = -\frac{1}{\mu^{(n+1)}} \Big(\sum_{k=1}^{N} \frac{\mu^{(n)}}{\lambda_k^{(n)}} \Delta N_k \Big) + N = 0, \tag{3.30}$$

that yields the following update rule

$$\mu^{(n+1)} = \frac{1}{N} \sum_{k=1}^{N} \frac{\mu^{(n)}}{\lambda_k^{(n)}} \Delta N_k. \tag{3.31}$$

Here μ is replaced by $\mu^{(n+1)}$ as it is the estimate found by setting the differential to zero and re-arranging it.

Similarly, we find the estimate $\alpha^{(n+1)}$ by differentiating $Q\Big(\theta|\theta^{(n)}\Big)$ with respect to α to get

$$\frac{\partial Q\Big(\theta|\theta^{(n)}\Big)}{\partial \alpha} = -\sum_{k=1}^{N} \frac{\partial Q_k\big(\theta|\theta^{(n)}\big)}{\partial \alpha} \Delta N_k + \frac{(1+\gamma^{(n)})}{\gamma^{(n)}} \alpha \Delta N_s + \Delta N_{N-1}. \tag{3.32}$$

Differentiating the $Q_k\Big(\theta|\theta^{(n)}\Big)$ term gives

$$\frac{\partial Q_k\big(\theta|\theta^{(n)}\big)}{\partial \alpha} = \sum_{l=1}^{k-1} \frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}} \frac{1}{\phi_{kl}} \frac{\partial \phi_{kl}}{\partial \alpha}, \tag{3.33}$$

where $\phi_{kl} = \alpha\gamma^{k-l-1}\Delta N_l$. So

$$\frac{\partial \phi_{kl}}{\partial \alpha} = \gamma^{k-l-1}\Delta N_l. \tag{3.34}$$

Hence

$$\begin{aligned}\frac{\partial Q_k\left(\theta|\theta^{(n)}\right)}{\partial \alpha} &= \sum_{l=1}^{k-1} \frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}} \frac{1}{\phi_{kl}} \gamma^{k-l-1}\Delta N_l \\ &= \sum_{l=1}^{k-1} \frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}} \frac{\gamma^{k-l-1}\Delta N_l}{\alpha\gamma^{k-l-1}\Delta N_l} \\ &= \sum_{l=1}^{k-1} \frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}} \frac{1}{\alpha}.\end{aligned} \tag{3.35}$$

The full derivative of the surrogate function becomes

$$\begin{aligned}\frac{\partial Q\left(\theta|\theta^{(n)}\right)}{\partial \mu} &= -\sum_{k=1}^{N}\sum_{l=1}^{k-1} \frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}} \frac{1}{\alpha^{(n+1)}}\Delta N_k + \frac{(1+\gamma^{(n)})}{\alpha^{(n)}}\alpha^{(n+1)}\Delta N_s + \Delta N_{N-1} \\ &= 0\end{aligned} \tag{3.36}$$

Using the quadratic equation to find $\alpha^{(n+1)}$ gives

$$A(\alpha^{(n+1)})^2 + B(\alpha^{(n+1)}) - C = 0 \tag{3.37}$$

where $A = \Delta N_s(1+\gamma^{(n)})/\alpha^{(n)}$, $B = \Delta N_{N-1}$, $C = \sum_{k=1}^{N}\sum_{l=1}^{k-1}\Delta N_k \phi_{kl}^{(n)}/\lambda_k^{(n)}$ are all constants. Solving this using the quadratic formula and taking the positive root only, to satisfy our parameter assumptions that $\alpha > 0$, gives

$$\alpha^{(n+1)} = \frac{-B + \sqrt{B^2 + 4AC}}{2A} > 0. \tag{3.38}$$

So $\alpha^{(n+1)}$ can be updated separately and the assumption that $\alpha^{(n+1)} > 0$ is guaranteed. If $B = \Delta N_{N-1} = 0$ then $\alpha^{(n+1)} = \sqrt{C/A}$, in other words if a data point is zero then calculating $\alpha^{(n+1)}$ becomes easier. However, the drawback of this update rule is that is has a double summation which can be computationally expensive.

The only thing left to find is the update rule for γ. Differentiating the surrogate gives

$$\frac{\partial Q\left(\theta|\theta^{(n)}\right)}{\partial \gamma} = -\sum_{k=1}^{N} \frac{\partial Q_k\left(\theta|\theta^{(n)}\right)}{\partial \gamma}\Delta N_k + \frac{\alpha^{(n)}}{(1+\gamma^{(n)})}(1+\gamma)\Delta N_s. \tag{3.39}$$

Differentiating $Q_k\left(\theta|\theta^{(n)}\right)$ only gives

$$-\frac{\partial Q_k\left(\theta|\theta^{(n)}\right)}{\partial \gamma} = -\sum_{l=1}^{k-1} \frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}} \frac{1}{\phi_{kl}} \frac{\partial \phi_{kl}}{\partial \gamma}, \tag{3.40}$$

where $\partial\phi_{kl}/\partial\gamma = (k-l-1)\gamma^{k-l-2}\alpha\Delta N_l$. So that gives

$$\begin{aligned} -\frac{\partial Q_k\left(\theta|\theta^{(n)}\right)}{\partial\gamma} &= -\sum_{l=1}^{k-1}\frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}}\frac{1}{\phi_{kl}}(k-l-1)\gamma^{k-l-2}\alpha\Delta N_l \\ &= -\sum_{l=1}^{k-1}\frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}}\frac{(k-l-1)\gamma^{k-l-2}\alpha\Delta N_l}{\gamma^{k-l-1}\alpha\Delta N_l} \\ &= -\sum_{l=1}^{k-1}\frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}}\frac{(k-l-1)}{\gamma}. \end{aligned} \tag{3.41}$$

Setting the differentiation to zero gives

$$\frac{\partial Q\left(\theta|\theta^{(n)}\right)}{\partial\gamma} = -\sum_{k=1}^{N}\sum_{l=1}^{k-1}\frac{\phi_{kl}^{(n)}}{\lambda_k^{(n)}}\frac{(k-l-1)}{\gamma^{(n+1)}} + \frac{\alpha^{(n)}}{(1+\gamma^{(n)})}(1+\gamma^{(n+1)})\Delta N_s = 0. \tag{3.42}$$

Similarly update rule for α uses the quadratic formula that gives

$$\gamma^{(n+1)} = \frac{-B+\sqrt{B^2+4AB}}{2B} \geqslant 0, \tag{3.43}$$

where $A = \sum_{k=1}^{N}\sum_{l=1}^{k-1}(k-l-1)\,\Delta N_k\,\phi_{kl}^{(n)}/\lambda_k^{(n)}$ and $B = \Delta N_s\,\alpha^{(n)}/(1+\gamma^{(n)})$. Note that the solution does not guarantee that $\gamma < 1$. So the solution for γ does not meet all model assumptions. Moreover, there is another double summation which can be computationally expensive for the update rule of γ.

Using the MM method to split the model's parameters to find an iterative way to update them is its main the advantage. The method guarantees convergence of the parameter values due to the two properties in 2.1 and 2.2, and it minimises the negative log-likelihood function with each iteration. On the other hand, the process is relatively slow when there are a large number of data points, and it doesn't guarantee convergence to the global minimum. In order to find the optimal parameters, one should run the update rules for a range of parameter values, but that can be computationally expensive and time-consuming. One way to reduce the time to find the parameter estimates is to use parallel computing to update each parameter value separately.

The coded function for finding the parameter estimates is in the appendix A.22.. The functions ran on a Google Colaboratory notebook and the double summation ran relatively quickly for 100 iterations with 5,000 data points. The parameter updates decreased the negative log-likelihood with each iteration as shown in fig, 3.6 as the property was guaranteed by the decent property of the MM method in 2.3. Fig. 3.7 shows that as the parameter error decreases so does the negative log-likelihood. The smallest negative log-likelihood is around $-73,091$ for 93 iterations and converges to $-73,093$ after 151 iterations starting from $-70,589$, whereas the smallest negative log-likelihood we found in 3.3a was $Lmin = -78,364$ and the best estimates for the model occurred when the negative log-likelihood is between the range $-74,446$ to $-78,364$. Therefore, the negative log-likelihood might have converged to a local minimum, so the parameter estimates from this run might not be optimal.

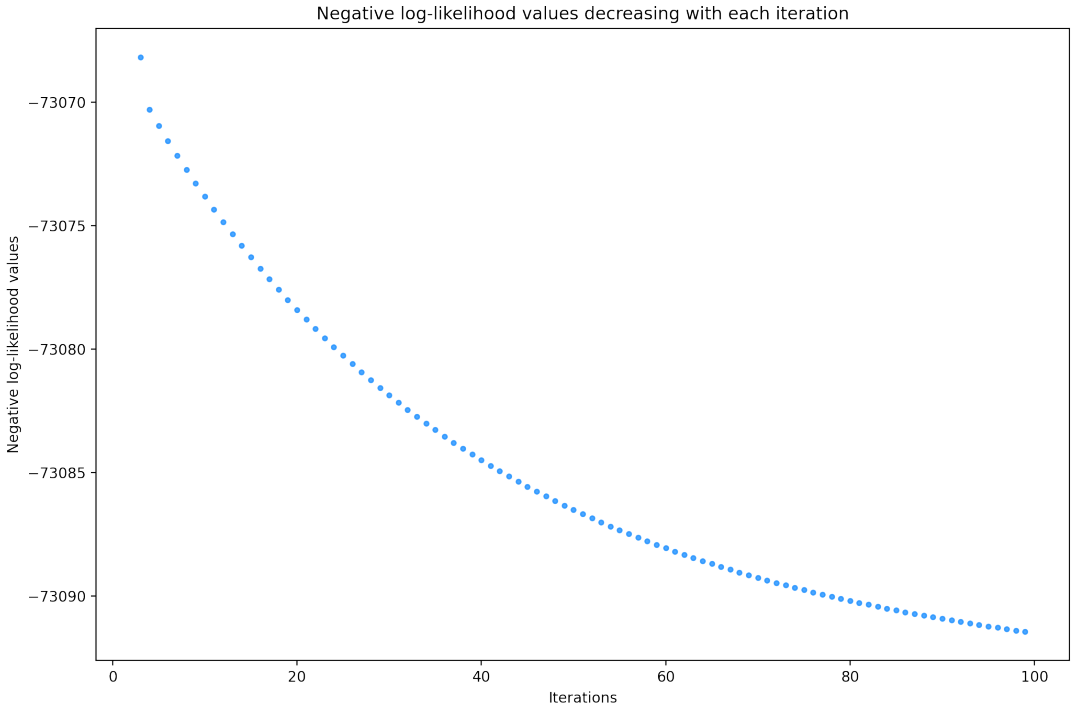


Figure 3.6: The negative log-likelihood decreases after each parameter iteration. (Starting from iteration 2).

Running the parameter update functions for different initial values sometimes gives null returns after some iterations. This is caused by the unbounded parameter γ. Once the parameter is greater than 1, the other parameters quickly converge to zero as γ continues to grow. We initially assumed that $0 < \gamma < 1$, but when finding the parameter update equation, that was not easy to factor in. Using the Lagrange multipliers method would not work with the strict inequality, and other methods would have taken more time to work out. Therefore, a simple fix for this at the moment could be increasing the initial α_0 value and running the update functions for a range of model parameters until it reaches relatively low negative log-likelihood, as well as a small γ.

The cost of iterations is the biggest drawback of the MM method. One might use a more powerful computer or use parallel computing to update each parameter separately. Note that even with parallel computing, the parameter γ is not guaranteed to stay below 1, and the negative log-likelihood is not guaranteed to converge to the global minimum.

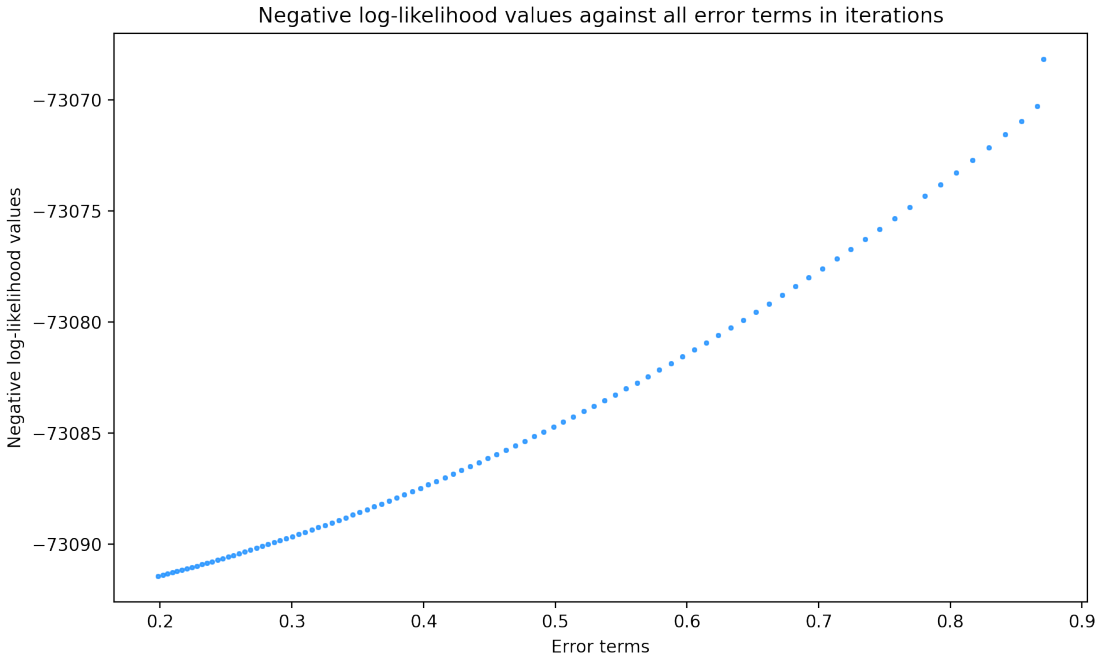


Figure 3.7: Negative log-likelihood decreased as the parameter error becomes smaller. (Starting from iteration 2).

3.7. COVID-19 data analysis

This section applies the model for the discrete Hawkes process to real-world data. The data is taken from the UK government's website, https://coronavirus.data.gov.uk/details/cases. It contains the number of new, daily COVID-19 cases, given by specimen date for the number of people with at least one positive COVID-19 test result, either lab-reported or rapid lateral flow test.

The start date for the data is Monday the 9th of September 2020, because on this date all children in the UK would have returned to school. IN addition, the website has the plotted data which shows the daily number of new cases increase just after children return back to school and the temperatures decrease. We can see the spike of new COVID-19 cases just after that time period, whereas the new case rates were flat, around 1,000 new cases a day during the summer of 2020. The data continues for 35 weeks until Sunday the 9th of May 2021. The data points were all above zero, which is desirable for running the update functions.

My supervisor suggested using the average number of new COVID-19 cases for that data as the initial μ_0. I ran the update function for a range of values for $\alpha = \{0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 3, 5\}$ with the same

initial value $\mu_0 = 16665.1879$ and $\gamma = 0.01$. The parameter values μ and α tend to decrease and even go to zero while γ increases when starting with an initial value $\alpha_0 < 0.5$. This could be because of the rapid increase in COVID-19 cases during the winter, meaning the infectious rate should be a larger value. Also, the negative log-likelihood 'explodes' for values of $\alpha_0 < 0.5$. We see this in fig. 3.8, where the negative log-likelihood increases exponentially as the γ term goes beyond 1. However, as the the initial parameter value for α_0 increases while the others stay fixed, the negative log-likelihood decreases. The results for the fixed μ_0 and γ_0, and increasing α_0 are summarised in table 3.1.

Parameters $\mu_0 = 16,665$, $\gamma_0 = 0.01$ and changing α_0	Parameters after 50 iterations	$-log\big(L(\boldsymbol{\theta})\big)$
$\alpha_0 = 0.4$	$\mu_{50} = 110.8685$,$\alpha_{50} = 0.0046$ $\gamma_{50} = 946.2102$	$infinity$
$\alpha_0 = 0.5$	$\mu_{50} = 1068.0696$,$\alpha_{50} = 0.9249$ $\gamma_{50} = 0.0193$	$-5,261,006.8524$
$\alpha_0 = 0.6$	$\mu_{50} = 1066.0088$,$\alpha_{50} = 0.9367$ $\gamma_{50} = 0.0064$	$-5,261,340.1447$
$\alpha_0 = 0.7$	$\mu_{50} = 1062.2640$,$\alpha_{50} = 0.9397$ $\gamma_{50} = 0.0035$	$-5,261,418.2127$
$\alpha_0 = 0.8$	$\mu_{50} = 1059.4260$,$\alpha_{50} = 0.9411$ $\gamma_{50} = 0.0023$	$-5,261,452.0182$
$\alpha_0 = 0.9$	$\mu_{50} = 1057.2070$,$\alpha_{50} = 0.9420$ $\gamma_{50} = 0.0016$	$-5,261,470.6408$
$\alpha_0 = 1$	$\mu_{50} = 1055.3961$,$\alpha_{50} = 0.9425$ $\gamma_{50} = 0.0012$	$-5,261,482.3519$
$\alpha_0 = 2$	$\mu_{50} = 1045.7261$,$\alpha_{50} = 0.9441$ $\gamma_{50} = 0.0003$	$-5,261,515.4963$

Table 3.1: Results from iterations for an increasing range of α initial values, fixed μ_0 and γ_0

To summarise, the update functions do not guarantee that the parameter γ will stay within its bounded region. The negative log-likelihood decreases with each iteration, but note that the MM method does not guarantee that it will converge to the global minimum. The most significant disadvantage of the MM method is the cost of iterations. In addition, finding the optimal parameter values will become more expensive as the dataset becomes larger. However, in this example the dataset is quite small with 35 values. That might not be enough to estimate the model parameters for the model.

On the other hand, the MM method provided an excellent way of separating the model parameters and finding update equations for each one. Different numerical methods such as the Newton-Raphson or the Gradient descent method could be used to find the update rules, using the surrogate function to make it more efficient. In addition, parallel computing can be employed to split the parameter update functions to speed up computation for larger datasets or more iterations.

Furthermore, the Hawkes process used in this report might be too simplistic to model the UK's new COVID-19 cases. The process can be extended to a multi-dimensional Hawkes process to model the new COVID-19 case spikes in a region or country. One might even want to add parameters based on the restrictions in an area, such as the model built by Imperial College London has done [33]. However, the simple 1-dimensional Hawkes

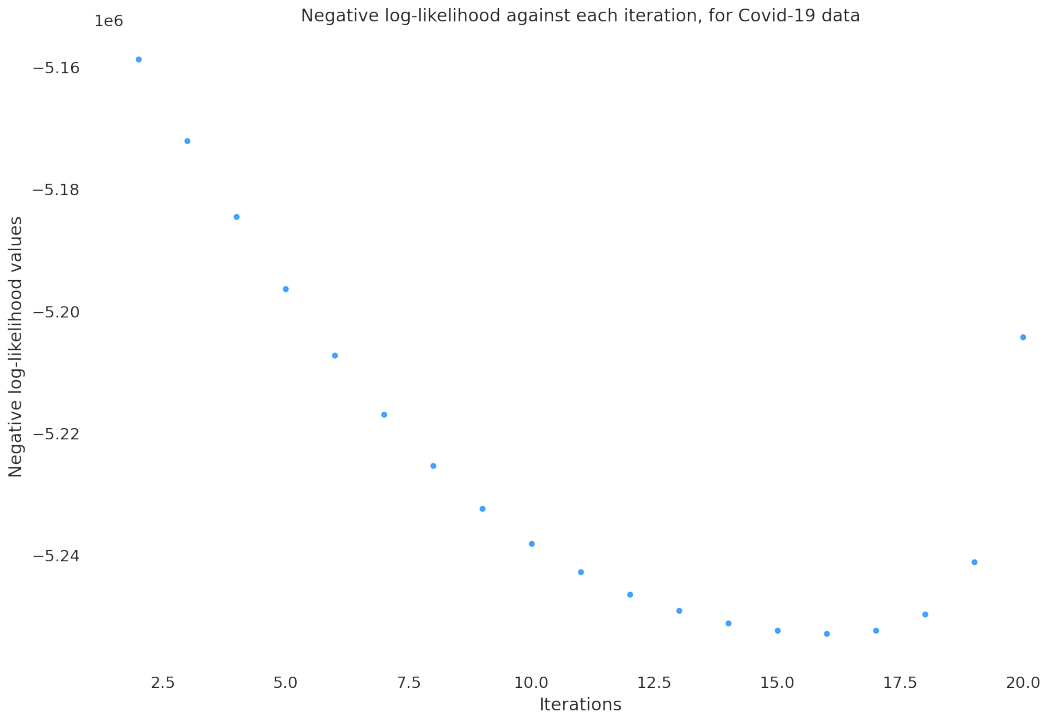


Figure 3.8: Negative log-likelihood decreases until the parameter values go outside their bounded regions.

process we have looked at is far too simple to model the COVID-19 data.

4

Conclusions and Outlook

In Chapter 2, I described the advanced optimisation technique, the MM method, and its special case, the EM algorithm. I start with the MM method because it is more general and is easier to understand. I studied ways of constructing surrogate functions through inequalities. The most famous of which is the Jensen inequality which is the building block to other inequalities, and is critical to constructing surrogate functions. I also explored the advantages and disadvantages of the MM method using some simple applications such as finding the sample median, linear regression using ordinary least squares and least absolute deviation. Some of the main advantages of the MM method are that

- It avoids large matrix inversions such as in the OLS example;
- Separates the variables of an optimisation problem as in the Hawkes process example;
- Turns a non-differentiable problem into a smooth one as with finding the sample median example.

However, the biggest drawback of the MM method is the price of iteration. The MM method also tends to have a slower convergence rate. In all of the examples, the cost functions began decreasing slowly after only 50 iterations. Therefore, running the algorithms for a range of initial values could be costly and time-consuming.

In Chapter 3, I learnt about the Hawkes process for count data. I learnt about a simple discrete case of the process that can be used when the data has no time stamps. I expanded my knowledge of the difference between a Poisson (memoryless) process and the Hawkes process which dependent on past counts. I then solidified my understanding of the MM method by applying it to the Hawkes process problem to separate the model parameters using Jensen's inequality and the AM-GM inequality. The outcome of the update functions for the generated data did not reach the desirable range of negative log-likelihood values but running the functions enough times could bring the negative log-likelihood and parameter error values down.The main point to note is that the update functions does not bound the parameter γ from above. If γ goes beyond 1, the next iterations send the other parameter values to zero, while the negative log-likelihood 'explodes'. The PCA analysis suggests

that an increase in one of γ or α would decrease the other. Therefore the simple fix that could stop the γ parameter from going beyond its bounded region was to increase the initial value of α. However, more could be done to investigate this, as well as to find out how much α_0 should be increased by, to get sensible estimates for the parameters.

In addition, I expanded my knowledge of Python programming, as I used packages such as Numpy, Matplotlib, Scikit-learn, Seaborn and Scipy to do the initial analysis, make plots and write functions.

However, this report is not exhaustive. Further investigations into the Hawkes process can be done. For example, the model can be extended to a multi-dimensional Hawkes process that could better model the arrival of new COVID-19 cases. Moreover, an EM method can be used to estimate the model parameters of the Hawkes process. In this case, we can assume a latent variable exists, as with the GMM example and create a new surrogate function to minimise. The EM method can also be extended into a multi-dimensional case and can also be compared to the MM method. In fact, more can be done to compare the rate of convergence and the computation time of the update functions of the MM method with traditional approaches such as the Newton-Raphson method, the gradient descent method, matrix inversion, etc.

Moreover, more could be done to find out how much data is enough to get the optimal parameter estimates. Note that we assume that γ is relatively small and disregard any powers of γ when finding the update functions. So one can further investigate how small should γ be for this assumption to hold. In addition, the assumption could be a reason why the one-dimensional fit does not model the arrival of new COVID-19 cases well. Consequently, one could find an alternative method to finding the surrogate function without approximating the parameter $\gamma \approx 0$.

Furthermore, one can try to create an update function that bounds the parameter γ so that the negative log-likelihood does not explode for smaller values of α_0. Additionally, the update functions can be written to use with parallel computing, which would speed up the computation for large datasets or many iterations. Also, the functions can be written using a *while - loop* instead of a *for-loop* as I have done, where the iterations can run until the difference between the new negative log-likelihood and the previous one is less than some ϵ.

In summary, I explore the MM and EM methods for optimisation problems and look at a simple application of the MM method. This helped to understand the advantages and disadvantages of the methods as well as expanded my programming skills in Python.

Appendices

A

Appendix

A.1. Section 2.3.2. theorem

Theorem A.1. *The Median as an Error Minimizer*
The median $\mu_{1/2}$ of a real valued random variable X minimizes the function

$$\min_{c\in\Re} E[|X-c|], \tag{A.1}$$

for some constant $c \geqslant 0$. In other words the sum of the absolute deviations of the data from X is minimized when X is a median of the data.

Proof: We can define the median as any number $\mu_{1/2} \in \Re$ such that it satisfies the following inequalities

$$p(X \leqslant \mu_{1/2}) \geqslant \frac{1}{2} \text{ and } p(X \geqslant \mu_{1/2}) \geqslant \frac{1}{2}. \tag{A.2}$$

Let $p(x)$ denote the probability density function of X with a cumulative distribution function $F(x)$. The expectation of $|X-c|$ is

$$E[|X-c|] = \int_{x\in\Re} p(x)|x-c|dx = \int_{-\infty}^{\infty} p(x)|x-c|dx$$

We can split the integral into two

$$E[|X-c|] = \int_{-\infty}^{c} p(x)|x-c|dx + \int_{c}^{\infty} p(x)|x-c|dx \tag{A.3}$$

To find the value of c for which the expectation function is minimized, we differentiate the function A.3 with respect to c and set it to zero.

$$\frac{dE|X-c|}{dc} = \frac{d}{dc}\int_{-\infty}^{c} p(x)|x-c|dx + \frac{d}{dc}\int_{c}^{\infty} p(x)|x-c|dx$$

Note that $|x-c| = (x-c)$ when $x \geqslant c$ and $|x-c| = (c-x)$ when $x \leqslant c$ so we can re-write the above as

$$\frac{dE|X-c|}{dc} = \frac{d}{dc}\int_{-\infty}^{c} p(x)(c-x)dx + \frac{d}{dc}\int_{c}^{\infty} p(x)(x-c)dx$$

Using Leibniz's rule for differentiation under the integral sign we get

$$\frac{d}{dc}\int_{-\infty}^{c} p(x)(c-x)dx = p(x)(c-c)\frac{dc}{dc} + \int_{-\infty}^{c} \frac{\partial}{\partial c}\left[p(x)(c-x)\right]dx = \int_{-\infty}^{c} p(x)dx$$

$$\frac{d}{dc}\int_{c}^{\infty} p(x)(x-c)dx = -p(x)(c-c)\frac{dc}{dc} + \int_{c}^{\infty} \frac{\partial}{\partial c}\left[p(x)(x-c)\right]dx = -\int_{c}^{\infty} p(x)dx$$

Hence we get the differential and set it to zero

$$\frac{dE|X-c|}{dc} = \int_{-\infty}^{c} p(x)dx - \int_{c}^{\infty} p(x)dx = 0$$

This gives us the following expression

$$\int_{-\infty}^{c} p(x)dx = \int_{c}^{\infty} p(x)dx$$

The integrals are simply the following probabilities

$$p(x \leqslant c) = p(x \geqslant c).$$

Comparing the results with A.2 we get that

$$p(x \leqslant c) = p(x \geqslant c) = \frac{1}{2}. \tag{A.4}$$

Hence so $c = \mu_{1/2}$ minimizes the absolute deviations of the data from X. □

You can also find the code for Chapter 2 and Chapter 3 on https://github.com/Ivelina0/Dissertation_code.

A.2. Fig 2.1 Convex chart

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# initial values of chord
x_1 = -0.8
x_2 = 0.7

y_1 = np.exp(x_1) + (x_1)**2
y_2 = np.exp(x_2) + (x_2)**2

# gradient of a simple linear equation
m = (y_2 - y_1)/(x_2 - x_1)

###################### Graph Set Up ######################

# Return evenly spaced values within [-1,1] with 0.01 step.
x = np.arange(-1.0, 1.0, 0.01)
y1 = np.exp(x) + x**2
y2 = x + 1

# Return evenly spaced values within [-0.8,1] with 0.75 step.
x2 = np.arange(-0.8, 1.0, 0.75)
y = m*x2 - m*x_1 + y_1

fig, ax = plt.subplots()

# Plot data on the (implicit) axes.
line1 = plt.plot(x, y1, label='f(x)', color='black',linewidth=1)
line2 = plt.plot(x, y2, '--', label='tangent',color='black',linewidth=1)
line3 = plt.plot(x2, y, label='chord', color='black', linewidth=1)
```

```
plt.xlim((-1,1)) # limit x axis
plt.ylim((0,4)) # limit y axis

fig.savefig('Convex Function Example.png')
```

A.3. Fig 2.2 (a) cos(x)

```
# inital guess
x_n = 0.1

# list of x_n values as they iterate
x_n_update = []
# list of update of cos(x) function with each iterate
cos_x_n_update = []

for i in range(10):

    # Calculate values of x_n and cos(x)
    x_n = x_n + np.sin(x_n)
    cos_x_n = np.cos(x_n)

    # append values to empty list
    x_n_update.append(x_n)
    cos_x_n_update.append(cos_x_n)

## --- Plotting for a few values of x_n ----

# X axis set up
x = np.arange(0, 4*np.pi, 0.01)

# The original function we want to minimise
f_x = np.cos(x)

# Take two iterations of the MM approach above
# and plug them into the surrogate function
```

```
x_3 = x_n_update[2]
x_4 = x_n_update[3]

# two surrogate functions
g_x3 = np.cos(x_3) - np.sin(x_3)*(x - x_3) + (1/2)*(x - x_3)**2

g_x4 = np.cos(x_4) - np.sin(x_4)*(x - x_4) + (1/2)*(x - x_4)**2

# Create a figure and an axes.
fig1, ax = plt.subplots()

# Plot the f(x) function
line1, = ax.plot(x, f_x, label='cos(x)', color='black')
# Plot third iteration
line2, = ax.plot(x, g_x3, '--', label='g(x3)', color= 'grey')
# Plot fourth iteration
line3, = ax.plot(x, g_x4, ':', label='g(x4)', color='grey')

ax.set_xlabel('x ') # Add an x-label to the axes.
# ax.set_ylabel('y label') # Add a y-label to the axes.
# ax.set_title("Cos(x) Plot") # Add a title.

ax.legend() # Add a legend.
ax.set_xlabel('(a) Minimisation method for
                f(x) = cos(x) \n
                for $x \in [0, 2\pi]$')

plt.xlim((0,np.pi*2)) # limit x axis
plt.ylim((-1,1)) # limit y axis

fig1.savefig('Cosx.png')
```

A.4. Fig. 2.2 (b) Quadratic function

```
x1 = np.arange(0,8)
y1 = np.array([20, 12, 6, 5, 8, 13, 15, 20])

x2 = np.linspace(0,8,100)
y2 = 6*x2**2 - 30*x2 + 6*2.5**2 + 6

fig2, ax = plt.subplots()

# f(x)
line1, = ax.plot(x1, y1,'o-', label='f(x)',color='black')

# Surrogate g(x)
line2, = ax.plot(x2-0.32, y2+0.5,dashes=[3, 2], label='g(x)',color='grey')
ax.set_xlabel("(b) A quadratic majorising function
                \n for the piecewise linear function \n
                f($\mu$) = |x-20| + |x-12| + |x-6| + |x-5| + |x-8| + |x-10|")


plt.xlim((0,5)) # limit x axis
plt.ylim((4,20)) # limit y axis

ax.legend()

fig2.savefig('Simple Example 1.png')
```

A.5. Table 2.1 Finding the sample median

```
# Sample list
x1 = np.array([1,3,4,8,10,11,15])

# The actual median
med = np.median(x1)

# Initial guess
theta_update = 6

# Count iterations
b = 0

# Empty list
theta_update_list = []

# for loop that finds the median of the sample
for i in range(9):

    # weights = 1/np.absolute(x1-theta)

    theta_update = (np.dot((1/np.absolute(x1-theta_update))
    ,x1))/sum((1/np.absolute(x1-theta_update)))

    theta_update_list.append(theta_update)

    f_x = sum(np.absolute(x1-theta_update))

    b = i+1

    print("After {} iteration: theta_k+1 = {}, and f(theta) = {}.
    The real median is {}".format(b, theta_update, f_x, med))
```

A.6. Fig. 2.3a OLS and fig. 2.3b LAD examples

```
import pandas as pd
import seaborn as sns
sns.set()

from sklearn.linear_model import LinearRegression

## Data Cleaning

first_time_buyers = pd.read_csv('First-Time-Buyer-Former-Owner-Occupied-2020-
10.csv')

# Limiting the data to only the region name and 1st time house buyers
first_time_buyers_2 = first_time_buyers[['Date',
'Region_Name','First_Time_Buyer_Average_Price']]

# Dropping all values with dates greater than the 1st Jan 2020
# and less than or equal to the 1st May 2019

first_time_buyers_3 = first_time_buyers_2[ first_time_buyers_2['Date'] <
'2020-01-01'].dropna()

first_time_buyers_4 = first_time_buyers_3[ first_time_buyers_3['Date'] >=
'2019-05-01'].dropna()

# Check the dates are correct
first_time_buyers_4.groupby(['Date']).mean()

# Group the home buyers by region and find the mean of the house price
## for 2020 year. Getting rid of dates

first_time_buyers_5 = first_time_buyers_4.groupby(['Region_Name'
]).mean().reset_index()
```

```
# Assign a new the column names: from Region_Name to
# Region, and set index Region for ease when joining with other data
first_time_buyers_5.columns = ['Region', 'First_Time_Buyer_Average_Price']

first_time_buyers_5= first_time_buyers_5.set_index('Region')

first_time_buyers_5.head()

# Adding unemployment data
## ------------------------

unempl = pd.read_csv('Employment_June_to_August_2019.csv')

unempl.columns
# ['Region', 'Employment_16 _to_64_years',
# 'Unemployment_16_years_and_over',
# 'Inactivity rate_16_to_64_years']

unempl = unempl.set_index('Region')

## Merge 2 datasets

df_both = pd.merge(first_time_buyers_5, unempl, how='inner', on=['Region'])

df_both.columns
## Index(['First_Time_Buyer_Average_Price', 'Region',
##        'Employment_16 _to_64_years', 'Unemployment_16_years_and_over',
##        'Inactivity rate_16_to_64_years'],
##       dtype='object')

# Define arrays for plotting functions
## ------------------------------------

frist_time_avr_price_array = df_both['First_Time_Buyer_Average_Price'].to_numpy()

employ_array = df_both['Employment_16_ _to_64_years'].to_numpy()
```

```
unemploy_array = df_both['Unemployment_16_years_and_over'].to_numpy()

# plot a simple scatter to get a feel for the data
plt.scatter(np.log(frist_time_avr_price_array), np.log(employ_array),
color='gray');

# Preparing data for regression
## --------------------------------

x_vals = np.log(employ_array)
y_vals = np.log(frist_time_avr_price_array)

x_vals = np.reshape(x_vals, (-1,1))
y_vals = np.reshape(y_vals, (-1,1))


#####################################################################
## OLS Regression Example: Fig. 2.3 (a)
#####################################################################

# add ones to 1st column of X
X = np.append(np.ones((x_vals.shape[0],1)), x_vals,axis = 1)
X_T = X.T

# p-value is 1

norm = np.linalg.norm(X_T, ord=1, axis=1).reshape(-1,1)
# row wise norm

alpha = np.absolute(X_T)/norm

A = np.sum(np.square(X_T)/alpha, axis=1)

A = A.reshape(2,1) ## bottom vector in calc
```

```
# A is A for all

# initial values of parameters
beta = np.array([[-2],[2]])

#iterations counter
b = 0

# no. of iterations
n=50

for i in range(0,n):

    # prediction values
    pred = np.matmul(X_T.T, beta)

    # find residuals
    res = np.matmul(X_T, (y_vals - pred))

    # the parameter update
    beta = res/A + beta

    # calculating the least squares we want to minimise
    least_squares =  np.sum((y_vals -  np.matmul(X_T.T, beta))**2 )

    b = i+1

    print('For i = {}, beta is {}, the least squares are
    {}'.format(b,beta, np.round(least_squares,2)))

# fitted values using the parameters above
pred = beta[1]*x_vals + beta[0]

# Plot the chart
## -----------------------------
fig = plt.figure(figsize=(8,5))
```

```
plt.scatter(x_vals, y_vals, color='black', marker = 'o')
plt.plot(x_vals, pred,  color='red', linewidth = 2)

plt.xlabel("Log of % of employed people aged 16 to 64, in the UK")
plt.ylabel("Log of the average house price for a first time buyer")

# download image
fig.savefig('LAD_with_London_outlier.png', dpi=300)

# evaluation
RSS1 = sum( (y_vals - pred)**2 )

#####################################################################
## LAD Regression Example:  Fig. 2.3 (b)
#####################################################################

# add ones in the column of X
X = np.append(np.ones((x_vals.shape[0],1)), x_vals,axis = 1)

# initial guess
beta_update = np.zeros(X.shape[1])

# create an empty matrix to be filled by the weights matrix
M = np.zeros((11,11))

# counter
b=0

# set the no. of iterations
n= 50

for i in range(n):

    # create diagonal matrix
    W = 1/np.absolute(y_vals.T - np.matmul(beta_update.T,X.T))
```

```
    np.fill_diagonal(M, W)

    # inversing the matrix
    inv = np.linalg.inv(np.matmul(np.matmul(X.T,M),X))

    # calculating the theta update
    beta_update = np.matmul(np.matmul( np.matmul(inv,X.T),M) , y_vals)

    least_abs_dev=  np.sum(np.absolute((y_vals -  np.matmul(X_T.T, beta_update))))

    b = i+1

    print("Iteration {} gives beta {}, the least absolute deviation is
    {} ".format(b, np.round(beta_update,2), np.round(least_abs_dev,4)))

## fitted values using the parameters above
pred = beta_update[1]*x_vals + beta_update[0]

## Plot the graph
fig = plt.figure(figsize=(8,5))

plt.scatter(x_vals, y_vals, color='black', marker = 'o')
plt.plot(x_vals, pred,  color='red', linewidth = 2)

plt.xlabel("Log of % of employed people aged 16 to 64, in the UK")
plt.ylabel("Log of the average house price for a first time buyer")

# save image
fig.savefig('LAD_with_London_outlier.png', dpi=300)
```

A.7. Fig. 2.5 GMM Example 1

```
import scipy.stats as stats

# Define the 3 means in the first array and the
# 3 standard deviations in the second
param = np.array([[-3,0,8], [2,1,3]])

# Create the x-axis, 3 s.t. away from the first mean (-3),
# and 3 s.t. from the last mean (8)
x = np.linspace(param[0][0] - 3*param[1][0], param[0][2] + 3*param[1][2], 100)

# Use scipy's stats function to create the gaussian distributions
y = 0.3*stats.norm.pdf(x, param[0][0], param[1][0]) +
    0.3*stats.norm.pdf(x, param[0][1], param[1][1]) +
    0.4*stats.norm.pdf(x, param[0][2], param[1][2])

# Set colours for the normal distributions
col = ["dodgerblue", "crimson", "limegreen"]

# Plot the mixtures
fig = plt.figure( figsize = (12,8))

for i in range(3):

  plt.plot(x, stats.norm.pdf(x, param[0][i], param[1][i]),
           color=col[i], linestyle='dashed',
           label = "$ \mu $={},
           $\sigma^2 $={}".format(param[0][i], np.square(param[1][i])))

  plt.legend()

plt.plot(x, y, color="black", label = "GMM density")

plt.ylabel("Probability density $p(x| \theta)$")
```

```
plt.xlabel("No. of standard deviations away from the means")
plt.legend()
plt.title("Gaussian Mixture Model");

fig.savefig( "/content/gdrive/My Drive/Colab Notebooks/GMM.png" ,
    dpi=200,transparent=True)
```

A.8. Fig. 2.6 GMM Example 2

```
## Functions from
#https://github.com/joferkington/oost_paper_code/blob/master/error_ellipse.py

import numpy as np

import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse

def plot_point_cov(points, nstd=2, ax=None, **kwargs):
    """
    Plots an `nstd` sigma ellipse based on the mean and covariance of a point
    "cloud" (points, an Nx2 array).
    Parameters
    ----------
        points : An Nx2 array of the data points.
        nstd : The radius of the ellipse in numbers of standard deviations.
            Defaults to 2 standard deviations.
        ax : The axis that the ellipse will be plotted on. Defaults to the
            current axis.
        Additional keyword arguments are pass on to the ellipse patch.
    Returns
    -------
        A matplotlib ellipse artist
    """
    pos = points.mean(axis=0)
    cov = np.cov(points, rowvar=False)

    return plot_cov_ellipse(cov, pos, nstd, ax, **kwargs)

def plot_cov_ellipse(cov, pos, nstd=2, ax=None, **kwargs):
    """
    Plots an `nstd` sigma error ellipse based on the specified covariance
    matrix (`cov`). Additional keyword arguments are passed on to the
```

```
    ellipse patch artist.
    Parameters
    ----------
        cov : The 2x2 covariance matrix to base the ellipse on
        pos : The location of the center of the ellipse. Expects a 2-element
            sequence of [x0, y0].
        nstd : The radius of the ellipse in numbers of standard deviations.
            Defaults to 2 standard deviations.
        ax : The axis that the ellipse will be plotted on. Defaults to the
            current axis.
        Additional keyword arguments are pass on to the ellipse patch.
    Returns
    -------
        A matplotlib ellipse artist
    """
    def eigsorted(cov):

        vals, vecs = np.linalg.eigh(cov)
        order = vals.argsort()[::-1]

        return vals[order], vecs[:,order]

    if ax is None:
        ax = plt.gca()

    vals, vecs = eigsorted(cov)
    theta = np.degrees(np.arctan2(*vecs[:,0][::-1]))

    # Width and height are "full" widths, not radius
    width, height = 2 * nstd * np.sqrt(vals)
    ellip = Ellipse(xy=pos, width=width, height=height, angle=theta,
    edgecolor='#292323',lw=2, facecolor='none', **kwargs)

    ax.add_artist(ellip)
    return ellip
```

```
#################################################################
## Plotting the image
#################################################################

# Getting the first multivariate Gaussain points
points = np.random.multivariate_normal(mean=(1,5), cov=[[7, 1],[1, 0.5]],
size=100)

# transposing them
x, y = points.T

# Getting the secnond multivariate Gaussain points
points_2 = np.random.multivariate_normal(mean=(3,1), cov=[[3, 1],[1, 1.5]],
size=100)

# transposing them
x2, y2 = points_2.T

fig = plt.figure(figsize=(12,10))

# Plot the raw points...
ax = plt.plot(x, y, 'x',color = 'purple')
ax = plt.plot(x2, y2, 'x',color = 'green')

# Plot a transparent 3 standard deviation covariance ellipse
ax = plot_point_cov(points, nstd=2.7)
ax = plot_point_cov(points, nstd=2.0)
ax = plot_point_cov(points, nstd=1.0)
ax = plot_point_cov(points, nstd=0.5)

ax = plot_point_cov(points_2, nstd=2.7)
ax = plot_point_cov(points_2, nstd=2.0)
ax = plot_point_cov(points_2, nstd=1.0)
ax = plot_point_cov(points_2, nstd=0.5)
```

```
plt.xlim(-8,10)
plt.ylim(-2.5,7)
```

A.9. Hawkes: Generating synthetic data

```
import numpy as np
import random
import matplotlib.pyplot as plt
%matplotlib inline

from mpl_toolkits import mplot3d
from mpl_toolkits.mplot3d import Axes3D

# Create synthetic data function

def generate_data(n, rnd_seed, mu, gamma, alpha ):

    # fixing the randomness
    np.random.seed(rnd_seed)

    Lambda_ = np.zeros(n+1)

    # First lambda(0) = mu
    Lambda_[0] = mu ## adding the first lambda_k = mu

    # Delta_n and lambda_n empty lists
    Delta_n = np.zeros(n) ## adding a dummy data point

    #Generating the lambda function and synthetic data
    for i in range(0,n):

        # draw from poisson distribution
        Delta_n[i] = np.random.poisson(Lambda_[i], size=1)

        # update lambda k+1
        Lambda_[i+1] = gamma*(Lambda_[i] - mu) + mu + alpha*Delta_n[i]

    return Lambda_, Delta_n
```

A.10. Hawkes: Fig. 3.2 The step and bar plots

```
##############################################################################
# Creating fig. 3.2: the step function and bar plot
##############################################################################

# Setting the no. of data points
n0 = 100
rnd_seed = 4 # random seed set

# Setting the parameters
mu0 = 8
gamma0 = 0.1
alpha0 = 0.25

Lambda0_, Delta0_n = generate_data(n0, rnd_seed, mu0, gamma0, alpha0)

fig = plt.figure( figsize = (14,12))

# subplot 1
plt.subplot(211)

x1 = range(1,n0+1)

plt.bar(x1, Delta0_n[0:] , width=0.5, color = 'royalblue')

plt.title(r'$\Delta N_k$ scatter plot for $ \mu = {} , \ \gamma = {} \
and \ \alpha = {} $'.format(mu0,gamma0, alpha0))
plt.xlabel('Time')
plt.ylabel('$ \Delta N_k$')

# subplot 2
plt.subplot(212)

# Step function
x2 = range(1,n0+2)
```

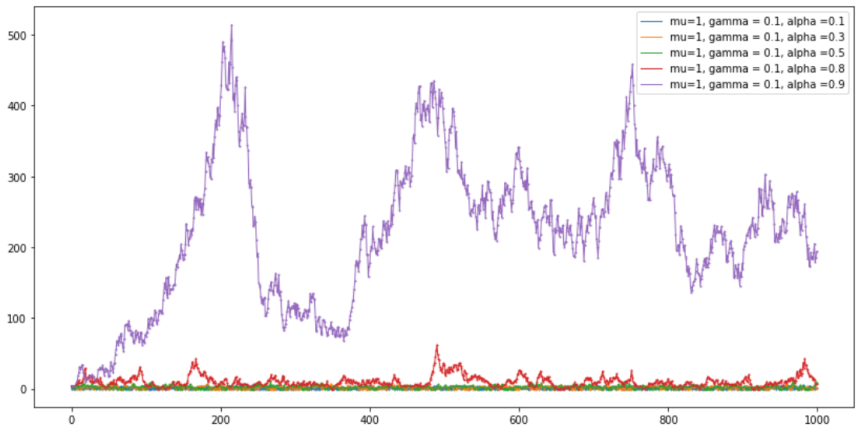
```
plt.step(x2, Lambda0_, color='darkblue', where='mid' , label = 'mid')
plt.plot(x2, Lambda0_, 'o--', color='grey', alpha=0.3)

plt.title(r'$\lambda␣$␣scatter␣plot␣for␣$␣\mu␣=␣{}␣,␣\␣\gamma␣=␣{}
␣␣␣␣\␣and␣\␣\alpha␣=␣{}␣␣$'.format(mu0,gamma0, alpha0))
plt.xlabel('Time')
plt.legend()
plt.ylabel('$␣\lambda$');
```

A.11. Hawkes: Additional plots of the synthetic data

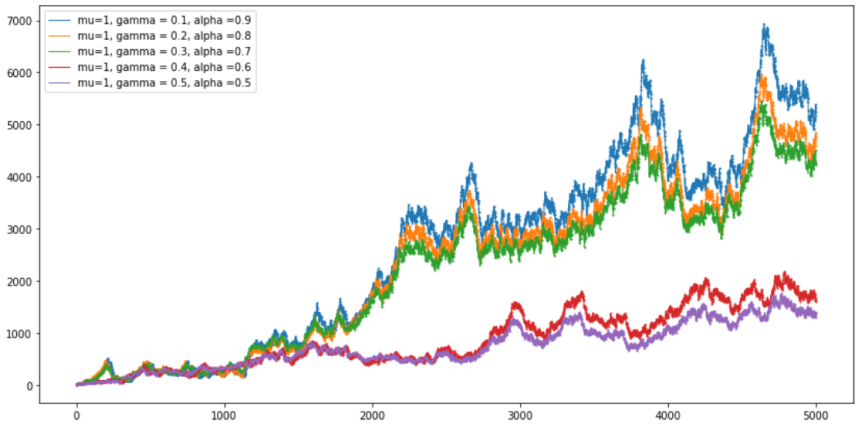
```
#########################################################################
## Plot 1: different alpha values, fixed gamma
#########################################################################

# Plotting data for different values of alpha
n = 1000

Lambda1, Delta1 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.1, alpha =0.1)
Lambda2, Delta2 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.1, alpha =0.3)
Lambda3, Delta3 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.1, alpha =0.5)
Lambda4, Delta4 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.1, alpha =0.8)
Lambda5, Delta5 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.1, alpha =0.9)

# Looking at different values of alpha, fixed gamma

fig = plt.figure(figsize=(14,7))

plt.scatter(range(0,n), Delta1, s=1)
plt.scatter(range(0,n), Delta2, s=1)
plt.scatter(range(0,n), Delta3, s=1)
plt.scatter(range(0,n), Delta4, s=1)
plt.scatter(range(0,n), Delta5, s=1)

plt.plot(range(0,n), Delta1, linewidth=1, label="mu=1, gamma = 0.1, alpha =0.1")
plt.plot(range(0,n), Delta2, linewidth=1, label="mu=1, gamma = 0.1, alpha =0.3")
plt.plot(range(0,n), Delta3, linewidth=1, label="mu=1, gamma = 0.1, alpha =0.5")
plt.plot(range(0,n), Delta4, linewidth=1, label="mu=1, gamma = 0.1, alpha =0.8")
plt.plot(range(0,n), Delta5, linewidth=1, label="mu=1, gamma = 0.1, alpha =0.9")

plt.legend();

#########################################################################
## Plot 2: exploring an inverse relationship between alpha and gamma
```

```
#########################################################################

n=5000
Lambda1, Delta1 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.1, alpha =0.9)
Lambda2, Delta2 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.2, alpha =0.8)
Lambda3, Delta3 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.3, alpha =0.7)
Lambda4, Delta4 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.4, alpha =0.6)
Lambda5, Delta5 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.5, alpha =0.5)

# Looking at different values of alpha

fig = plt.figure(figsize=(14,7))

plt.scatter(range(0,n), Delta1, s=1)
plt.scatter(range(0,n), Delta2, s=1)
plt.scatter(range(0,n), Delta3, s=1)
plt.scatter(range(0,n), Delta4, s=1)
plt.scatter(range(0,n), Delta5, s=1)

plt.plot(range(0,n), Delta1, linewidth=1, label="mu=1, gamma = 0.1, alpha =0.9")
plt.plot(range(0,n), Delta2, linewidth=1, label="mu=1, gamma = 0.2, alpha =0.8")
plt.plot(range(0,n), Delta3, linewidth=1, label="mu=1, gamma = 0.3, alpha =0.7")
plt.plot(range(0,n), Delta4, linewidth=1, label="mu=1, gamma = 0.4, alpha =0.6")
plt.plot(range(0,n), Delta5, linewidth=1, label="mu=1, gamma = 0.5, alpha =0.5")

plt.legend();

#########################################################################
## Plot 3: what if alpha and gamma do not add to 1?
#########################################################################

# Plotting data for different values of gamma
n = 1000

Lambda1, Delta1 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.1, alpha =0.91)
Lambda2, Delta2 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.2, alpha =0.81)
```

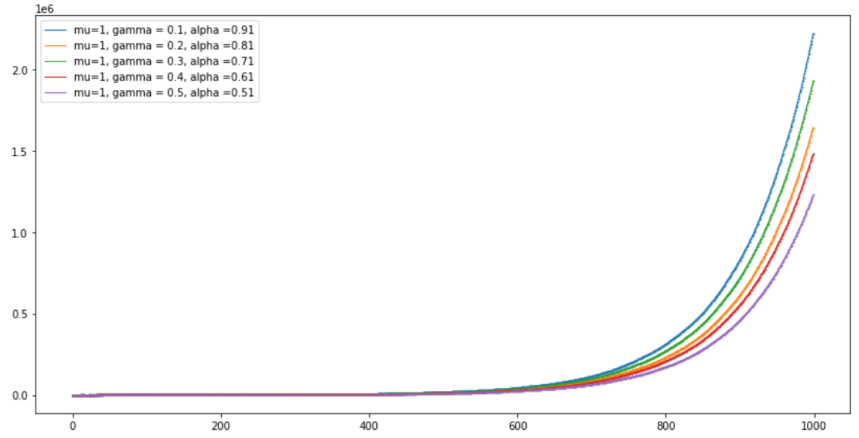
```
Lambda3, Delta3 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.3, alpha =0.71)
Lambda4, Delta4 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.4, alpha =0.61)
Lambda5, Delta5 = generate_data(n, rnd_seed=4, mu=1, gamma = 0.5, alpha =0.51)
# lambda gets too large for values beyond 0.45

fig = plt.figure(figsize=(14,7))

plt.scatter(range(0,n), Delta1, s=1)
plt.scatter(range(0,n), Delta2, s=1)
plt.scatter(range(0,n), Delta3, s=1)
plt.scatter(range(0,n), Delta4, s=1)
plt.scatter(range(0,n), Delta5, s=1)

plt.plot(range(0,n), Delta1, linewidth=1, label="mu=1, gamma = 0.1, alpha =0.91")
plt.plot(range(0,n), Delta2, linewidth=1, label="mu=1, gamma = 0.2, alpha =0.81")
plt.plot(range(0,n), Delta3, linewidth=1, label="mu=1, gamma = 0.3, alpha =0.71")
plt.plot(range(0,n), Delta4, linewidth=1, label="mu=1, gamma = 0.4, alpha =0.61")
plt.plot(range(0,n), Delta5, linewidth=1, label="mu=1, gamma = 0.5, alpha =0.51")

plt.legend();
```



(a) Fixed γ and growing α



(b) $\gamma + \alpha = 1$



(c) $\gamma + \alpha \geqslant 1$

A.12. Hawkes: Full 3D plot of the model parameters with the negative log-likelihood values

```
#############################################################################
## Negative Log-likelihood with 5,000 data points
#############################################################################

## Generating Data
n = 5000  ## samples
rnd_seed = 4

# Set the parameters
mu0 = 8
gamma0 = 0.1
alpha0 = 0.25

Lambda_, data5000 = generate_data(n, rnd_seed, mu0, gamma0, alpha0)

## ------------------

## any number
mu = np.linspace(1, 10, 25)
# infection rate larger than 0
alpha = np.linspace(0.1, 0.9, 25)
# decay rate between 0 and 1
gamma = np.linspace(0.1, 0.9, 25)

mm, aa, gg = np.meshgrid(mu, alpha, gamma)

## ------------------

## making an array for the lambdas

l = mm.copy()
```

```
## copying the meshgrid shape for lambda estimates
# 0 - array for lambda estimate
lambda_est = l - l

## ------------------

n = len(data5000)

l_list = [mm]  ## list for the element wise multiplication
of the data points with each lambda estimate

## the first multiplication of the negative
# log-likelihood

for i in range(0,n):

    lambda_est = np.multiply((l_list[i] - mm),gg) + mm + aa*data5000[i]

    l_list.append(lambda_est)

## ------------------

## finding the negative log-likelihood function

ll_list = []

for i in range(0,n):

    vals = np.multiply(data5000[i], np.log(l_list[i]))

    ll_list.append(vals)

## ------------------

## finding the negative log-likelihood function
## by summing all 10 x 10 x 10 arrays for each
```

```
## multiplication and summing all 10x10x10 arrays
## of just lambdas

neg_log_like =  -sum(ll_list) + sum(l_list)

#########################################################################
## Initial/ Full  plot
#########################################################################

## Plot not in the dissertation main

fig = plt.figure(figsize = (12,10))
ax = Axes3D(fig)

AX = ax.scatter3D(mm, aa, gg, c = neg_log_like, cmap='jet')

# plot the actual parameter point on the figure
ax.scatter(mu0, alpha0, gamma0, c='black', s= 100)

ax.set_xlabel('mu', fontweight ='bold')
ax.set_ylabel('alpha', fontweight ='bold')
ax.set_zlabel('gamma', fontweight ='bold')

plt.title("Negative Log-likelihood 3D scatter plot",
fontweight='bold', fontsize=24)

plt.colorbar(AX, ax = ax, shrink = 0.6) ;
```

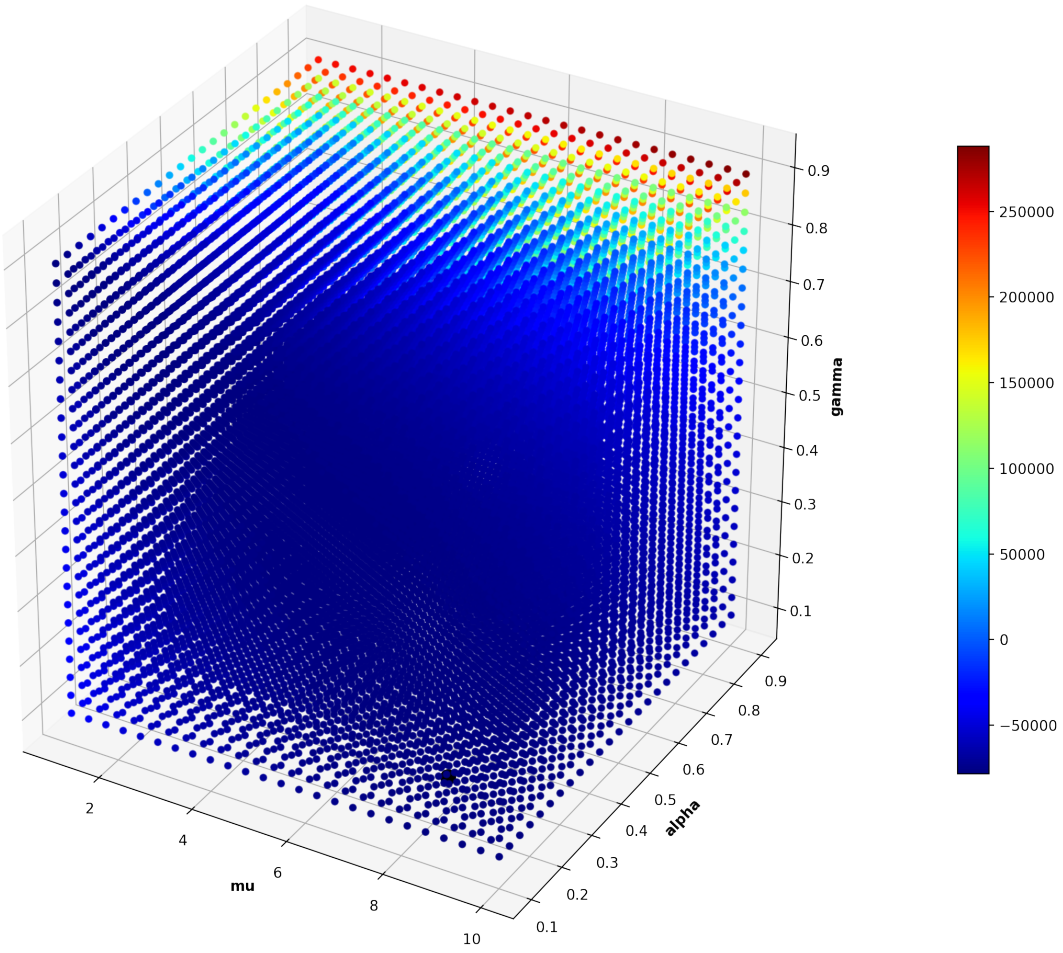


Figure A.2: Initial negative log-likelihood plot for each parameter

A.13. Hawkes: Fig. 3.3a Filtered 3D plot

```
#########################################################################
## Creating fig. 3.3a) 3D plot of the bottom 5% of the
# negative log-likelihood
#########################################################################

# create canvas
fig = plt.figure(figsize = (12,10))
ax = Axes3D(fig)

# the 3D plot
AX = ax.scatter3D(mm[neg_log_like < Lmin*0.95],
     aa[neg_log_like < Lmin*0.95],
     gg[neg_log_like < Lmin*0.95],
     c = neg_log_like[neg_log_like < Lmin*0.95],
     cmap='jet')

# plot the actual parameter point on the figure
ax.scatter(mu0,alpha0,gamma0, c='black', s= 100)

# label axis
ax.set_xlabel('mu', fontweight ='bold')
ax.set_ylabel('alpha', fontweight ='bold')
ax.set_zlabel('gamma', fontweight ='bold')

# add colour bar
plt.colorbar(AX, ax = ax, shrink = 0.6);
```

A.14. Hawkes: Fig. 3.3c The negative log-likelihood against the error terms

```
#######################################################################
## Define a function that can estimate lambdas using the given data points
#######################################################################

def find_lambda(data, mu_init, ga_init, al_init):

  n = len(data)

  lambda_list = np.ones(n+1)

  lambda_list[0] = mu_init

  for i in range(1,n+1):

      lambda_k_plus_1 = (lambda_list[i-1] - mu_init)*ga_init + mu_init +
        al_init*data[i-1]

      lambda_list[i] = lambda_k_plus_1

  lambda_list = lambda_list.reshape(n+1,1)

  return lambda_list

#######################################################################
# Using data from the 3D plot we can get the error terms
#######################################################################

# create the true parameter in the same array shape as the mashgrid arrays
t = mm.copy()
true_par = t - t + 1

# multiplying & stacking them in the same way
# mu0 = 0.5  gamma0 = 0.1  alpha0 = 0.25
True_stack = np.vstack((true_par*mu0, true_par*gamma0, true_par*alpha0))
```

```
## stack the meshgrid in the same vertical way
# mm, gg, aa
Mesh_stack = np.vstack((mm, gg, aa))

# take the Euclidean norm of the difference between the true - meshgrid values
diff = True_stack - Mesh_stack

# Placing the 3 parameter value differences into arrays of 3, for ease of use
par = []

for i in range(0,24):

  for j in range(0,24):

    for k in range(0,24):

      ## mm, gg, aa

      three_parm_arr = np.hstack((diff[i][j][k], diff[i+25][j][k],
        diff[i+50][j][k]))

      par.append(three_parm_arr)

# Make the list into an array
diff_parameters = np.array(par)

# Taking the norm of the eroor terms vs true parameters
euc_norm = np.linalg.norm(diff_parameters, axis=1)

# Similarly combining the meshgrid combinations of parameters
# Mesh_stack - the ones used to estimate the true param

par_est = []

for i in range(0,24):
```

```
  for j in range(0,24):

    for k in range(0,24):

      ## mm, gg, aa

      parm_arr_est = np.hstack((Mesh_stack[i][j][k], Mesh_stack[i+25][j][k],
        Mesh_stack[i+50][j][k]))

      par_est.append(parm_arr_est)

# Make the list into an array
est_param = np.array(par_est)

## finding the negative log-likelihood function

m = est_param.shape[0]

neg_log_likelihood = np.ones((m,1))

for i in range(m):

  lambda_list = find_lambda(data5000, mu_init = est_param[i][0],
    ga_init = est_param[i][1], al_init= est_param[i][2])

  mult_vals = np.multiply(data5000, np.log(lambda_list))
  neg_log_likelihood[i] =  -np.sum(mult_vals) + np.sum(lambda_list)

### ------------- 13-14mins to complete

# plot the error chart
fig = plt.figure( figsize = (14,10))

# mu0 = 8, gamma0 = 0.1, alpha0 = 0.25
```

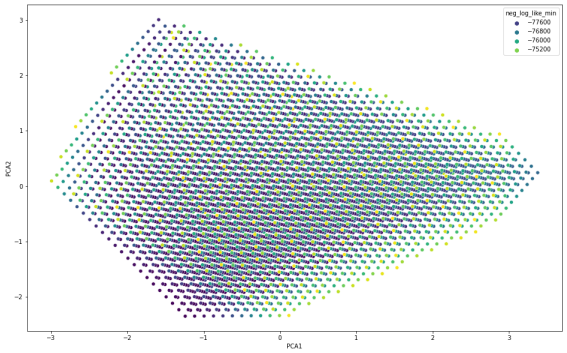
```
plt.scatter(euc_norm, neg_log_likelihood,c='dodgerblue', alpha=0.8, s=5)
plt.xlabel("Error terms" ) # \n for the true parameters $ \mu = 8$,
    $\gamma = 0.1 $ and $ \alpha = 0.25 $ ')
plt.ylabel("Negative log-likelihood values \n$(million)$")
plt.title("The negative log-likelihood against the error
    terms using the 3D meshgrid points");
```

A.15. Hawkes: PCA using Scikit-Learn

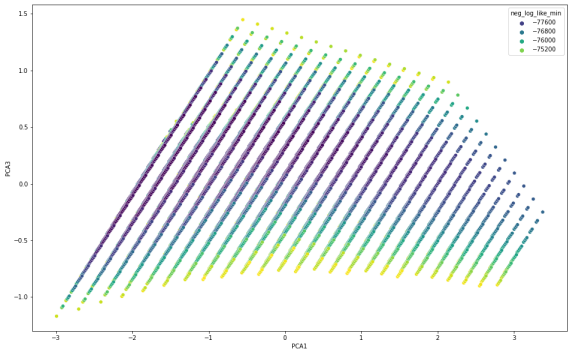
```
##############################################################################
## PCA using ScikitLearn on the bottom 5% of the Neg. log. likelihood
##############################################################################

mm_min = mm[neg_log_like < Lmin*1.05]
aa_min = aa[neg_log_like < Lmin*1.05]
gg_min = gg[neg_log_like < Lmin*1.05]

neg_log_like_min = neg_log_like[neg_log_like < Lmin*1.05]

# stack them next to each other
two = np.vstack((mm_min, aa_min))
all_param = np.vstack((two, gg_min))

# transpose the parameters
x = all_param.T

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

# standarise the data
x = StandardScaler().fit_transform(x)

# Initialize the PCA class by passing the number
# of components =3 to the constructor
# PCA() here is an object
pca = PCA(n_components=3)

# fit the data to the object
principalComponents = pca.fit_transform(x)

# Variances
print('PCA Variances are ',pca.explained_variance_)
```

```
# Out:    PCA Variances are  [1.41714506 1.28837098 0.29753584]

# Make dataframe that contains PCs
components = pd.DataFrame(pca.components_.T, columns=['PC1', 'PC2', 'PC3'],
        index=["mm_min","aa_min", "gg_min"])

# print the dataframe
components

# Axis 1 has strong positive loading for gamma, and negative loading for mu
and alpha.
# Axis 2 has strong positive loading for mu, and negative one for alpha and
gamma.
# Axis 3 has all negative loadings with the minimum one being the one for gamma.

#######################################################
# Plot PCs against each other using the
# seaborn's hue to find the optimal values
#######################################################

# make into a dataframe to use with the seaborn package
# which is built using the matplotlib package

import pandas as pd
import seaborn as sns

# make dataframes with principle components
princdf = pd.DataFrame(data = principalComponents,
        columns = ['PC1', 'PC2', 'PC3'])

# make a dataframe with the negative log-likelihood values
df_target = pd.DataFrame(data=neg_log_like_min,
        columns=["neg_log_like_min"])

# join the dataframes
finalDf = pd.concat([princdf, df_target], axis = 1)
```

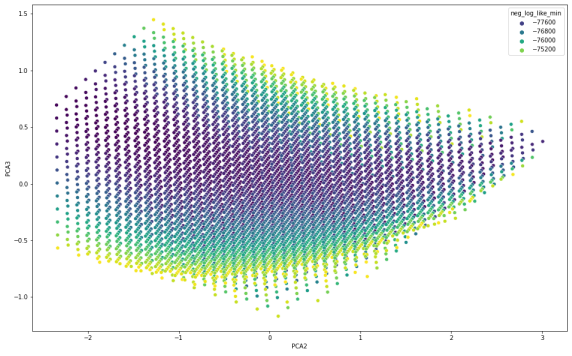
```
# plot 1
fig = plt.figure(figsize=(16,10))
sns.scatterplot(
    x="PC2", y="PC3",
    hue="neg_log_like_min",
    palette=sns.color_palette("viridis", as_cmap=True),
    data=finalDf
    # ,legend="full",
    #alpha=0.3
)

# plot 2
fig = plt.figure(figsize=(16,10))
sns.scatterplot(
    x="PC1", y="PC3",
    hue="neg_log_like_min",
    palette=sns.color_palette("viridis", as_cmap=True),
    data=finalDf
    # ,legend="full",
    #alpha=0.3
)

# plot 3
fig = plt.figure(figsize=(16,10))
sns.scatterplot(
    x="PC1", y="PC2",
    hue="neg_log_like_min",
    palette=sns.color_palette("viridis", as_cmap=True),
    data=finalDf
    # ,legend="full",
    #alpha=0.3
)
```



(a) PC1 vs. PC2



(b) PC1 vs. PC3



(c) PC2 vs. PC3

A.16. Hawkes: PCA using Numpy

```
##########################################################################
## PCA using NumPy on the bottom 5% of the Neg. log. likelihood
##########################################################################

# Using the arrays from the Scikit-learn example:

# mm_min, aa_min, gg_min, neg_log_like_min
# all_param

# find the number of valuess in the array
m = mm_min.shape[0] # 984

# Standardise the whole matrix/array
means = np.mean(all_param, axis=1).reshape(3,1)
M = means*np.ones((3,m))
std = np.std(all_param, axis=1).reshape(3,1)

x = (all_param - M)/std

# PCA using NumPy
u, s, vh = np.linalg.svd(x, full_matrices=False)
u.shape, s.shape, vh.shape

# Out:  ((3, 3), (3,), (3, 984))

# check the singular values
print(s)
Out:
    array([37.32363312, 35.5874792 , 17.10198042])

u

# Out:
#    array([[-0.22142741, -0.80906495, -0.54441143],
```

```
#            [-0.59743216,  0.55378277, -0.57999953],
#            [ 0.77074296,  0.19682111, -0.60598411]])


# print the dataframe from the Scikit-Learn example
components

# Out:
#             PC1          PC2                PC3
#mm_min  -0.221427        0.809065        -0.544411
#aa_min  -0.597432       -0.553783        -0.580000
#gg_min   0.770743       -0.196821        -0.605984

# the directions and weights are the same in both methods

## double checking the singular values by
# computing the e-values of the symmetric XX^t

C = np.dot(x, x.T)
eigen_vals, eigen_vecs = np.linalg.eig(C)

np.sqrt(eigen_vals)

# Out:
#     array([17.10198042, 35.5874792 , 37.32363312])
# They are the same as the ones above

# The variances from Scikit-Learn are different to the ones from NumPy.
# I will choose the NumPy variances to plot the Heatmap and Variance Barplot,
# since the manual eigenvalue calculation also agrees with the outcome.
```

A.17. Hawkes: Fig. 3.3b The PCA heatmap

```
## u is the data we want to plot
```

```
PC_labels = ["PC1", "PC2", "PC3"]

parm = ["mu", "alpha", "gamma" ]

fig, ax = plt.subplots(figsize=(8,6))
im = ax.imshow(u, cmap='GnBu_r', interpolation='nearest')

# We want to show all ticks...
ax.set_xticks(np.arange(len(PC_labels)))
ax.set_yticks(np.arange(len(parm)))

# ... and label them with the respective list entries
ax.set_xticklabels(PC_labels)
ax.set_yticklabels(parm);

# Loop over data dimensions and create text annotations.
for i in range(len(PC_labels)):
    for j in range(len(parm)):
        text = ax.text(j, i, np.round(u[i, j],4),
                       ha="center", va="center", color="black")


ax.set_title("Heatmap of the principle components")
fig.tight_layout();
```

A.18. Hawkes: Fig. 3.3d The PCA variance bar plot

```
# Get the variance from the singular values
variance = np.square(s)

# create plot
fig, ax = plt.subplots(figsize=(8,6))
```

```
# the labels using Latex
labs = ["$\sigma_1^2$", "$\sigma_2^2$", "$\sigma_3^2$"]

# barplot
ax.barh(labs, variance, color = "darkturquoise")

ax.invert_yaxis()
ax.set_xlabel("Variance values", fontsize= 13)
plt.title("The variance explained from the PCA");

fig.savefig("BarplotPCA.png",transparent=True);
```

A.19. Hawkes: The expectation of infinite Hawkes process, iteratively

One way to find the expectation of λ_k is iterative. We will use the law of total expectation where if we have two random variables X and Y, the expectation of X is given by

$$E\big(X\big) = E_Y\big(E\big(X|Y\big)\big). \tag{A.5}$$

In other words the expected value of X is the conditional expected value of X given Y. Using the simplified model equation in 3.9 to get

$$\begin{aligned} E\big(\lambda_{k+1}\big) &= E\big(E\big(\lambda_{k+1}|\lambda_k\big)\big) \\ &= E\big(E\big(\mu + (\lambda_k - \mu)\gamma + \alpha\Delta N_k\big)\big)\ sub-in\ 3.9 \\ &= E\big(\mu + \big(E\big(\lambda_k\big) - \mu\big)\gamma + \alpha\lambda_k\big)\big)\ expand\ first\ expectation \\ &= \mu + \gamma E\big(E(\lambda_k)\big) - \mu\gamma + \alpha E(\lambda_k)\ take\ out\ constants \\ &= \mu(1-\gamma) + (\alpha+\gamma)E(\lambda_k)\ expand\ and\ simplify. \end{aligned} \tag{A.6}$$

If we go through the same process of finding the expectation of λ_k we will get that

$$\begin{aligned} E(\lambda_k) &= \mu(1-\gamma) + (\alpha+\gamma)E(\lambda_{k-1}) \\ E(\lambda_{k-1}) &= \mu(1-\gamma) + (\alpha+\gamma)E(\lambda_{k-2}) \\ &\dots \\ E(\lambda_1) &= \mu(1-\gamma) + (\alpha+\gamma)E(\lambda_0) \\ E(\lambda_0) &= \mu\ since\ \lambda_0 = \mu. \end{aligned} \tag{A.7}$$

Putting the above all together we get

$$\begin{aligned} E\big(\lambda_{k+1}\big) &= \mu(1-\gamma) + (\alpha+\gamma)\Big(\mu(1-\gamma) + (\alpha+\gamma)E(\lambda_{k-1})\Big) \\ &= \mu(1-\gamma) + \mu(1-\gamma)(\alpha+\gamma) + (\alpha+\gamma)^2E(\lambda_{k-1}) \end{aligned} \tag{A.8}$$

If we keep substituting the $E(\lambda_k)$ values as shown above we get

$$\begin{aligned} E\big(\lambda_{k+1}\big) &= \mu(1-\gamma) + \mu(1-\gamma)(\alpha+\gamma) + (\alpha+\gamma)^2E(\lambda_{k-1}) \\ &= \mu(1-\gamma) + \mu(1-\gamma)(\alpha+\gamma) + \mu(1-\gamma)(\alpha+\gamma)^2 + (\alpha+\gamma)^3E(\lambda_{k-2}) \\ &\dots \\ &= \mu(1-\gamma) + \mu(1-\gamma)(\alpha+\gamma) + \mu(1-\gamma)(\alpha+\gamma)^2 + \dots + \mu(1-\gamma)(\alpha+\gamma)^{k-1} + (\alpha+\gamma)^kE(\lambda_1) \\ &= \mu(1-\gamma) + \mu(1-\gamma)(\alpha+\gamma) + \mu(1-\gamma)(\alpha+\gamma)^2 + \dots + \mu(1-\gamma)(\alpha+\gamma)^k + (\alpha+\gamma)^{k+1}E(\lambda_0) \end{aligned} \tag{A.9}$$

Summarising the above gives

$$E\big(\lambda_{k+1}\big) = \mu(1-\gamma) + \sum_{i=1}^{k}\mu(1-\gamma)(\alpha+\gamma) + (\alpha+\gamma)^{k+1}\mu\ since\ \lambda_0 = \mu. \tag{A.10}$$

If we take the limit of $k \to \infty$, we will get the sum of a geometric series with $r = (\alpha + \gamma)$ and $a = \mu(1 - \gamma)$ which would give

$$\lim_{k \to \infty} E\big(\lambda_{k+1}\big) = \frac{\mu(1-\gamma)}{1-(\alpha+\gamma)}, \tag{A.11}$$

since we know that $(\alpha + \gamma) < 1$ from the initial analysis in A.11. then $\lim_{k\to\infty}(\alpha+\gamma)^{k+1} \to 0$. In summary, for the values of $\mu > 0, 0 < \gamma < 1$ and $0 < \alpha < 0$ where $\alpha + \gamma < 1$, the expectation of the deterministic value λ_k at infinite time steps or data points, is A.11.

A.20. Hawkes: Fig. 3.4 Expectation ODE

```
# import packages
from scipy.integrate import odeint

# function that returns dy/dt
def model(y,t):
    mu_ode = mu0
    gamma_ode = gamma0
    alpha_ode = alpha0

    dydt = mu_ode*(1-gamma_ode)  + (gamma_ode+ alpha_ode -1) * y
    return dydt

# initial condition y0 = E{t=0} = mu
y0 = mu0 ##+ alpha0*mu0

#####################################################
# Create a range of parameters to test the ODE
#####################################################

mu = np.linspace(0.5, 3.0, num=5)
gamma = np.linspace(0.01, 0.1, num=5)
alpha = np.linspace(0.5,0.8, num=5)

y_arr = np.zeros((5,50))

# time points
t = np.linspace(0,100)

for i in range(5):

  mu0 = mu[i]
  gamma0 = gamma[i]
  alpha0 = alpha[i]
```

```
  # solve ODE
  y = odeint(model,y0,t)

  y = y.reshape(1,50)

  # add to array
  y_arr[i] = y

# Set Seaborn's figure layout as a default layout
sns.set()

# Create the canvas for plotting the figure
fig, ax = plt.subplots(figsize=(10,7))

for i in range(5):

  # plot results
  plt.plot(t,y_arr[i], label="mu0 = {}, gamma0 = {}, alpha0 = {}".format(mu[i],
    np.round(gamma[i],2),np.round(alpha[i],2) ))
  plt.xlabel('Time')
  plt.ylabel('E($\lambda(t+\delta t)$)')
  plt.legend()

# checking the ODE makes sense by manually calculating the
# value of lambda_k as t goes to infinity

# estimate - when mu = 0.5, gamma = 0.01, alpha = 0.5
est = (0.5*(1-0.01))/(1-0.01-0.5)

# the ODE calculation - the last value of the ODE
actual = y_arr[0][-1]

print("Multiplication estimate: ", np.round(est,4))
print("ODE calculation", np.round(actual,4))
```

```
# Out:
#       Multiplication estimate:   1.0102
#       ODE calculation 1.0102
```

A.21. Hawkes: Fig. 3.5 Variance ODE

```
# function that returns dV/dt
def model_var(V,t):
    mu_ode = mu0
    gamma_ode = gamma0
    alpha_ode = alpha0

    dVdt = 2*(gamma_ode+ alpha_ode - 1) * V + alpha_ode**2 * y_arr[0][-1]
    return dVdt

# initial condition y0 = E{t=0} = mu
# y0 = mu0 ##+ alpha0*mu0

mu = np.linspace(0.5, 3.0, num=5)
gamma = np.linspace(0.01, 0.1, num=5)
alpha = np.linspace(0.5,0.8, num=5)

V0 = 0

V_arr = np.zeros((5,50))

# time points
t = np.linspace(0,100)

for i in range(5):

  mu0 = mu[i]
  gamma0 = gamma[i]
  alpha0 = alpha[i]

  # solve ODE
  y = odeint(model,y0,t)
  y = y.reshape(1,50)
  # add to array
```

```
    y_arr[i] = y

    # solve ODE
    V = odeint(model_var,V0,t)
    V = V.reshape(1,50)
    # add to array
    V_arr[i] = V

# Plot the ODE Variance
fig, ax = plt.subplots(figsize=(10,7))

for i in range(5):

    # plot results
    plt.plot(t,V_arr[i], label="mu0 = {}, gamma0 = {}, alpha0 = {}".format(mu[i],
      np.round(gamma[i],2),np.round(alpha[i],2) ))
    plt.xlabel('Time')
    plt.ylabel('Var($\lambda(t+\delta t)$)')
    plt.legend()

# check variance for mu = 0.5, gamma = 0.01 alpha = 0.5
( 0.5*(1-0.01)*0.5**2 )/(2* (0.5 - 1 + 0.01)**2 )
# 0.25770512286547276

actual = V_arr[0][-1]
actual
# 0.2577051228551178
```

A.22. Hawkes: The update functions

```
# Note the data must not have zero values otherwise the update function
# do not work.

# Setting the decimal places style from the e610 style to normal decimal places
np.set_printoptions(formatter={'float_kind':'{:f}'.format})

## Generating Data
# mu = 8 gamma = 0.1 alpha = 0.25

n = 5000  ## samples
rnd_seed = 4
mu0 = 8
gamma0 = 0.01
alpha0 = 0.25

# Generate data
Lambda_, data5000 = generate_data(n, rnd_seed, mu0, gamma0, alpha0)

data5000.shape ## (1000,)

#reshape the data array
data5000 = data5000.reshape(n,1)
data5000.shape # (1000, 1)

#########################################################################
## Find lambdas from data points
#########################################################################

def find_lambda(data, mu_init, ga_init, al_init):

  n = len(data)

  lambda_list = np.ones(n+1)
```

```
    lambda_list[0] = mu_init

    for i in range(1,n+1):

        lambda_k_plus_1 = (lambda_list[i-1] - mu_init)*ga_init + mu_init +
          al_init*data[i-1]

        lambda_list[i] = lambda_k_plus_1

    lambda_list = lambda_list.reshape(n+1,1)

    return lambda_list


#########################################################################
## Mu update function
#########################################################################

def find_mu(data, mu_, lambda_list):

    n = len(data)

    mu_update = (1/n)*np.sum(np.divide(mu_*data , lambda_list[:-1, :] ))

    return mu_update

#########################################################################
## Alpha update function
#########################################################################

def find_alpha(data, alpha_, gamma_, lambda_list):

    n = len(data)

    # starts from k=2 to k = N, so we need N-1 balues
```

```
  A = np.zeros((n-1, 1))

  # create constants - need to start from second value k=2 (index is 1 )
  C = (data/lambda_list[:-1, :])

  # add initial value
  A[0] =  data[0]*C[1]

  for j in range(1, n-1):

    A[j] = (gamma_*(A[j-1]/C[j]) +  data[j])*C[j+1]

  D = np.sum(A*alpha_)

  # A = sum of the first N-2 data points * (1+gamma)/alpha
  A = (1 + gamma_)*np.sum(data[:-2, :])/alpha_

  B = data[-1,:]

  alpha_update = ( -B + np.sqrt(B**2 + 4*A*D) )/ (2*A)

  return alpha_update

#######################################################################
## Gamma update function
#######################################################################

def find_gamma(data, alpha_, gamma_, lambda_list):

  n = len(data)

  # starts from k=3 to k = N, so we need N-2 values
  K = np.zeros((n-2, 1))

  # create constants - need to start from second value k=3 (index is 2 )
  G = (data/lambda_list[:-1, :])
```

```
  # add initial two values
  K[0] =  gamma_*data[0]*G[2]
  K[1] = ((K[0]/G[2])*2*gamma_ + gamma_*data[1])*G[3]

  # for loop the rest N-2-2 values
  for l in range(2, n-2):

    K[l]  =((K[l-1]/G[l+1])*2*gamma_ - (K[l-2]/G[l])*(gamma_**2) +
        gamma_*data[l])*G[l+2]

  F = - np.sum(K*alpha_)

  # this one is both a and c in the quadratic eqn
  P = (alpha_/ (gamma_ + 1))* np.sum(data[:-2,:])

  gamma_update = (-P + np.sqrt(P**2 - 4*F*P))/(2*P)

  return gamma_update

#########################################################################
######## Find all parameters function
#########################################################################

######## FIND ALL PARAMETERS function & negative log-likelihood

def find_param(itr, mu0, ga0, al0, data):

  n = len(data)

  iter = itr

  neg_log_likelihood = np.zeros((iter,1))

  mu_arr = np.zeros(iter)
  mu_arr[0] = mu0
```

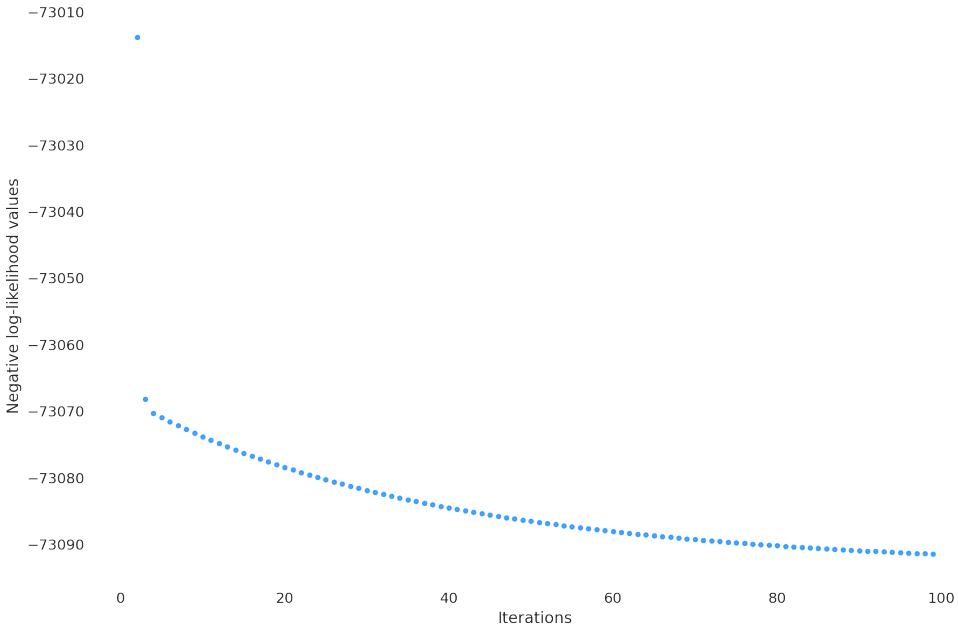
```
    alpha_arr = np.zeros(iter)
    alpha_arr[0] = al0

    gamma_arr = np.zeros(iter)
    gamma_arr[0] = ga0

    for i in range(1,iter):

      # initialising the values & updating them for the loop
      mu_val = mu_arr[i-1]
      al_val = alpha_arr[i-1]
      ga_val = gamma_arr[i-1]

      # finding the lambda estimates with the parameter values
      lambda_list = find_lambda(data, mu_val, ga_val, al_val)

      mu_arr[i] = find_mu(data, mu_val, lambda_list)

      alpha_arr[i] = find_alpha(data, al_val, ga_val, lambda_list)

      gamma_arr[i] = find_gamma(data, al_val, ga_val, lambda_list)

      mult_vals = np.multiply(data, np.log(lambda_list[:-1, :]))

      neg_log_likelihood[i] = -np.sum(mult_vals) + np.sum(lambda_list)

    return mu_arr, alpha_arr, gamma_arr, neg_log_likelihood

# Testing it
# initial guesses
mu_init = 10
ga_init = 0.001
al_init = 0.4
itr = 100
```

```
mu_new, alpha_new, gamma_new, neg_log_likelihoods = find_param(itr,
    mu_init, ga_init, al_init, data5000)

# What are the last parameters after 100 iterations
print("Estimates after 100 iterations: ", np.round(mu_new[-1],4),
    np.round(alpha_new[-1],4), np.round(gamma_new[-1],4))
print("Real values: ",mu0, alpha0, gamma0)

# Function est after 100 iterations:  7.5089 0.3164 0.0396
# Estimates after 100 iterations:  7.5089 0.3164 0.0396
# Real values:  8 0.25 0.1
```

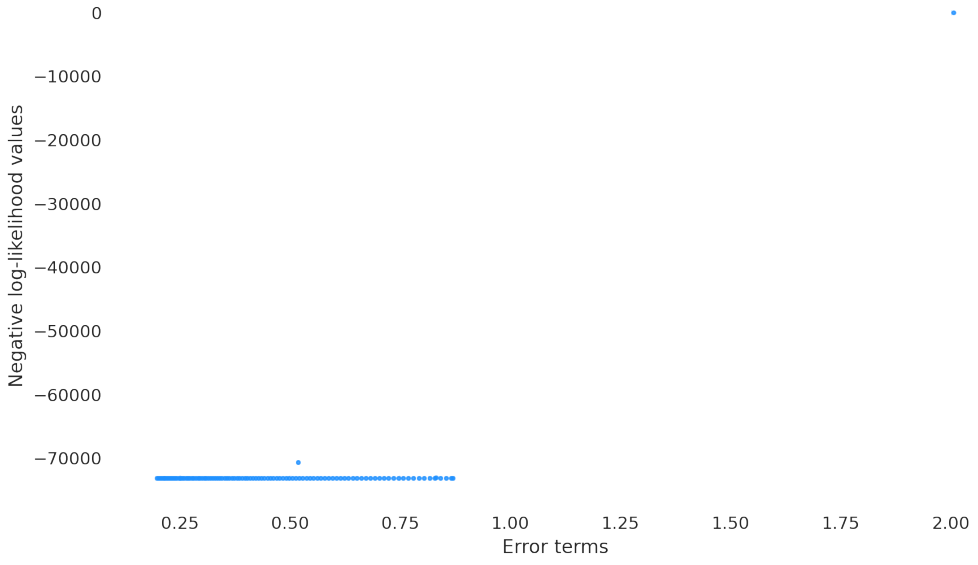
A.23. Hawkes: Fig. 3.6 and fig. 3.7 The parameters and negative log-likelihood updates

```
#########################################################################
######## Find the parameter errors at each iterations
#########################################################################

# true parameters array of the same size as the parameter iterates arrays
# mu = 8 gamma = 0.01 alpha = 0.25
x = np.array([8, 0.01, 0.25])

# Making the array containing the true parameters of the same size as
# the one above
true_param = np.ones((all_param_iter.shape[0],3))*x
true_param.shape

# Taking the difference betweeen the true and estimated parameters
diff = true_param - all_param_iter

euc_norm = np.linalg.norm(diff, axis=1)
euc_norm.shape

#########################################################################
######## Negative log-likelihood against the parameter error
######## terms at each iteration
#########################################################################

# Get rid of the initial guess and first two iterations
# because they have quite large negative log-likelihoods.

fig = plt.figure( figsize = (10,6))

plt.scatter(euc_norm[3:], neg_log_likelihoods[3:], c='dodgerblue',
    alpha=0.8, s=5)
plt.xlabel("Error terms" ) # \n for the true parameters $ \mu = 8$,
    $\gamma = 0.1 $ and $ \alpha = 0.25 $ ')
```

```
plt.ylabel("Negative log-likelihood values \n $(million)$");

#########################################################################
########Negative log-likelihood decreasing at each iteration
#########################################################################

# Exclude the negative log-likelihood for the first two parameter iterations
# because they have quite large negative log-likelihood values.

# x-axis is the number of iterations
x = np.arange(neg_log_likelihoods.shape[0])

fig = plt.figure( figsize = (12,8))

plt.scatter(x[3:], neg_log_likelihoods[3:], c='dodgerblue', alpha=0.8, s=10)
plt.xlabel("Iterations ") #\n for the true parameters $ \mu = 8$,
    $\gamma = 0.1 $ and $ \alpha = 0.25 $ ')
plt.ylabel("Negative log-likelihood values")
```



(a) Negative log-likelihood decreasing with each iteration



(b) Negative log-likelihood decreasing with the decreasing parameter error values

A.24. Hawkes: COVID data and fig. 3.8

```
## Covid-19 Data - Cases by Specimen date

# https://coronavirus.data.gov.uk/details/cases

# import data with Pandas into a DataFrame: COV_all
import pandas as pd
COV_all = pd.read_csv('data_2021-May-11_Cases_by_specimen_date.csv')

# make the dates column into datetime data type for ease of use
COV_all['Date']= pd.to_datetime(COV_all['date'])

# Filter out all the dats for covid cases after the first date back to school
Cov_sept = COV_all[COV_all['Date'] > pd.to_datetime('2020-09-06')]

# checking the values in column date are between 08-09-2020 and the
# last date in 2021
Cov_sept['Date'].sort_values(ascending=True)

# Checking the number of values can be divided by 7 and that there are
# no 0 values
cases_day = Cov_sept['newCasesBySpecimenDate'].values
np.where(cases_day == 0)  # no 0 cases!

# We can have 35 means: n=35 x 7 array
# reshape into an n by 7 array
cs_day_seven = cases_day.reshape(-1,7)
# have 35 means

# take the average number of cases per week
avr_week_cases = np.average(cs_day_seven, axis=1)
avr_week_cases

# reshape into an array I can plug into the parameter updating function
avr_week_cases = avr_week_cases.reshape(-1,1)
```

```
# Set the initial mu to be the overall average cases for the 35 week period
mu_init = np.average(avr_week_cases)
mu_init

# 16665.18775510204 = mu

# Testing it for 50 iterations
mu_updates, alpha_updates, gamma_updates, neg_log_likelihood_updates =
    find_param(itr= 50, mu0 = mu_init, ga0 = 0.01, al0 = 0.25, data =
        avr_week_cases)

# Explodes after iteration 21!

# Testing it for 21 iterations

mu_updates, alpha_updates, gamma_updates, neg_log_likelihood_updates =
    find_param(itr= 21, mu0 = mu_init, ga0 = 0.01, al0 = 0.25,
        data = avr_week_cases)

# plotting the negative log-likelihood
# Exclude the negative log-likelihood for the initial parameters.

# Setting the x-axis to be the number of iterations
x = np.arange(neg_log_likelihood_updates.shape[0])

fig = plt.figure( figsize = (12,8))

plt.scatter(x[2:], neg_log_likelihood_updates[2:], c='dodgerblue',
    alpha=0.8, s=10)
plt.xlabel("Iterations" ) # \n for the true parameters $ \mu = 8$,
    $\gamma = 0.1 $ and $ \alpha = 0.25 $ ')
plt.ylabel("Negative log-likelihood values");

x2 = range(0,avr_week_cases.shape[0])
```

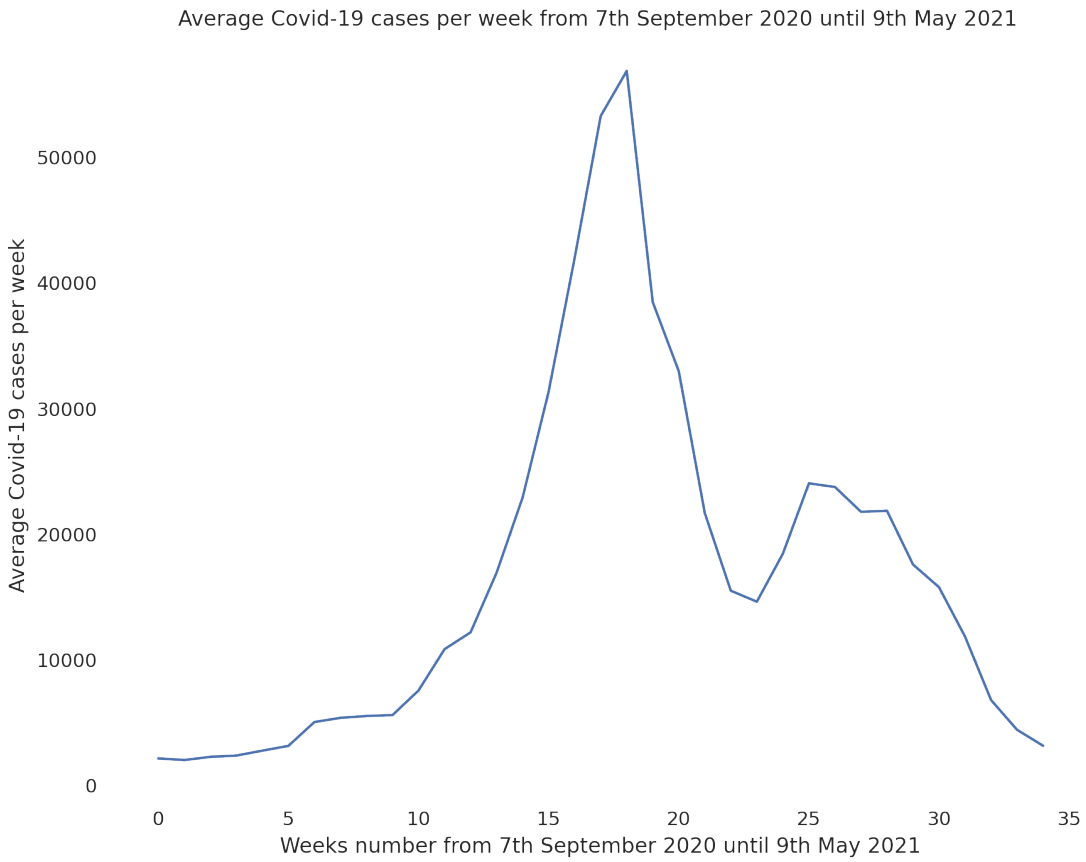


Figure A.5: Covid-19 average cases per day.

```
fig = plt.figure(figsize=(10,8))

plt.plot(x2, avr_week_cases)
plt.xlabel('Weeks number from 7th September 2020 until 9th May 2021')
plt.ylabel('Average Covid-19 cases per week')

plt.title("Average Covid-19 cases per week from 7th September 2020
    until 9th May 2021");
```

References

[1] T. J. Ypma, *Historical development of the Newton–Raphson method,* SIAM review **37** (1995) 531.

[2] C. Lemaréchal, *Cauchy and the Gradient Method,* .

[3] P. E. Gill and W. Murray, *Quasi-Newton methods for unconstrained optimization,* IMA Journal of Applied Mathematics **9** (1972) 91.

[4] K. Lange, *Numerical Analysis for Statisticians Second Edition,* Springer, 2010.

[5] K. Lange, *MM optimization algorithms,* SIAM, 2016.

[6] M. Ortega James and C. Rheinboldt Werner. *Iterative solution of nonlinear equations in several variables,* 1970.

[7] A. P. Dempster, N. M. Laird, and D. B. Rubin, *Maximum likelihood from incomplete data via the EM algorithm,* Journal of the Royal Statistical Society: Series B (Methodological) **39** (1977) 1.

[8] A. R. De Pierro, *On the convergence of an EM-type algorithm for penalized likelihood estimation in emission tomography,* IEEE transactions on medical imaging **14** (1995) 762.

[9] Z. Zhang, J. T. Kwok, and D.-Y. Yeung, *Surrogate maximization/minimization algorithms and extensions,* Machine Learning **69** (2007) 1.

[10] A. Argyriou, R. Hauser, C. A. Micchelli, and M. Pontil. *A DC-programming algorithm for kernel selection,* . In *Proceedings of the 23rd international conference on Machine learning* 412006.

[11] D. Böhning and B. G. Lindsay, *Monotonicity of quadratic-approximation algorithms,* Annals of the Institute of Statistical Mathematics **40** (1988) 641.

[12] J. De Leeuw and W. J. Heiser, *Convergence of correction matrix algorithms for multidimensional scaling,* Geometric representations of relational data **36** (1977) 735.

[13] P. J. Groenen, *The majorization approach to multidimensional scaling: some problems and extensions,* DSWO Press, 1993.

[14] D. R. Hunter and K. Lange, *Quantile regression via an MM algorithm,* Journal of Computational and Graphical Statistics **9** (2000) 60.

[15] D. R. Hunter and K. Lange, *A tutorial on MM algorithms,* The American Statistician **58** (2004) 30.

[16] P. J. Rousseeuw, *Least Median of Squares Regression,* Journal of the American Statistical Association **79** (1984) 871.

[17] D. R. Hunter and K. Lange, *Quantile Regression via an MM Algorithm,* Journal of Computational and Graphical Statistics **9** (2000) 60.

[18] R. Koenker and G. Bassett, *Regression Quantiles,* Econometrica **46** (1978) 33.

[19] G. D. Hutcheson, *Ordinary least-squares regression,* L. Moutinho and GD Hutcheson, The SAGE dictionary of quantitative management research(2011) 224.

[20] P. J. Rousseeuw and A. M. Leroy, *Robust regression and outlier detection,* John wiley & sons, 2005.

[21] E. Schlossmacher, *An iterative technique for absolute deviations curve fitting,* Journal of the American Statistical Association **68** (1973) 857.

[22] G. Merle and H. Späth, *Computational experiences with discrete l p-approximation,* Computing **12** (1974) 315.

[23] S. Y. Kung, M.-W. Mak, S.-H. Lin, M. Mak, and S. Lin, *Biometric authentication: a machine learning approach,* Prentice Hall Professional Technical Reference New York, 2005.

[24] M.-S. Yang, C.-Y. Lai, and C.-Y. Lin, *A robust EM clustering algorithm for Gaussian mixture models,* Pattern Recognition **45** (2012) 3950 .

[25] G. J. McLachlan and T. Krishnan, *The EM algorithm and extensions,* John Wiley & Sons, 2007.

[26] K. Lange, *Mathematical and statistical methods for genetic analysis,* Springer Science & Business Media, 2003.

[27] N. G. Becker, *Uses of the EM algorithm in the analysis of data on HIV/AIDS and other infectious diseases,* Statistical Methods in Medical Research **6** (1997) 24.

[28] Z. Hu. *Initializing the EM algorithm for data clustering and sub-population detection*. PhD thesisThe Ohio State University2015.

[29] M.-A. Rizoiu, Y. Lee, S. Mishra, and L. Xie, *A tutorial on hawkes processes for events in social media,* arXiv preprint arXiv:1708.06401(2017).

[30] L. Amaral and A. Papanicolaou, *Price impact of large orders using Hawkes processes,* NYU Tandon Research Paper(2019).

[31] X. Lu and F. Abergel, *High-dimensional Hawkes processes for limit order books: modelling, empirical analysis and numerical calibration,* Quantitative Finance **18** (2018) 249.

[32] L. Adamopoulos, *Cluster models for earthquakes: Regional comparisons,* Journal of the International Association for Mathematical Geology **8** (1976) 463.

[33] S. Flaxman, S. Mishra, A. Gandy, H. J. T. Unwin, H. Coupland, T. A. Mellan, H. Zhu, T. Berah, J. W. Eaton, P. N. Guzman, and others, *Estimating the number of infections and the impact of non-pharmaceutical interventions on COVID-19 in European countries: technical description update,* arXiv preprint arXiv:2004.11342(2020).