CHALMERS | UNIVERSITY OF GOTHENBURG

MASTER'S THESIS

Application of L-BFGS to a Large-Scale Poisson MAP Estimation

Torbjörn Wästerlid

Department of Mathematical Sciences
CHALMERS UNIVERSITY OF TECHNOLOGY
GOTHENBURG UNIVERSITY
Göteborg, Sweden 2012

Thesis for the Degree of Master of Science

Application of L-BFGS to a Large-Scale Poisson MAP Estimation

Torbjörn Wästerlid

Department of Mathematical Sciences
Chalmers University of Technology and Gothenburg University
SE-412 96 Göteborg, Sweden
Göteborg, September 2012

Mathematical Sciences
Göteborg 2012

Abstract

This thesis considers the solution of a large-scale optimization problem obtained when fitting a large Bayesian Poisson model to data. The model is a simplification of the one used by the online advertising optimization company Admeta to successfully predict online ad performance. It is shown that the problem can be solved using the L-BFGS algorithm, a low memory version of the BFGS algorithm developed by Broyden, Fletcher, Goldfarb and Shanno. In order to speed up the solution process further the objective function and gradient computations are parallelized on the GPU, resulting in a factor 3 speed increase for the overall solution compared to the CPU implementation. Finally, the solution quality is evaluated and compared to that obtained when using Stochastic Gradient Descent (SGD). L-BFGS is found to obtain solutions where the objective is about 0.05% lower than the solution found by the SGD and does so more efficiently. However, if a lower degree of accuracy is acceptable then the SGD is competitive.

Acknowledgements

First and foremost I'd like to thank my partner in crime Albin Lindbäck for being a good friend and a great programming help. Many long hours in front of the screen would undoubtedly have been at least twice as long had you not been around. Anders Sjögren, my supervisor, also deserves a wad of cash, had I had any. Examiner Michael Patriksson has my thanks for being so thoroughly professional and honest.

I am thankful to all the other folks at Admeta as well, for being friendly and for letting me occupy such a large swath of their limited office space and breathing their warm and carbon-dioxide rich air. Finally, thanks to mom and dad for telling me that writing a report is "not so bad"; come to think of it, I rather enjoyed it.

Contents

Glossary of Notation

Notation/Concept	Short explanation	Section
α	Step length	4.3
ϵ	A model effect	3.1
θ	Model parameters	Tab 1, Sec 3.1
$\theta^{P:M}$, $\theta^{M:P}$	Matrices for interaction effects	3.1
$\theta(X_i)$	Parameters corresponding to covariate X_i	3.1
μ_i	Expected number of clicks for observation i	3.1
$\Psi(\theta_j)$	Function returning hyperparameter of θ_j's effect	3.1
Ψ^{ξ}	Hyperparameter for parameters of effect ξ	3.1
Ψ_I	Hyperparameter for interaction effects	6
Ψ_S	Hyperparameter for simple effects	6
BFGS	A quasi-Newton update	4.7.2
C	The number of parameters in the simplified model	3.1
C^p	The family of p-times continuously differentiable functions	-
Convergence rate	A way of measuring algorithm speed	A.1
Covariate	Variables in a statistical model	3.1, 2.3
CUDA	Language for computational GPU programming	-
CUDA-L-BFGS	L-BFGS with parallel function/gradient calculation	-
GAM	Generalized Additive Model	2.3
GPU	Graphical Processing Unit	-
Hessian	-	Def 3
Hyperparameters	Parameters of a prior distribution	2.2
Impression	A showing of an ad on a website	3.1
Interaction effect	-	Def 8
Large-Real	Set of real data from Admeta	6
Large-Simulated	Simulated data set	6
L-BFGS	Limited-memory BFGS	4.7.3
Likelihood function	-	2.2
Link function g	Function linking expectation to model parameters	2.3
MAP	Maximum a-posteriori	2.2
m_T	The true model used for investigating fit	6.3
N	Number of observations	-
Observation	Outcome and covariates of an impression	3.1

Notation/Concept	**Short explanation**	**Section**
Prior distribution	Belief of parameter distribution before seeing data	2.2
Posterior density	Belief of parameter distribution after seeing data	2.2
$rmse$	Root mean squared error	6.3
SGD	Stochastic Gradient Descent	4.5
Simple effect	-	Def 7
Small-Real	Set of real data from Admeta	6
w_i	Weight of observation i	5.4
x^*	The solution to an optimization problem	4
X	Set of covariates for observations $1...N$	3.1
X_i	Covariate of observation i	3.1
Y	Vector of clicks for observation $1...N$	3.1
Y_i	Number of clicks for observation i	3.1

1 Introduction

This master thesis deals with the development and evaluation of an algorithm to fit Admeta's statistical model to their historical data.

1.1 Context

Statistical models are seeing increased use in virtually every branch of science and the humanities. Especially the availability of computing power and vast amounts of data over the Internet has facilitated the increase and led to more and more complex statistical models being used to model and make predictions about phenomena all over the world [1].

Admeta does revenue optimization for publishers of on-line ads. Their main tool for doing so is a statistical model containing a very large number of parameters, in the order of hundreds of thousands. Using this model they are able to make predictions about which ad material is likely to generate the biggest revenue when a user enters a web-page.

However, in order to make predictions, the model must first be fit to historical observations of user behaviour. If the model is simple, such as a linear model with a small number of parameters, it may be fit using simple analytical calculations, but for complex models, with a large number of parameters, more sophisticated methods are used. Model fitting can be formulated and solved as a mathematical optimization problem, in this case, a maximum a-posteriori problem. If the number of historical data points and model parameters is very large the optimization problem can be very time consuming to solve.

It is in Admeta's interests to fit their model quickly and efficiently since this will enable them to rerun the optimization often, for instance when new observations arrive, and to add further effects to the model, making it more accurate. Furthermore, a quicker optimization will also speed up evaluation of new models on large amounts of data.

1.2 Background

There are many books and articles discussing statistical modeling, for instance [2, 3]. The latter book discusses General Additive Models, which is closely related to what Admeta uses, whereas the former is more elementary. A lot of work has been invested in developing methods for fitting statistical models. Stochastic Gradient Descent (SGD) has been shown to be successful for large-scale machine learning problems and on-line learning [4]. Another traditional method is the Gauss-Newton algorithm. L-BFGS has also been

used in model fitting and can be found in for instance the statistical programming language R. It may be, however, that it has seen more widespread use in other areas [5, 6, 7]. For books in nonlinear and numerical optimization, see [8, 9, 10, 11].

1.3 Purpose

The aims and goals of this thesis are:

- to formulate the optimization problem obtained when estimating the parameters of the statistical model and investigate its properties.
- to compare and analyze the ability of several optimization algorithms to solve a simplified version of Admeta's problem and assess their suitability to parallelization.
- to select/design and implement an algorithm that solves the parameter optimization problem. The algorithm should be parallelizable and ideally at least 10 times faster than the one currently used.
- to evaluate the quality of the solution obtained.

1.4 Scope

Admeta uses a large statistical model; however, this work has been done using a simplified version of the model. Therefore, the number of parameters are fewer. Hopefully this should not reduce the applicability of the result of this thesis to Admeta's actual model.

1.5 Thesis outline

Section 2 introduces some basic mathematical concepts and includes a brief overview of statistical models. This can be useful when reading section 3, which introduces Admeta's statistical model and formulates the corresponding optimization problem. Concepts in optimization are introduced in Section 4 and section 5 deals with the method used to solve the problem. Finally, the results, followed by a discussion and conclusion are presented.

2 Mathematical background

This section introduces some mathematical concepts used throughout the thesis.

2.1 Basic concepts

Definition 1 (Positive definite)**.** A matrix $M \in \mathbb{R}^{n\times n}$ is positive definite if for any vector $v \in R^n \setminus \{0\}$,

$$v^T M v > 0. \tag{1}$$

It is positive semi-definite if $v^T M v \geq 0 \ \forall v \in R^n$. Furthermore, the inverse of a positive definite matrix is also positive definite.

Positive definiteness is an important concept in linear algebra and is useful in optimization in order to ensure convergence of certain algorithms.

Definition 2 (Convexity)**.** A function $f : \mathbb{R}^n \to \mathbb{R}$ is said to be convex if for any two points $x_1, x_2 \in \mathbb{R}^n$ and any $t \in$ [0,1],

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2).$$

Intuitively a function f is convex if a line between any two points $(x_1, f(x_1))$, $(x_2, f(x_2))$ lies above the graph of $f(x)$ on the interval (x_1, x_2). Convexity has important consequences for optimization as a local minimum is then equal to a global minimum.

Definition 3 (Hessian)**.** Let $f : \mathbb{R}^n \to \mathbb{R}$ be twice differentiable. Then the Hessian, $H(x) \in \mathbb{R}^{n\times n}$, is a symmetric matrix defined as

$$H(x) = \nabla^2 f(x),$$

so that element (i,j) is given by

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j}.$$

The Hessian contains information regarding the shape of a function and is of importance in various optimization algorithms. For instance, the Newton method utilizes the Hessian directly. Other methods, such as the Quasi-Newton methods, use only Hessian approximations.

Definition 4 (Optimization problem)**.** Let $f : \mathbb{R}^n \to \mathbb{R}$. The problem of finding an x^* such that $f(x^*) \leq f(x)$ for all $x \in \mathbb{R}^n$ is formulated as,

$$\min f(x) \tag{2}$$
$$\text{s.t. } x \in \mathbb{R}^n. \tag{3}$$

Optimization problems are found in all areas of science and are classified into different types depending on the properties of f and the constraints on x.

Theorem 5 (Sherman–Morrison formula)**.** *Let $A \in \mathbb{R}^{n\times n}$ be invertible and u,$v \in \mathbb{R}^n$ be arbitrary. If the rank one update $(A+uv^T)$ of A is also invertible it is given by*

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}.$$

The Sherman–Morrison formula is a useful analytical tool for calculating the inverse of a slightly modified matrix for which the inverse is known.

2.2 Bayesian statistics

Admeta's simplified statistical model, looked at later in the thesis, is a Bayesian model. Bayesian statistics differ from the more standard, frequentists approach, in its interpretation of statistics. Whereas a frequentist would view a parameter in a model or distribution as a constant, a Bayesian practitioner would assign a *prior* belief of the parameter before seeing any data and a *posterior* belief after seeing the data. These beliefs are represented using probability densities, usually called the prior and posterior density, respectively. The prior density of a parameter often has parameters of its own; these parameters are called *hyperparameters*.

The key instrument in Bayesian statistics is Bayes' formula, which states how a prior belief should change given new evidence:

Theorem 6 (Bayes' formula)**.** *Let X, Y be two random variables, and $f_{X,Y}$ be their joint distribution. Then,*

$$f_X(x \mid Y = y) = \frac{f_Y(y \mid X = x)f_X(x)}{f_Y(y)}.$$

Here, $f_X(x \mid Y = y)$ is the *posterior density* of X, $f_X(x)$ is the *prior density* of X and $f_Y(y \mid X = x)$ is the *likelihood function*, giving the probability of the outcome y given the parameters $X = x$. Bayes' formula will be used later on in order to fit Admeta's simplified model to data.

2.3 Statistical models

This section is not intended to give the reader a deep understanding of the subject of statistical modeling or the subset of General Additive Models. Rather, it is intended as a very brief overview, so that the reader can get a slightly better understanding of how Admeta's model works and how the optimization problem is linked to it.

A statistical model is often an attempt to relate a response variable Y to some predictive variables, henceforth called covariates, X. Usually it is assumed that there is some unknown description of the relationship between X and Y, and that Y deviates from this description by a random error ϵ. A simple form of statistical model is the linear model,

$$Y = X\beta + \epsilon, \quad \mathbb{E}[\epsilon] = 0, \tag{4}$$

where X is the covariate matrix and β is a vector of parameters of the model. To denote row i of the covariate matrix X, X_i will be used. Note that here X and the expectation $\mu = \mathbb{E}[Y]$ have a simple linear relation,

$$\mu = \mathbb{E}[Y] = \mathbb{E}[X\beta + \epsilon] = X\beta.$$

Thus, $\mu_i = X_{i1}\beta_1 + X_{i2}\beta_2 + \ldots + X_{in}\beta_n$.

A more advanced and powerful class of models are the generalized additive models, henceforth shortened to GAM. In such models there is no longer a linear relationship between μ_i and the covariates. The expectation μ_i is instead linked to the covariates via a so called *link function*. Commonly used link functions are the logarithm, $\log(\mu_i) = X_i\beta$, the inverse, $\frac{1}{\mu_i} = X_i\beta$ and the logit, $\log(\frac{\mu_i}{1-\mu_i}) = X_i\beta$. Often the choice of link function depends on the distributional assumption on Y. In a GAM, Y is assumed to belong to some distribution from the exponential family. If for instance $Y \sim Poisson(\mu)$, then the logarithm is commonly used as link, since this bears with it nice properties. The general structure of a GAM, as found in [3], is:

$$g(\mu_i) = X_i\theta + f_1(x_1^i) + f_2(x_2^i) + f_3(x_3^i, x_4^i) + \ldots, \tag{5}$$

where g is the link function, $x_j^i \in \mathbb{R}$ is the jth element in covariate row X_i, f_j are smooth functions of the covariates and θ are parameters for the strictly parametric parts of the model.

Compared to a linear model, GAM allows the capture of much more complex relations between predictors and response variables. However, the flexibility of the model comes at a greater risk of overfitting. Overfitting the model to known data might lead to poor predictions for new data. An approach to counter overfitting is to use a Bayesian model (see Section 2.2). In that case the prior distributions assigned to the parameters will act as regulators, hopefully preventing overfitting.

3 Problem specification and analysis

This section discusses a simplification of the model used by Admeta and formulates the optimization problem to be solved. The problem is then analyzed. For a short introduction to general additive models see Section 2.3. During the entire Section 3, f will be used to denote various probability densities. Note that the arguments to f clarify which probability density is being referred to. Thus, for instance, $f(\theta)$ and $f(\Psi)$ refer to two different probability densities.

3.1 The simplified model

Admeta's statistical model optimizes the revenue for on-line publishers. It provides a link from the covariates (such as material, website, time of day), via the model parameters, to an expected number of clicks for a material. The simplified model is a GAM, or, more precisely, a Bayesian additive Poisson model with log link.

Most readers will have been online and seen an online advertisement. Often, they are similar to an ordinary newspaper advertisement, consisting of a picture and some text, however, they may also play sound, animations or video. An advertisement is given a unique id by Admeta, the so called material id, which is used in the statistical model. The material id is thus simply a name given by Admeta to some advertisement.

Visitors of websites will also have noticed that there are usually more than one ad on a website and that these appear at different positions on the site. These positions are known to Admeta as placements, and each placement, on every website connected to Admeta, is given a unique name called the placement-id.

The placement-id and material-id are just two of the various variables that Admeta can take into account in order to model the performance of an ad. Other factors might be the number of times the ad has been shown to a visitor previously, called the frequency id, or which advertiser has designed the ad, the so called advertiser id. If a site requests you to log in, other variables, such as your sex or interests, can be used in the model.

Let an ad be shown on some website; such an event will be called an **impression**. By Y_i, denote the outcome, that is, the number of clicks the ad received. The outcome of an impression, together with the covariates of that impression, will be called an **observation**. In the simplified model there are 5 covariates: *advertiser id (a), frequency id (r), material id (m), order id (o) and placement id (p)*, where the letter in parenthesis is the short form. The capital letter of the short form, for example A, will denote the set of

all such covariates (in this example, A is the set of advertisers). The set containing all five covariates of the ith observation will be denoted by $X_i := \{a_i, r_i, m_i, o_i, p_i\}$. Such an X_i will be referred to as a *complete covariate*, and a member of complete covariate i is subscripted by i. For example, a_i refers to the advertiser in complete covariate X_i. An actual complete covariate might look something like $X_{12341231} = \{745, 12, 5444, 32, 52982\}$, since the real world attributes, that is, covariates, used in the model are each given an integer id.

The outcome, Y_i, is assumed to follow the Poisson distribution:

$$Y_i \sim f(Y_i \mid \theta, X_i) = \frac{\mu_i^{Y_i}}{Y_i!} e^{-\mu_i}. \tag{6}$$

Here, and in what follows, $\mu_i \equiv \mathbb{E}[Y_i]$ and θ denotes the set of all model parameters. The parameters are the unknowns of the model and must be estimated, using previously collected observations in order to make predictions of the future. In the simplified model there are seven effects, where each effect is an attempt to model, or take into account, some part of advertisement performance.

An effect is a function mapping one or more covariates to a real number, using the model parameters. They correspond to the smooth functions f found in Section 2.3, discussing generalized additive models. There are two main type of effects in the simplified model: the simple effect and the interaction effect.

Definition 7 (Simple effect)**.** Let Z be some set of covariates and $z \in Z$. Furthermore, let $\theta^\xi \in \mathbb{R}^{|Z|}$ denote the set of parameters corresponding to the specific effect ξ. A simple effect is a function, $\xi : Z \to \theta^\xi$. Thus, a simple effect takes elements of a covariate set, such as the set of materials, and maps each element to a distinct parameter.

Advertiser- Frequency- Material- Order- and Placement effects are all **simple** effects. The Intercept effect can also be considered a simple effect by looking at it as a constant function that returns the same parameter regardless of the covariate it operates on. It represents the average performance of an advertisement.

Definition 8 (Interaction effect)**.** An interaction effect is a function taking two covariates and returning a number representing whether these two covariates tend to work well together (a positive number) or counteract each other (a negative number). Letting Z, W be any two covariate sets, $z \in Z$, $w \in W$ and K be a positive integer, then an interaction effect is defined as:

$$\xi : (z,w) \to \mathbb{R}$$

In the simplified model, the interaction effects consist of two matrices of parameters. Each matrix is of size $K\times \mid Z \mid$ and $K\times \mid W \mid$, respectively, so there is one column in the corresponding matrix for each element in the corresponding covariate set. To obtain a real number describing the interaction between covariates z, w, the dot product is taken between a column corresponding to z from the first matrix and a column corresponding to w from the second.

The reason behind the arcane definition of the interaction effects, rather than having just one, huge, matrix with a parameter for each of the possible pairs z, w, is that such a matrix would be too large. Using two matrices is a way of factoring the enormous matrix into something manageable. Although the effect of factoring the full matrix in this way is somewhat unclear, it is hoped that the factorization will capture the behaviour of the full matrix in roughly the same way that the K eigenvectors associated with the K biggest eigenvalues would.

The material-placement effect is an interaction effect between materials and placements and consists of the two matrices, $\boldsymbol{\theta}^{M:P} \in \mathbb{R}^{K\times|M|}$ and $\boldsymbol{\theta}^{P:M} \in \mathbb{R}^{K\times|P|}$. For the simplified model studied here, $K = 10$. The column of $\boldsymbol{\theta}^{P:M}$ corresponding to placement $p \in P$ is denoted $\boldsymbol{\theta}^{P:M}_p$.

The names of effects and corresponding parameters are shown in Table 1. For future reference, the set of all parameters in the model will be denoted θ. The set of all parameters of a certain effect, ξ, is denoted θ^ξ.

Table 1: Model effect names and the associated parameters. K is a constant determining the size of the interaction effects.

Effect	**Parameters**
Intercept effect	$\theta^I \in \mathbb{R}$
Advertiser effect	$\theta^A \in \mathbb{R}^{\lvert A\rvert}$
Frequency effect	$\theta^R \in \mathbb{R}^{\lvert R\rvert}$
Material effect	$\theta^M \in \mathbb{R}^{\lvert M\rvert}$
Order effect	$\theta^O \in \mathbb{R}^{\lvert O\rvert}$
Placement effect	$\theta^P \in \mathbb{R}^{\lvert P\rvert}$
Material-placement effect	$\boldsymbol{\theta}^{M:P} \in \mathbb{R}^{K\times\lvert M\rvert}, \quad \boldsymbol{\theta}^{P:M} \in \mathbb{R}^{K\times\lvert P\rvert}$

The set of parameters corresponding to a complete covariate X_i is defined as the set of parameters that are used when applying the model effects to the covariates in X_i. The set is denoted by

$$\theta(X_i) = \{\theta^I, \theta^A_{a_i}, \theta^F_{f_i}, \theta^M_{m_i}, \theta^O_{o_i}, \theta^P_{p_i}, \boldsymbol{\theta}^{M:P}_{m_i}, \boldsymbol{\theta}^{P:M}_{p_i}\}. \tag{7}$$

In the simplified model, the link between effects, parameters and the expectation is the logarithm:

$$\begin{aligned}\log \mu_i =& \xi^I(a_i) + \xi^A(a_i) + \xi^R(r_i) + \xi^M(m_i) + \xi^O(o_i) + \xi^P(p_i) + \xi^{MP}(m_i, p_i) = \\ & \theta^I + \theta^A_{a_i} + \theta^R_{r_i} + \theta^M_{m_i} + \theta^O_{o_i} + \theta^P_{p_i} + (\boldsymbol{\theta}^{M:P}_{m_i})^T \cdot \boldsymbol{\theta}^{P:M}_{p_i}. \end{aligned} \tag{8}$$

Here, $\xi^x()$, is one of the effects. The logarithm is chosen as link partly because it is the canonical link function for the Poisson distribution, which means that it simplifies calculations involving the model. A canonical link function also bears with it other nice properties which can be read about in [3]. As can be seen the model is such that logarithm of the expected value consists of the sum of 6 simple effects and one interaction effect.

Since Admeta uses a Bayesian approach to the model all parameters are given prior distributions. The prior assumption is that parameters within each effect are normally distributed, with a different variance for each effect. This assumption is made mainly out of convenience, the Normal distribution has a simple analytical form and very light tails, meaning that extreme values are unlikely. Thus letting ξ denote any effect,

$$\theta^\xi_j \sim \mathcal{N}(0, \Psi^\xi) = \frac{1}{\sqrt{2\pi\Psi^\xi}} \exp\left(-\frac{(\theta^\xi_j)^2}{2\Psi^\xi}\right), \tag{9}$$

where Ψ^ξ is the variance of parameters corresponding to effect ξ. The variance Ψ^ξ is the so called hyperparameter of the effect. As mentioned in Section 2.3, the hyperparameter can be adjusted, depending on how much the parameter values are believed to vary. Strong evidence (a lot of observations) is needed in order for parameters to take values that are unlikely to the prior distribution.

The ultimate goal of the model is to predict the outcome of future impressions. To achieve this, the model is fit to existing observations. In Bayesian terms, fitting a model amounts to finding the posterior distribution of the parameters, $f(\theta \mid Y, X)$, given observations $\{Y_i, X_i\}_{i=1}^N$. Here $Y = \{Y_i\}_{i=1}^N$ and $X = \{X_i\}_{i=1}^N$. Using Bayes formula (Thm 6),

$$f(\theta \mid Y, X) = \frac{f(Y \mid \theta, X) f(\theta \mid X)}{f(Y \mid X)} \propto f(Y \mid \theta, X) f(\theta), \tag{10}$$

where $f(Y \mid X)$ is the probability of the observed outcome and is constant with regard to θ. It is also assumed that

$$f(\theta \mid X) = f(\theta).$$

This assumption follows from the fact that

$$f(\theta \mid X) = \frac{f(X \mid \theta)}{f(X)} f(\theta) \approx f(\theta),$$

where the inequality is due to $f(X)$, the distribution of complete covariates, having a weak dependence on the parameters θ. Therefore, $\frac{f(X|\theta)}{f(X)} \approx 1$. By assuming equality it is essentially assumed that the complete covariates are independent of the parameters.

The first part of the right–hand side of eq. (10), $f(Y \mid \theta, X)$, is called the likelihood function and is equivalent to the distribution of outcomes. The distribution of outcomes Y given θ and X is, as mentioned previously, assumed to be Poisson distributed. The second part, $f(\theta)$, is the prior distribution of the parameters. Both of the above distributions are known, so eq. (10) is of practical value.

In eq. (10) one can clearly tell the structure of the posterior distribution. The prior distribution, $f(\theta)$, is small for values that are a-priori unlikely, meaning that the likelihood function, $f(Y \mid \theta, X)$, must be large for us to believe in that value of θ after seeing data.

The posterior distribution can be utilized in many ways. One of its many applications is for obtaining point estimates of the model parameters, θ. One of the most common estimates is the maximum a-posteriori estimate (MAP). It is simply the θ that maximizes the posterior distribution. That θ consists of the most likely parameter values, and should therefore be suitable for future predictions. To find the MAP estimate the following optimization problem must be solved:

$$\max_{\theta} f(\theta \mid Y, X) \propto f(Y \mid \theta, X) f(\theta). \tag{11}$$

3.2 The objective function

In this section the objective function that is the topic of this thesis is derived. Firstly, it is assumed that the outcomes of individual impressions are conditionally independent. This means that knowing the outcome of some other impression will not impact the probability of the outcome for the current impression. Due to the independence, the distribution for outcomes factors into,

$$f(Y \mid \theta, X) = f(Y_1, Y_2, \ldots, Y_N \mid \theta, X_1, \ldots, X_N) = \prod_{i=1}^{N} f(Y_i \mid \theta, X_i).$$

The prior on each parameter is an independent normal distribution. Therefore, due to independence, we obtain,

$$f(\theta) = f(\theta_1, \ldots, \theta_C) = f(\theta_1) \ldots f(\theta_C) = \prod_{s=1}^{C} f(\theta_s).$$

Here, $C = |\theta|$. Furthermore, the logarithm is strictly monotonically increasing, so problem (11) can be reformulated as,

$$\max_{\theta} \prod_{i=1}^{N} f(Y_i \mid \theta, X_i) f(\theta) = \max_{\theta} \sum_{i=1}^{N} \log f(Y_i \mid \theta, X_i) + \sum_{s=1}^{C} \log(f(\theta_s)).$$

Due to the distributional assumption, eq. (6), on $f(Y_i \mid \theta, X_i)$,

$$\log\left(f(Y_i \mid \theta, X_i)\right) = \log\left(\frac{\mu_i^{Y_i}}{Y_i!} e^{-\mu_i}\right) = Y_i \log(\mu_i) - \mu_i - \log(Y_i!). \tag{12}$$

The prior distribution on θ, eq. (9), means that the following expression for $\log(f(\theta_s))$ is obtained:

$$\log\left(f(\theta_s)\right) = \log\left(\frac{1}{\sqrt{2\pi\Psi(\theta_s)}} \exp(-\frac{\theta_s^2}{2\Psi(\theta_s)})\right) = -\frac{\log(2\pi)}{2} - \frac{\log(\Psi(\theta_s))}{2} - \frac{\theta_s^2}{2\Psi(\theta_s)}. \tag{13}$$

The optimization problem is thus to

$$\begin{aligned} \underset{\theta}{\text{minimize}} & -\sum_{i=1}^{N} w_i \left(Y_i \log(\mu_i) - \mu_i - \log(Y_i!)\right) \\ & + \sum_{s=1}^{C} \left(\frac{\log(2\pi)}{2} + \frac{\log(\Psi(\theta_s))}{2} + \frac{\theta_s^2}{2\Psi(\theta_s)}\right) \\ \text{subject to } & \theta \in \mathbb{R}^C, \end{aligned} \tag{14}$$

where N is the number of observations, $w_i > 0$ is a weight introduced due to a weighted sampling of the observations, $\log(\mu_i)$ is defined by eq. (8), $\Psi(\theta_s)$ is a function selecting the right Ψ depending on the effect θ_s belongs to. Finally, the right hand sides of eq. (12) and eq. (13) have been multiplied by -1 in order to turn maximization into minimization and $C = |\theta|$. The objective function thus consists of two additive parts: a sum over all the observations, and a second sum over all the parameters.

3.3 Gradient calculation

The gradient of the objective function is needed in many optimization algorithms. To calculate the gradient of the objective function, eq. (14), let ξ be any effect and θ_j^ξ be the jth parameter corresponding to that effect. As before, μ_i is the expectation of observation i. The gradient is the vector of all partial derivatives of the first order. One element is thus given by,

$$\frac{\partial \sum_{i=1}^{N} \log f(Y_i \mid \theta, X_i) + \sum_{s=1}^{C} \log(f(\theta_s))}{\partial \theta_j^\xi}. \tag{15}$$

Due to the linearity of the differential operator the above expression can be split into two parts, one concerning the observations and the second concerning the prior distributions. Before moving on to the first of the two parts, note that,

$$\frac{\partial \log(\mu_i)}{\partial \theta_j^\xi} = \frac{\partial \log(\mu_i)}{\partial \mu_i} \frac{\partial \mu_i}{\partial \theta_j^\xi} = \frac{1}{\mu_i} \frac{\partial \mu_i}{\partial \theta_j^\xi},$$

so that

$$\frac{\partial \mu_i}{\partial \theta_j^\xi} = \mu_i \frac{\partial \log(\mu_i)}{\partial \theta_j^\xi}.$$

Thus, for the first part,

$$\begin{aligned} \frac{\partial \log(f(Y_i \mid \theta, X_i))}{\partial \theta_j^\xi} &= \frac{\partial Y_i \log(\mu_i)}{\partial \theta_j^\xi} - \frac{\partial \mu_i}{\partial \theta_j^\xi} - \frac{\partial \log(Y_i!)}{\partial \theta_j^\xi} \\ &= \frac{\partial \log(\mu_i)}{\partial \theta_j^\xi} (Y_i - \mu_i). \end{aligned} \tag{16}$$

From eq. (8),

$$\frac{\partial \log(\mu_i)}{\partial \theta_j^\xi} = \begin{cases} 1, & \text{if } \xi \text{ is a simple effect and } \theta_j^\xi \in \theta(X_i), \\ \theta_{mk}^{M:P}, & \text{if } \theta_j^\xi = \theta_{pk}^{P:M} \text{ and } \boldsymbol{\theta}_p^{P:M} \in \theta(X_i), \\ \theta_{pk}^{P:M}, & \text{if } \theta_j^\xi = \theta_{mk}^{M:P} \text{ and } \boldsymbol{\theta}_m^{M:P} \in \theta(X_i), \\ 0, & \text{otherwise,} \end{cases} \tag{17}$$

where $\theta_{pk}^{P:M}$ is the kth element in the $K \times 1$ vector $\boldsymbol{\theta}_p^{P:M}$ and $\theta(X_i)$ is defined in eq. (7).

For the second part, the partial gradient contribution is:

$$\frac{\partial \sum_{s=1}^{C} \log(f(\theta_j^\xi))}{\partial \theta_j^\xi} = -\frac{\partial \left(\frac{\log(2\pi)}{2} + \frac{\log(\Psi(\theta_j^\xi))}{2} + \frac{(\theta_j^\xi)^2}{2\Psi(\theta_j^\xi)} \right)}{\partial \theta_j^\xi} = -\frac{\theta_j^\xi}{\Psi(\theta_j^\xi)}. \tag{18}$$

Using equations (16), (17) and (18) each partial derivative, as defined in eq. (15) can be calculated. Thus the full gradient of the objective function can also be calculated using these expressions.

3.4 Problem properties

Several things are notable from an optimization point of view, when studying the optimization problem (14). To begin with, the problem is smooth, that is, it has derivatives of all orders. Furthermore, it is a nonlinear unconstrained optimization problem. The prior contribution (13) to the objective function is strictly positive and acts as a constraint that limits the parameters from taking very extreme negative values. Moreover, as the following theorem states, a model without interaction effects is convex.

Theorem 9. *For the Admeta simplified model, Section 3.1, without the interaction effects, the resulting optimization problem is convex.*

Proof. Consider eq. (14) without the interaction effects. Applying eq. (8) and moving the minus sign inside the summation yields the following form for the objective function,

$$\begin{aligned}\sum_{i=1}^{N}\big(&- Y_i(\theta^I + \theta^A_{a_i} + \theta^F_{f_i} + \theta^M_{m_i} + \theta^O_{o_i} + \theta^P_{p_i}) + \\ &+ \exp\left(\theta^I + \theta^A_{a_i} + \theta^F_{f_i} + \theta^M_{m_i} + \theta^O_{o_i} + \theta^P_{p_i}\right) + \log(Y_i!)\big) w_i + \\ &+ \sum_{s=1}^{C}\left(\frac{\log(2\pi)}{2} + \frac{\log(\Psi(\theta_s))}{2} + \frac{\theta_s^2}{2\Psi(\theta_s)}\right).\end{aligned} \tag{19}$$

Now $Y_i \geq 0$ and $w_i > 0$ for $i = 1 \ldots N$. The parameters occur only as linear terms, in the exponential and the square, but the linear, exponential and square functions are all convex and a sum of convex functions is also convex. Hence the above expression is convex. □

Unfortunately, the introduction of interaction parameters adds a term $\boldsymbol{\theta}^{M:P}_{m_i} \cdot \boldsymbol{\theta}^{P:M}_{p_i}$ to all sums involving the parameters in eq. (19), making the problem non-convex. This means that the Theorems 19, 20 and 22, for convergence and convergence rate, derived in the forthcoming section, cannot be expected to hold for our problem.

One final observation about the problem at hand is that for interaction effect parameters, the origin acts as a saddle point. By inspection of equation (17) and (18) it is evident that the gradient is 0 at the origin for those parameters. Analytical calculation of the Hessian is very complicated and

therefore only the three contributions that are relevant (not equal to zero at the origin) are stated below:

$$\frac{\partial \log(f(Y_i \mid \theta, X_i))}{\partial \theta_j^\xi \partial \theta_l^\zeta} = \begin{cases} -w_i\mu_i(1 + \theta_{mk}^{M:P}\theta_{pk}^{P:M}) & \text{if } \theta_j^\xi = \theta_{pk_1}^{P:M} \in \theta(X_i), \\ & \theta_l^\zeta = \theta_{mk_2}^{M:P} \in \theta(X_i) \\ & \text{and } k_1 = k_2 \\ -w_i\mu_i(1 + \theta_{pk}^{P:M}\theta_{mk}^{M:P}) & \text{if } \theta_j^\xi = \theta_{mk_1}^{M:P} \in \theta(X_i), \\ & \theta_l^\zeta = \theta_{pk_2}^{P:M} \in \theta(X_i) \\ & \text{and } k_1 = k_2 \end{cases} \tag{20}$$

$$\frac{\partial \log\left(f(\theta)\right)}{\partial \theta_j^\xi \partial \theta_l^\zeta} = -\frac{1}{\Psi^\xi} \text{ if } \theta_j^\xi = \theta_l^\zeta, \tag{21}$$

where $\theta_{mk}^{M:P}$ is the element at position k in the vector $\boldsymbol{\theta}_m^{M:P}$.

Studying equations (20) and (21) for θ at the origin one can tell that for the interaction effect parameters the Hessian will be a diagonal matrix with a band, K indexes from the diagonal on either side. Here K is the size of the interaction effects. A matrix, such as would be obtained using only one observation, and $K = 3$, is,

$$\begin{pmatrix} -\frac{1}{\Psi^\xi} & 0 & 0 & -w_i\mu_i & 0 & 0 \\ 0 & -\frac{1}{\Psi^\xi} & 0 & 0 & -w_i\mu_i & 0 \\ 0 & 0 & -\frac{1}{\Psi^\xi} & 0 & 0 & -w_i\mu_i \\ -w_i\mu_i & 0 & 0 & -\frac{1}{\Psi^\xi} & 0 & 0 \\ 0 & -w_i\mu_i & 0 & 0 & -\frac{1}{\Psi^\xi} & 0 \\ 0 & 0 & -w_i\mu_i & 0 & 0 & -\frac{1}{\Psi^\xi} \end{pmatrix}. \tag{22}$$

Such a matrix is generally indefinite, even for this case, where all the non-zero elements are negative. Hence, the origin is a saddle point. This has the implication that the origin should be avoided when selecting a starting point for an optimization algorithm.

4 Nonlinear unconstrained optimization

This section will discuss fundamental concepts in optimization used in the thesis. Throughout the entire section $f : R^n \to R$ will denote a smooth function and $g \equiv \nabla f$. Unconstrained optimization deals with problems of the type

$$\begin{aligned} &\text{minimize } f(x) \\ &\text{s.t. } x \in \mathbb{R}^n, \end{aligned} \tag{23}$$

A solution to (23) is usually denoted x^*, with $f(x^*)$ being the corresponding minimum value. An unconstrained optimization problem is called nonlinear when f is nonlinear.

This section will present several algorithms, so for the entire section a superscript ^{k} on a variable is taken to mean the kth iterate of that variable. A negative superscript $^{-k}$, on a matrix, will be the inverse of the kth iterate of that matrix. Unless stated explicitly all norms are taken to be the standard 2-norm.

4.1 Optimality conditions

Optimality conditions are used to construct solution algorithms for optimization problems, and suitable termination criteria for these. They specify the conditions that must hold in order for a point to be minimal, and thus be a solution to problem (23). The theorems stated in this section can also be found in any book on continuous optimization, such as [8, 9].

Theorem 10 (Necessary conditions)**.** *Let x^* be a local minimum of $f : R^n \to R$, where $f \in C^1$. Then,*

$$\nabla f(x^*) = \mathbf{0}^n.$$

Furthermore, if $f \in C^2$ then,

$$\nabla^2 f(x^*) \text{ is positive semi definite.}$$

Theorem 11 (Sufficient conditions)**.** *Let $f : R^n \to R$ and $f \in C^2$. A point satisfying*

$$\nabla f(x^*) = \mathbf{0}^n \text{ and } \nabla^2 f(x^*) \text{ is positive definite,}$$

is a strict local minimum of f.

4.2 Descent methods

Consider again problem (23). The most common approach to solving such a problem is to find a direction, $d^k \in \mathbb{R}^n$, such that

$$f(x^k + \alpha^k d^k) < f(x^k), \tag{24}$$

for all $\alpha^k > 0$ sufficiently small. Such a direction d^k is called a descent direction and the basic idea of descent methods is that by continuously decreasing the function value one eventually ends up in a minimum. To find an α^k, such that a large decrease in function value is obtained, a line search is performed along the direction of d^k. Rather than trying to find α^k such that $f(x^k + \alpha^k d^k)$ is minimized an α^k yielding a sufficient decrease is usually accepted, since minimizing $f(x^k + \alpha^k d^k)$ is too expensive. The whole process of finding descent directions is then repeated until the optimality conditions are fulfilled. The general algorithm outline is found in Algorithm 1.

Algorithm 1 A brief outline of a descent method.

1: Choose some starting point $x^0 \in \mathbb{R}^n$.
2: **loop**
3: Check optimality conditions, break loop if fulfilled.
4: Calculate a descent direction d^k.
5: Calculate step length $\alpha^k > 0$ through line search.
6: $x^{k+1} = x^k + \alpha^k d^k$.
7: $k \leftarrow k + 1$.
8: **end loop**

4.3 Line searches

Line searches are used to stabilize optimization algorithms, as it is often the case that using step length $\alpha^k = 1$, results in divergence or slow convergence. The formal problem is to

$$\min_{\alpha \geq 0} h(\alpha) = f(x^k + \alpha d^k), \tag{25}$$

where α is the step length, d^k the search direction and x^k the current position. Solving (25) exactly is usually avoided in practice, and is not required for convergence. Instead α is chosen so as to satisfy certain conditions.

Definition 12 (Armijo criterion)**.** Let $m_1 > 0$. The Armijo criterion accepts a step length α if

$$h(\alpha) \leq h(0) + m_1 \alpha h'(0).$$

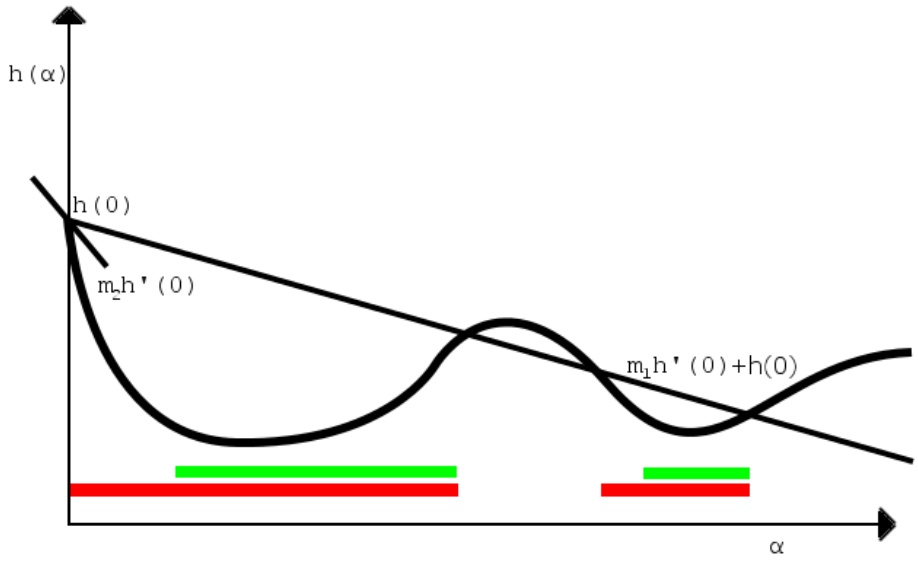


Figure 1: The figure is an illustration of the Armijo criterion (Def. 12) and the Wolfe criterion (Def. 13). The red, bottom bar, and green bar above it signify accepted values of the step length α for the Armijo and Wolfe rule, respectively.

The Armijo condition is easy to implement and is suitable for use when the gradient is expensive to evaluate. See Figure 1 for a graphical illustration of the Armijo criterion.

Definition 13 (Wolfe criterion)**.** Let $m_1, m_2 > 0$. The Wolfe criterion classifies a step length α as too short, too long, or acceptable according to:

$$\text{if } h(\alpha) \leq h(0) + m_1 \alpha h'(0) \text{ and } h'(\alpha) \geq h'(0) m_2 \text{ then } \alpha \text{ is acceptable;}$$
$$\text{if } h(\alpha) \leq h(0) + m_1 \alpha h'(0) \text{ and } h'(\alpha) < h'(0) m_2 \text{ then } \alpha \text{ is too small;}$$
$$\text{if } h(\alpha) > h(0) + m_1 \alpha h'(0) \text{ then } \alpha \text{ is too large.}$$

Note that the last condition is equivalent to the Armijo criterion.

The Wolfe criterion is theoretically sounder than that of Armijo. For instance, there is no constraint preventing a step length accepted by the Armijo criterion from being very small. If the gradient is very expensive to calculate, one might consider not using the Wolfe criterion, for all other purposes it is standard. Figure 1 illustrates the Armijo and Wolfe conditions graphically.

4.4 Gradient/Steepest descent

Gradient/Steepest descent is a descent method where $d^k = -g^k$, where $g^k \equiv \nabla f(x^k)$. It is a descent method by the definition of g^k. Gradient descent is known to suffer from slow convergence both in theory and practice [8, 10, 11].

4.5 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is used for minimization problems where the objective function can be written as a sum of independent functions, $f = \sum_{j=1}^{N} f_j$. The sum may for instance be over a set of data points where f_j is then the function evaluated for data point j. Evaluating the whole sum during every iteration of the optimization algorithm can be costly, so instead, typically, the f_j are shuffled and for each iteration $d^k = -g_l^k$, where $g_l^k = \nabla f_l(x^k)$ and $l \in \{1 \dots N\}$. Usually, a fixed number of passes over all the f_j are done and in expectation d^k will behave like the full gradient. Bottou [4] has achieved good practical and theoretical results using SGD for large-scale machine learning problems. An outline of the algorithm employed later on in the thesis is presented in Algorithm 2.

Algorithm 2 A general outline of the SGD algorithm.

```
B ← number of passes over data.
Set starting step length α^0.
N ← number of data points.
k = 0
for i = 1 → B do
    Update α^i according to some rule.
    Shuffle the f_j:s.
    for j = 1 → N do
        d^k = −g_j^k
        x^{k+1} = x^k + α^i d^k
        k = k + 1
    end for
end for
```

4.6 The Newton method

The key idea of the Newton method is to use second order information about the function to be minimized, in form of the Hessian. The step direction d^k is calculated as

$$G^k d^k = -g^k, \tag{26}$$

where $G^k \equiv \nabla^2 f(x^k)$. The Newton algorithm is known to converge very quickly for well posed problems: if f is twice continuously differentiable and the Hessian is positive definite then the speed is Q-quadratic, see appendix A.1 [10]. For a quadratic problem, for which the second order approximation is exact, Newton's method converges in one step. Nevertheless, there are some major flaws inherent in the algorithm:

1. Computing the Hessian can be very difficult and solving equation (26) takes $O(n^3)$ operations.

2. The descent property is guaranteed only if the Hessian is positive definite (see Theorem 17).

4.7 Quasi-Newton methods

Quasi-Newton methods is a class of descent algorithms for unconstrained optimization. They are inspired by the Newton method, but eliminate a lot of its drawbacks. The general Quasi-Newton method calculates d^k by

$$B^k d^k = -g^k. \tag{27}$$

In this case, $B^k \in \mathbb{R}^{n\times n}$ is some approximation of the Hessian matrix, G^k. In what follows the Quasi-Newton equation, which B^k adheres to, is introduced and motivated. Later on the most common way of constructing B^k, the BFGS update, will be discussed. The BFGS update, along with most other updates, is such that B^{k+1} can be constructed from B^k. Even more convenient is the possibility of constructing $B^{-(k+1)}$ directly, as then, the step of solving for d^k, in eq. (27), is reduced to a matrix-vector multiplication. Throughout this section and its subsections, $y^k \equiv g^{k+1} - g^k$ and $s^k \equiv x^{k+1} - x^k$.

4.7.1 The Quasi-Newton equation

A second order approximation of $f(x)$ at x^k is

$$f(x) \approx f(x^k) + (x - x^k)^T \nabla f(x^k) + \frac{1}{2}(x - x^k)^T \nabla^2 f(x^k)(x - x^k). \tag{28}$$

Differentiating (28) with regards to x gives

$$\nabla f(x) \approx \nabla f(x^k) + \nabla^2 f(x^k)(x - x^k).$$

Letting $x = x^{k+1}$ yields

$$\nabla f(x^{k+1}) - \nabla f(x^k) \approx \nabla^2 f(x^k)(x^{k+1} - x^k) \Leftrightarrow y^k \approx G^k s^k, \tag{29}$$

where $y^k \equiv \nabla f(x^{k+1}) - \nabla f(x^k)$, $s^k \equiv x^{k+1} - x^k$ and $G^k \equiv \nabla^2 f(x^k)$. By exchanging G^k with B^k and the approximate equality with strict equality we obtain the Quasi-Newton and dual Quasi-Newton equations.

Definition 14 (Quasi-Newton equation)**.** The Quasi-Newton equation is

$$B^{-k} y^k = s^k. \tag{30}$$

Definition 15 (Dual Quasi-Newton equation)**.** The dual Quasi-Newton equation is

$$y^k = B^k s^k. \tag{31}$$

Clearly the Quasi-Newton and dual Quasi-Newton equations are equivalent; separate definitions are done purely for notational convenience. All Quasi-Newton methods require the Hessian approximation B^k to satisfy eq. (31). There are two main motivations for this. Firstly, note the resemblance of eq. (31) to the mean value theorem. For a function, $h(x)$, of a single variable, the mean value of h'' between x^k and x^{k+1}, denoted G_m, satisfies $y^k = G_M s^k$. The Quasi-Newton equation thus forces B^k to have the same effect as G_M on subspaces of dimension 1. Secondly, the Quasi-Newton equation ensures that the approximation of f obtained by replacing $\nabla^2 f(x^k)$ in eq. (28) by B^k is exact at at least one point and the resulting approximation of the gradient is correct at two points. Theorem 16 states this formally.

Theorem 16. *Any Hessian approximation B^k satisfying the Quasi-Newton equation* (31) *defines an approximation,*

$$m^k(x) = f(x^k) + (x - x^k)^T g^k + \frac{1}{2}(x - x^k)^T B^k (x - x^k),$$

to f satisfying $m^k(x^k) = f(x^k)$, $\nabla m^k(x^k) = g^k$ and $\nabla m^k(x^{k+1}) = g^{k+1}$, where $g^{k+1} \equiv \nabla f(x^{k+1})$.

Proof. The first two claims follow trivially. To prove the third statement consider

$$\nabla m^k(x^{k+1}) = g^k + B^k(x^{k+1} - x^k) = g^k + B^k s^k.$$

Using eq. (31) the above becomes

$$\nabla m^k(x^{k+1}) = g^k + y^k = g^k + g^{k+1} - g^k = g^{k+1}.$$

□

Theorem 17. *Assume that $g^k \neq 0$, then a positive definite B^k guarantees that d^k, as obtained from eq.* (27)*, is a descent direction.*

Proof. First note that the inverse of a positive definite matrix is also positive definite. Then,

$$d^k = -B^{-k}g^k \Rightarrow g^k d^k = -g^k B^{-k} g^k < 0,$$

by the definition of a positive definite matrix (see Def. 1). □

A property useful for error checking Quasi-Newton implementations is the curvature condition. By multiplying the Quasi-Newton equation (31) by s^k one obtains,

$$(s^k)^T y^k = (s^k)^T B^k s^k.$$

If ever $(s^k)^T y^k < 0$, then there is something wrong with the implementation, since B^k ought to be positive definite.

4.7.2 BFGS update

The BFGS update, eq. (32), is a rank 2 update developed by Broyden [12], Fletcher [13], Goldfarb [14] and Shanno [15] independently. It has a number of good properties that can be read about in the aforementioned articles or [10, 11]. One of the most important is that the successive updates of B^{-k} stay positive definite, so that the method always generates descent directions. The BFGS update is given by

$$B^{-(k+1)} = \left(I - \frac{s^k (y^k)^T}{(s^k)^T y^k}\right) B^{-k} \left(I - \frac{s^k (y^k)^T}{(s^k)^T y^k}\right) + \frac{s^k (s^k)^T}{(s^k)^T y^k}. \tag{32}$$

Theorem 18. *The BFGS update eq.* (32) *is a rank two update satisfying the Quasi-Newton equation* (30)*.*

Proof. A rank two update of B^k can be expressed as

$$B^{k+1} = B^k + auu^T + bvv^T, \tag{33}$$

where u,$v \in \mathbb{R}^n$ and a and b are scalars to be chosen so that B^k satisfies the Quasi-Newton equation (30). Applying (33) to eq. (31) yields,

$$B^{k+1}s^k = B^k s^k + auu^T s^k + bvv^T s^k = y^k.$$

To satisfy the above equality choose

$$u = y^k, \quad v = B^k s^k \Rightarrow B^k s^k + a y^k (y^k)^T s^k + b B^k s^k (B^k s^k)^T s^k = y^k$$

and let

$$a = \frac{1}{(y^k)^T s^k}, \quad b = \frac{-1}{(s^k)^T B^k s^k}.$$

It is simple to verify that these choices lead to the desired equality. To obtain $B^{-(k+1)}$ apply the Sherman–Morrison formula (Thm. 5) twice to

$$B^{k+1} = B^k + \frac{y^k (y^k)^T}{(y^k)^T s^k} - \frac{B^k s^k (B^k s^k)^T}{(s^k)^T B^k s^k}.$$

Long calculations finally lead to

$$B^{-(k+1)} = \left(I - \frac{s^k (y^k)^T}{(s^k)^T y^k}\right) B^{-k} \left(I - \frac{s^k (y^k)^T}{(s^k)^T y^k}\right) + \frac{s^k (s^k)^T}{(s^k)^T y^k}.$$

□

The BFGS update is considered the best Quasi-Newton update [10, 11] and there are numerous results for local and global convergence as well as rate of local convergence. For proofs of the next two theorems we refer to [10]. The third theorem has been adapted from a proof for a similar update which can also be found in [10].

Theorem 19 (Local convergence rate)**.** *Assume that the BFGS algorithm with Wolfe's line search converges to x^* in a neighbourhood of f which is strictly convex and has a Lipschitzian Hessian. Then the convergence rate is Q-super-linear, see Def. 24.*

Theorem 20 (Global convergence)**.** *Assume that f is twice continuously differentiable and that f is uniformly convex, i.e, $\exists\, m, M > 0$ such that for all $x \in L(x) = \{x \mid f(x) < f(x_0)\}$, which is convex, it applies that*

$$m\|u\|^2 \leq u^T \nabla^2 f(x) u \leq M\|u\|^2, \forall u \in \mathbb{R}^n.$$

Then using a Wolfe line search (Def. 13) the sequence $\{x^k\}$ generated by the the BFGS algorithm converges to the minimizer x^* of f.

Theorem 21. *If $(s^k)^T y^k > 0$ and B^{-k} is positive definite then the BFGS update $B^{-(k+1)}$ is also positive definite.*

Proof. Let $v \in \mathbb{R}^n$, $v \neq \emptyset$ and assume B^{-k} is positive definite. Then,

$$\begin{aligned} v^T B^{-(k+1)} v &= \left(v^T - v^T \frac{s^k (y^k)^T}{(s^k)^T y^k}\right) B^{-k} \left(v - \frac{s^k (y^k)^T}{(s^k)^T y^k} v\right) + v^T \frac{s^k (s^k)^T}{(s^k)^T y^k} v = \\ &= \left(v^T - v^T \frac{s^k (y^k)^T}{(s^k)^T y^k}\right) B^{-k} \left(v - \frac{s^k (y^k)^T}{(s^k)^T y^k} v\right) + \frac{(s^k v^T)^2}{(s^k)^T y^k}. \end{aligned}$$

Now the second term is strictly greater than zero as long as $s^k v^T \neq 0$. Since B^{-k} is positive definite the first term is also strictly greater than zero, as long as,

$$v^T - v^T \frac{s^k (y^k)^T}{(s^k)^T y^k} = v^T - \frac{v^T s^k}{(s^k)^T y^k} (y^k)^T \neq 0. \tag{34}$$

In particular, condition (34) holds when, $v^T s^k = 0$. Hence $B^{-(k+1)}$ is positive definite. □

In connection to Theorem 21 it is of interest to discuss when one can expect $(s^k)^T y^k > 0$ to hold. Generally $(s^k)^T y^k = s^k g^{k+1} - s^k g^k$ and $s^k g^k < 0$, since $s^k g^k = \alpha^k d^k g^k < 0$ due to d^k being a descent direction. If using an exact line search then $s^k g^{k+1} = 0$, since α was chosen as to minimize f along d^k. An exact line search is not a requirement for $(s^k)^T y^k > 0$ to hold however. The Wolfe criterion also ensures that $(s^k)^T y^k > 0$ [11]. Using the Armijo criterion however, there are no guarantees.

4.7.3 L-BFGS

Even though the Quasi-Newton method removes the need to solve the system of equations $G^k d^k = -g^k$ it still requires the storage of B^{-k}, which is an $n \times n$ matrix. For large n this can be quite intractable. The limited memory BFGS update, L-BFGS, solves this problem by storing only a fixed number, m, of the latest y^k:s and s^k:s, where y^k, s^k are defined as previously. Using these, an approximation of the BFGS-update is constructed and only $O(mn)$ space and operations are needed. L-BFGS was first introduced in [16] and is discussed further in [10, 11, 17, 18]. It is one of the most successful algorithms for large scale optimization [11].

The BFGS update formula is restated here for ease of reference:

$$B^{-(k+1)} = \left(I - \frac{s^k (y^k)^T}{(s^k)^T y^k} \right) B^{-k} \left(I - \frac{s^k (y^k)^T}{(s^k)^T y^k} \right) + \frac{s^k (s^k)^T}{(s^k)^T y^k}.$$

To simplify slightly set $\rho^k \equiv \frac{1}{(s^k)^T y^k}$ and $V^k \equiv \left(I - \rho^k s^k (y^k)^T \right)$. The BFGS

formula can now be applied recursively,

$$
\begin{aligned}
B^{-(k+1)} &= (V^k)^T B_k^{-k} V^k + \rho_k s^k (s^k)^T \\
&= (V^k)^T \left((V^{k-1})^T B_k^{-(k-1)} V^{k-1} + \rho^{k-1} s^{k-1} (s^{k-1})^T \right) V^k + \rho^k s^k (s^k)^T \\
&= (V^k)^T (V^{k-1})^T B^{-(k-1)} V^{k-1} V^k + (V^k)^T \rho^{k-1} s^{k-1} (s^{k-1})^T V^k + \rho^k s^k (s^k)^T \\
&= ... \qquad (35) \\
&= (V^k)^T \cdots (V^{k-m+1})^T B_0^{-(k-m+1)} (V^{k-m+1} \cdots V^k) \\
&+ \rho^{k-m+1} (V^k)^T \cdots (V^{k-m+2})^T s^{k-m+1} (s^{k-m+1})^T (V^{k-m+2} \cdots V^k) \\
&+ \rho^{k-m+2} (V^k)^T \cdots (V^{k-m+3})^T s^{k-m+2} (s^{k-m+2})^T (V^{k-m+3} \cdots V^k) \\
&+ \cdots \\
&+ \rho^k s^k (s^k)^T.
\end{aligned}
$$

By performing only m levels of recursion and replacing $B^{-(k-m+1)}$ by some initial positive definite B_0^{-k} the L-BFGS update is obtained. Here, B_0^{-k}, is the positive definite inverse Hessian approximation used in the kth iteration to calculate $B^{-(k+1)}$. Note that by selecting m very large the L-BFGS is equivalent to the original BFGS-update. To calculate $d^k = -B^{-k} g^k$ in practice there is an efficient two loop formula, see Algorithm 3.

Algorithm 3 The two-loop formula for calculating $d^k = -B^{-k} g^k$.

1: $d := g^k$
2: **for** $i := k \to k - m + 1$ **do**
3: $\quad a_i := \rho^i (s^i)^T d$
4: $\quad d := d - a_i y^i$
5: **end for**
6: $d := B_0^{-k} d$
7: **for** $i := k - m + 1 \to k$ **do**
8: $\quad b := \rho^i (y^i)^T d$
9: $\quad d := d + s^i (a_i - b)$
10: **end for**
11: **return** $-d$

To verify Algorithm 3, note first, from observing eq. (35), that

$$V^i g^k = (I - \rho^i y^i (s^i)^T) g^k = g^k - \rho^i y^i (s^i)^T g^k.$$

Also note that

$$a_{k-j} = \rho^{k-j} (s^{k-j})^T V^{k-j} V^{k-j+1} \cdots V^k g^k,$$

for each $j > 1$, so that q at line 6 in Algorithm 3 is equal to

$$B_0^{-k}(V^{k-m+1} \cdots V^k)g^k.$$

At line 9 in Algorithm 3 another of the terms in (35) is added for each iteration. The full details are left for the reader to work out. The complete L-BFGS algorithm outline is stated in Algorithm 4.

Algorithm 4 The L-BFGS algorithm in pseudo-code.

$k = 0$
Set a starting point $x^0 \in \mathbb{R}^n$.
Set an initial positive definite inverse Hessian $B_0^{-k} \in \mathbb{R}^{n \times n}$.
Set m, the number of s^k:s, y^k:s to store.
repeat
  Compute g^k.
  Compute $d^k = -B^{-k}g^k$ using Algorithm 3.
  Do a line search to obtain step length α^k.
  $x^{k+1} = x^k + \alpha d^k$.
  $k \leftarrow k + 1$
until convergence criteria fulfilled.

Table 2: The number of operations per iteration and the amount of memory required for the Newton method and its derivatives.

Algorithm	Operations	Memory
Newton	$O(n^3)$	$O(n^3)$
BFGS	$O(n^2)$	$O(n^2)$
L-BFGS	$O(mn)$	$O(mn)$

Naturally, since L-BFGS is an approximation of the BFGS update, its theoretical convergence properties are not as good. However, Table 2, giving the number of operations per iteration, is clearly in favour of L-BFGS, and the important property of maintaining positive definiteness still applies, as long as B_0^{-k} is positive definite.

Theorem 22. *Let f be twice continuously differentiable and uniformly convex (as defined in Theorem 20). Assume that the sequence $\{x^k\}$ generated by L-BFGS converges to the unique minimizer x^* of f. Then the rate of convergence is at least R-linear, see Def 26.*

Proof. The interested reader is referred to [10]. □

5 Method

This section presents how the problem of fitting the simplified model to data is solved using L-BFGS and SGD.

5.1 Applying L-BFGS to the problem

L-BFGS was implemented using C++ according to Algorithm 4. The statistical model, providing objective function and gradient evaluation, was also implemented in C++. Read [19] for further details on how that was done. It was decided to use a backtracking line-search using the Armijo criteria (Def 12) for the L-BFGS routine, since evaluating the gradient was the single most time-consuming operation of the algorithm. The initial step length α^0 was set to 1 and reduced by 1/2 each time the step was rejected. Even when other types of line searches are used $\alpha^0 = 1$ is often tried first, the reason being that the original Newton method converges in one step for quadratic functions using that step length.

As a convergence criteria

$$\Delta\theta = \|\theta^{k+1} - \theta^k\| < \epsilon, \tag{36}$$

was used, where ϵ is some small positive number. During evaluations performed for this thesis, $\epsilon = 0.0001$ was used. The preference for eq. (36), over some bound on the value of $\|g^k\|$, was due to the former criteria fluctuating slightly less, although it was observed that $\|g^k\| \to 0$.

L-BFGS, Algorithm 4, also requires the specification of an initial inverse Hessian approximation, B_0^{-1}. This is an important choice since it affects the algorithm convergence. Due to a lack of a simple way of approximating the true inverse Hessian, a scaled identity matrix suggested by [10] was used:

$$B_0^{-k} = \frac{(s^k)^T y^k}{\|y^k\|^2} \cdot I, \tag{37}$$

where I is the identity matrix. A schematic motivation for eq. (37) is that

$$\frac{1}{\frac{(s^k)^T y^k}{\|y^k\|^2}} = \frac{(y^k)^T y^k}{(s^k)^T y^k},$$

where the last equality can be interpreted as a very rough, scalar, approximation of the Hessian. If y^k changes a lot and s^k but a little then the Hessian in some sense is large, and if the opposite event occurs the Hessian might be considered small.

After extensive testing, m, the number of function and gradient differences (s^k:s, y^k:s) kept in the L-BFGS, was set to 30.

5.2 Implementing SGD

As a mode of reference Admeta's implementation of SGD was ported to C++. It is, however, not immediately clear how to apply SGD to problem (14). The optimization problem (14) consists of two sums, where the first sum over all the observations, eq. (12), fits naturally into the SGD framework. Each observation can be treated as defining an independent function for which the gradient can be calculated. This gradient is thus a vector with $C =\mid \theta \mid$ elements, of which only the few positions corresponding to the parameters in $\theta(X_i)$, are non–zero. The sum over the parameters however, eq. (13), does not fit the framework. The problem here is that there is no inherent way of splitting eq. (13), or its gradient eq. (18), across all the observations.

To solve that difficulty eq. (13) is distributed across the iterations by using the weights w_i. The idea behind this is that a fraction of (13), corresponding to the relative importance of the current observation, is applied at each iteration. After looping through all the observations once, (13) will have been applied exactly once for each parameter, but smeared across the observations.

To achieve this, the total weight of every parameter is calculated before commencing the algorithm. Let W_j^ξ, be the total weight of some parameter θ_j^ξ. By total weight is meant the sum of the weights of all observations that satisfy $\theta_j^\xi \in \theta(X_i)$. During a step of the SGD, where $\theta_j^\xi \in \theta(X_i)$, the current value of the prior gradient contribution, eq. (18), is calculated and the fraction w_i/W_j^ξ of it, where w_i is the weight of the current observation, is applied to θ_j^ξ.

The actual implementation of SGD follows the pseudo-code stated in algorithm 2 closely; however, there are a few unspecified parameters in that code. Those are the number of passes to make over the data, as well as a starting value and updating scheme for the step length. The initial step length α^0 was set depending on the problem at hand and a new step length, equal to a third of the former step length, was calculated whenever another quarter of the total number of passes over the data had completed.

5.3 Parallelization of objective function and gradient

Parallel computing in mathematical optimization has attracted a lot of attention since the early nineties [20, 21, 22, 23]. Lately a lot of interest has been given to the possibility of utilizing the massively parallel processors fitted in the graphic processing unit (GPU) [24]. The GPU is also what will be used for parallelization in this thesis.

When discussing parallelism it is unavoidable to first touch upon the subject of what processor architecture is used to achieve parallelism, since this limits which algorithms that are applicable. The two most common architectures are SIMD and MIMD, acronyms for Single Instruction Multiple Data and Multiple Instruction Multiple Data [25]. Modern GPUs use SIMD type processors, which means that they are very capable of receiving a large amount of independent data points and performing the same computation on each data point. Doing many different calculations simultaneously, however, is inefficient.

The three main ways of parallelizing an optimization algorithm are:

1. Parallelize the function/gradient evaluation,
2. parallelize the algorithm in itself and
3. parallelize the linear algebra (if there is any).

When it comes to parallelizing the algorithm itself, a few different approaches are discussed in [20, 23]. A problem is that a lot of classical optimization methods, such as L-BFGS, are very sequential in nature. Usually, the next iterate depends heavily on the previous iteration, so that parallelization requires the development of a completely new algorithm. Strategies, such as probing the neighbourhood of an iterate at multiple locations in parallel to approximate the Hessian, or performing parallel line-searches, have been tried, see for instance [26]. The usefulness of these methods, however, depend a lot on the specific problem at hand and many are also better suited to MIMD type parallelism. Parallelism has found greater success in, for example, genetic algorithms and various direct searches, usually applied when the function landscape is very complex, yet contains fewer variables than in this project [27, 28].

In order to speed up the run time of L-BFGS in this thesis the function and gradient evaluation were parallelized. For the objective function, eq. (14), this is quite readily done since each partial sum is independent of the others. Therefore, the most basic idea of partitioning the sum over all observations across the available processors and then adding the resulting partial sums together works well. The sum over parameters can be treated in a similar way. Although conceptually quite simple, actually implementing the above ideas on the GPU, using the CUDA programming language, is demanding. For a full treatment on how it was done see [19].

When parallelizing the gradient evaluation the approach of splitting the sum across all available processors, as in the function evaluation, is natural. This naive idea turns out to work badly, as for the gradient, different observations may affect the same parameter. This leads to write conflicts, which

cost a lot of time. Measures were therefore needed to reduce such conflicts. Once again, confer [19] for full details.

5.4 Simulating observations

In order to make the results reproducible to others, it was decided to construct data through simulations. The simulation goals are:

- That the result is somewhat representative of Admeta's data,
- that it is straight forward to implement and repeat.

The basic idea for simulating data, which in this case is equivalent to observations, is to first construct a **true** model, m_T, describing the distribution of the number of clicks an impression gets. Secondly, complete covariates, X_i are constructed, for which the number of clicks, Y_i, can be calculated using m_T. Depending on the number of clicks, a weight w_i can be assigned and the observation is thus complete.

To create m_T a set of parameters, θ, corresponding to the model effects, is sampled from the prior distribution of each parameter, in this case a $\mathcal{N}(0, \Psi^\xi)$ distribution, where Ψ^ξ is the hyperparameter corresponding to effect ξ.

The complete covariates, X, are subsequently generated using a uniform distribution. In effect this means that for instance any of the available materials are equally likely to occur in the complete covariate X_i.

The last step is to generate the number of clicks, Y_i, each complete covariate, X_i, obtained. Since $Y_i \sim Pois(\mu_i)$, the Poisson distribution, with parameter μ_i, can be sampled in order to obtain the clicks associated with X_i. This completes the observation (Y_i, X_i). However, in order to reduce the data to store on disk and process, Admeta samples incoming impressions, throwing away those with 0 clicks with probability $1 - 1/w$, where typically $w = 100$. Impressions with clicks are very rare, contain a lot of information, and are therefore always kept. The sampling increases the concentration of clicks in the remaining data. To take this into account the $Pois(\mu_i)$ distribution can be modified slightly.

To be more precise, let $p(k)$ be the probability of an impression getting k clicks as defined by the original Poisson distribution, and let $\alpha = 1 - (1 - \frac{1}{w})p(0)$ be the approximate relative size of the set of observations after sampling. Let $p(k > 0)$ be the probability of at least 1 click and $p'(k > 0)$ be the corresponding probability after sampling, that is, $p'()$, denotes the pdf of modified Poisson distribution. Due to the total set size decreasing by approximately α, but the number of observations with more than 0 clicks

remaining constant, it follows that

$$p'(k > 0) \approx p(k > 0)/\alpha. \tag{38}$$

Equation (38) follows from that the proportion of observations with more than 0 clicks has increased by $1/\alpha$ and that probabilities can seen as proportions of the total sample. Finally, the probability of an impression having 0 clicks after sampling is given by,

$$p'(0) \approx p(0)\frac{1}{w\alpha}. \tag{39}$$

The above statement follows from the requirement that the total probability must be equal to 1. A sketch of the final data simulation algorithm is presented in Algorithm 5.

Algorithm 5 Generation of simulated data.

$N \leftarrow$ number of observations to generate.
$C \leftarrow$ number of different covariates.
$w \leftarrow$ sampling weight.
for $i = 1 \rightarrow N$ **do**
    Generate complete covariate X_i by sampling from $\mathcal{U}(0,C)$ five times.
end for
Generate true model m_T by sampling from $\mathcal{N}(0, \Psi)$.
for $i = 1 \rightarrow N$ **do**
    Calculate μ_i by using m_T and X_i.
    Use eq. (38) and (39) to sample Y_i from modified $Pois(\mu_i)$.
    if $Y_i > 0$ **then**
        $w_i = 1$.
    else
        $w_i = w$.
    end if
end for

6 Results

Three algorithms: L-BFGS, CUDA-L-BFGS and SGD, were tried and tested on problem (14), where CUDA-L-BFGS is identical to L-BFGS, except that it utilizes parallelized gradient and function evaluation. Both real and simulated observation sets were used in the tests, where the properties of the sets used are stated in Table 3.

Table 3: The observation sets used in testing. The Large-Simulated set was designed in order to test the performance of the algorithm for a larger number of parameters. The Small-Real set was used exclusively for Hessian investigations.

Test name	#Observations	#Parameters
Large-Real	13619739	32587
Small-Real	470243	18986
Large-Simulated	10000000	250000

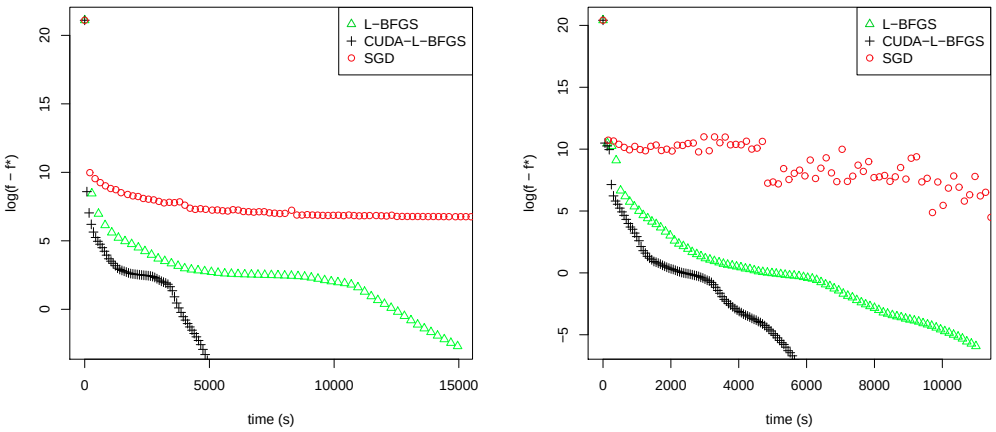
Apart from the observation sets used, the outcome also depends on the values of the hyperparameters used. Those values will be stated in connection to graphs and tables. During the entire section hyperparameters corresponding to simple effects will be denoted by Ψ_S and hyperparameters of interaction effects by Ψ_I. A final remark to be made is that the initial step length, α, used in the SGD, was set to 0.01 for the Large-Simulated data set and 0.001 for the Large-Real data set.

6.1 Performance evaluation

Figure 2a shows the progression of the three algorithms for the Large-Real data set, where $\Psi_S = 1$ for simple effects and $\Psi_I = 0.2$ for interaction effects. As can be seen the CUDA-L-BFGS performs the best, taking roughly 5000 seconds to converge, slightly less than a third of the time L-BFGS takes. The SGD was terminated after 160 passes of the data, taking 16000 seconds, with a final function value 0.045% higher than the two L-BFGS methods.

Figure 2b shows the result from running the three algorithms on the Large-Simulated data set. As can be seen, the speed up when using CUDA-L-BFGS over L-BFGS, on the Large-Simulated data set, is only about 2, compared to 3, when running on the Large-Real data set.

To investigate the effect of hyperparameters on the convergence speed, tests were run, using smaller hyperparameter values. This sped up convergence significantly, as can be seen in Table 4, which summarizes the results of the timing tests done.



(a) Progression on the Large-Real data set with $\Psi_S = 1$, $\Psi_I = 0.2$.

(b) Progression on the Large-Simulated data set with $\Psi_S = 0.1$, $\Psi_I = 0.02$.

Figure 2: Plots of algorithm progression versus time. The y-axis shows $\log(f^k - f^*)$, where f^* is taken to be the lowest function value found.

When looking at Table 4, note that function values are not directly comparable for results obtained on the same data set but using different hyperparameters as the hyperparameters affect the objective function value. The discrepancy in the number of iterations between L-BFGS and CUDA-L-BFGS, when run on the same problem, are due to numerical differences.

The objective function values found by CUDA-L-BFGS are about 0.045% lower than those found by the SGD on average. Comparing run speed, CUDA-L-BFGS reaches the lowest function value found by the SGD in about 4 minutes for the simulated data, this corresponds to a speed up of 75. However, CUDA-L-BFGS is only about 4 times quicker than the SGD if you compare the the total running times of the two algorithms.

6.2 Local convergence rate

In order to calculate the rate of convergence for the test runs in Figure 2, it was assumed that $x^* = x^K$, where x^K are the parameters of the CUDA-L-BFGS at convergence. According to Theorem 22 one can expect L-BFGS to have an R-linear convergence rate, however, the conditions posed by that theorem are not fulfilled by this problem. Nevertheless, using Definition 26, note that the rate of convergence, r, can be found by taking the logarithm,

$$\log(\|\theta^k - \theta^*\|) \leq \log(Cr^k) = \log(C) + k\log(r).$$

Table 4: Results and parameters used in the tests performed. Note that the L-BFGS and CUDA-L-BFGS results differ slightly due to numerical differences. For the SGD, iterations denote the number of steps taken by the algorithm.

data set	algorithm	Ψ_S	Ψ_I	time (s)	f	iterations
Large-Real	SGD	1	0.2	16670	2719921.133	$2.2 \cdot 10^9$
Large-Real	L-BFGS	1	0.2	16408	2719073.251	9190
Large-Real	CUDA-L-BFGS	1	0.2	5138	2719073.247	8780
Large-Real	CUDA-L-BFGS	0.01	0.002	116	2696394.081	199
Large-Real	CUDA-L-BFGS	0.1	0.02	1773	2699893.189	3019
Large-Simulated	CUDA-L-BFGS	0.1	0.02	5917	8011632.151	4720
Large-Simulated	L-BFGS	0.1	0.02	11017	8011632.153	4208
Large-Simulated	SGD	0.1	0.02	18842	8016445.668	$1.2 \cdot 10^9$

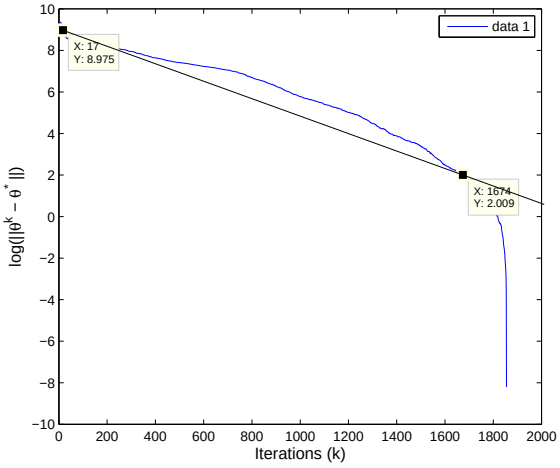


Figure 3: Plot of $\log(\|\theta^k - \theta^*\|)$ versus iterations during a run of L-BFGS on the Large-Real data set using $\Psi_S = 1$ and $\Psi_I = 0.2$.

Using the above formula Figure 3 was obtained. Figure 3 uses the Large-Real data set, with $\Psi_S = 1$ and $\Psi_I = 0.2$. As can be seen the graph is roughly linear except near the end, and from the marked values the slope can be calculated to be -0.0042. The convergence rate is thus $r = 0.9958$, which is interpreted as the approximate factor that the function value decreases by at each iteration. The sharp dip towards the end of figure 3 is due to the final iterates, x^k, being very close to x^K.

The same calculations were also done on the same set but using smaller hyperparameters, namely $\Psi_S = 0.1$ and $\Psi_I = 0.02$. The convergence rate was then found to be $r = 0.9989$, slower than the convergence rate for larger hyperparameters.

6.3 Investigating the model fit

The simulated data is useful, as the true model m_T behind it is available. Hence it is possible to compare various properties of the solution to the true model.

What interests Admeta, is mainly the accuracy of the predicted expected number of clicks μ_i. To investigate this property, the rmse (root mean square error) was calculated, where

$$rmse = \frac{\|\mu_T - \mu\|}{\sqrt{N}}. \tag{40}$$

Here, μ_T, is the vector of true expected values, μ, is the vector of fitted expected values and, N, is the number of observations.

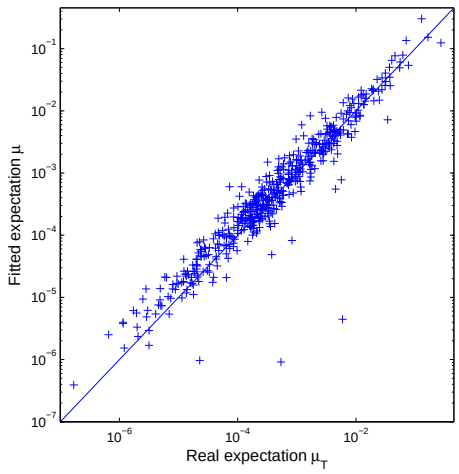
The $rmse$ was calculated for a few values of the hyperparameters and the results are presented in Table 5. As can be observed the lowest $rmse$ is achieved by the SGD. The CUDA-L-BFGS, however, is not far behind. It achieves almost the same $rmse$ score in a lot less time.

The $rmse$ scores obtained using large hyperparameters are bad, as are those using very small hyperparameters. One last thing to note in Table 5 is that the hyperparameters can probably be tuned in a bit more. Terminating CUDA-L-BFGS after 1000 iterations, corresponding to changing the convergence criterion to 0.07 instead of 0.0001, achieves as good a fit, or slightly better, than letting the algorithm run to convergence. Ideally, with the right hyperparameter values, a lower function value should also yield a better fit.

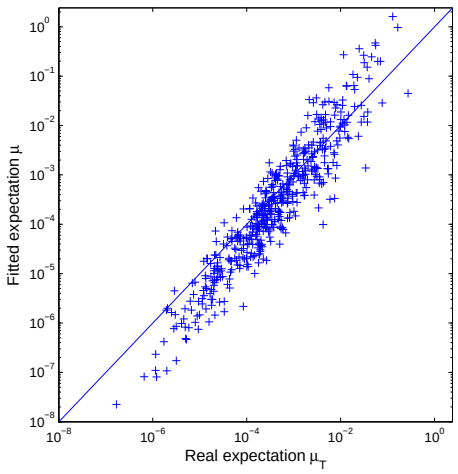
Table 5: Run times, iteration count and $rmse$ of various tests.

data set	algorithm	Ψ_S	Ψ_I	time (s)	iterations	$rmse$
Large-Simulated	L-BFGS	1	0.2	129	50	0.0964
Large-Simulated	CUDA-L-BFGS	1	0.2	27503	22000	0.1461
Large-Simulated	SGD	0.1	0.02	18842	$1.2 \cdot 10^9$	**0.0298**
Large-Simulated	CUDA-L-BFGS	0.1	0.02	5917	4720	**0.0302**
Large-Simulated	CUDA-L-BFGS	0.01	0.002	95	80	0.0762
Large-Simulated	CUDA-L-BFGS	0.1	0.02	243	200	0.0412
Large-Simulated	CUDA-L-BFGS	0.1	0.02	1250	1000	**0.0301**
Large-Simulated	CUDA-L-BFGS	0.1	0.02	2495	2000	**0.0302**

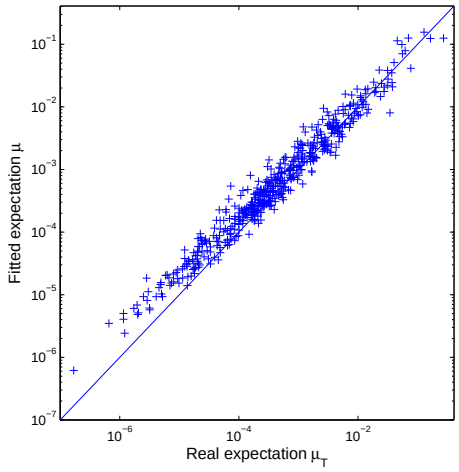
Plots of fitted expected values, μ, against the true expected values, μ_T, can be studied in Figure 4. In Figure 4b the fit is clearly the poorest. Figure 4a also suggests that the fit is poor, seeing as though there are several points where the fitted expected values are small compared to the real values. Furthermore, the SGD (Figure 4d), seems to be achieving a good fit, which is almost identical to that achieved by the CUDA-L-BFGS in Figure 4c.



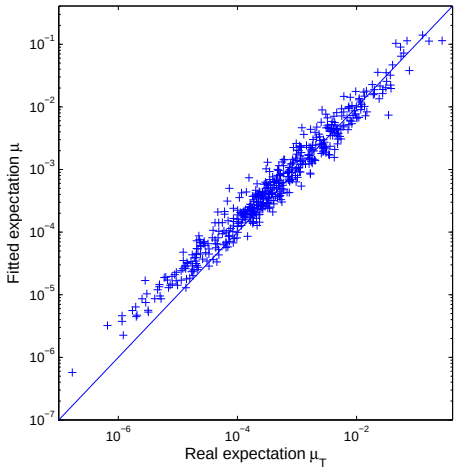
(a) Hyperparameters $\Psi_S = 1$ and $\Psi_I = 0.2$. Terminated after 50 iterations.



(b) Hyperparameters $\Psi_S = 1$ and $\Psi_I = 0.2$. Terminated after 22000 iterations.



(c) Hyperparameters $\Psi_S = 0.1$ and $\Psi_I = 0.02$. Test run until convergence, 4720 iterations.



(d) Hyperparameters $\Psi_S = 0.1$, $\Psi_I = 0.02$. Test run with SGD, 1200 passes over data.

Figure 4: Plots of μ against μ_T on a log log scale. The figures were generated using the Large-Simulated test set. Interestingly enough, Figure 4a, showing CUDA-L-BFGS running for only 50 iterations, achieves a better fit than CUDA-L-BFGS running for 22000 iterations, shown in Figure 4b. Note also the similarity of the plots obtained using CUDA-L-BFGS and SGD, Figure 4c and Figure 4d, respectively.

Other aspects of the model fit which are of interest is the convergence of the actual parameter values. Figure 5 shows how 20 randomly sampled parameter values converge toward their final value. The Large-Real data set is used, with the L-BFGS algorithm and hyperparameters $\Psi_S = 1$, $\Psi_I = 0.2$. As can be seen, some values quickly stabilize, whereas others take a long time.

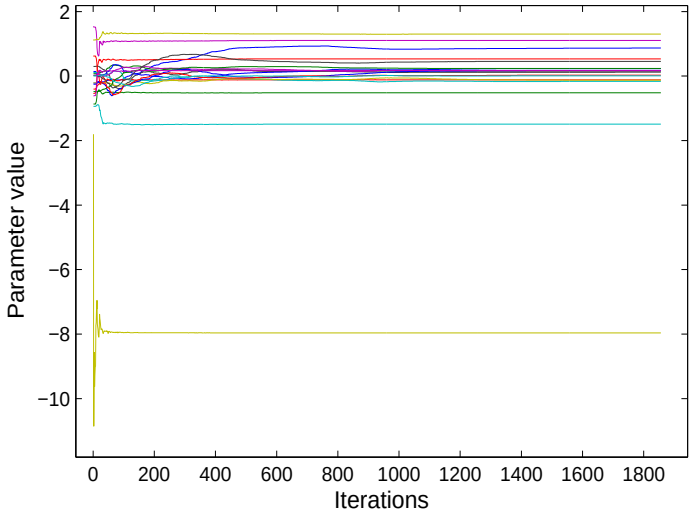
Figure 6 shows a random subset of the calculated and true parameter sets plotted against each other. Worth noting is that in Figure 6a there is a clear trend toward the absolute value of the parameters being too large. In Figure 6b on the other hand, where smaller hyperparameters are used, the trend has disappeared.

6.4 Investigating the true Hessian

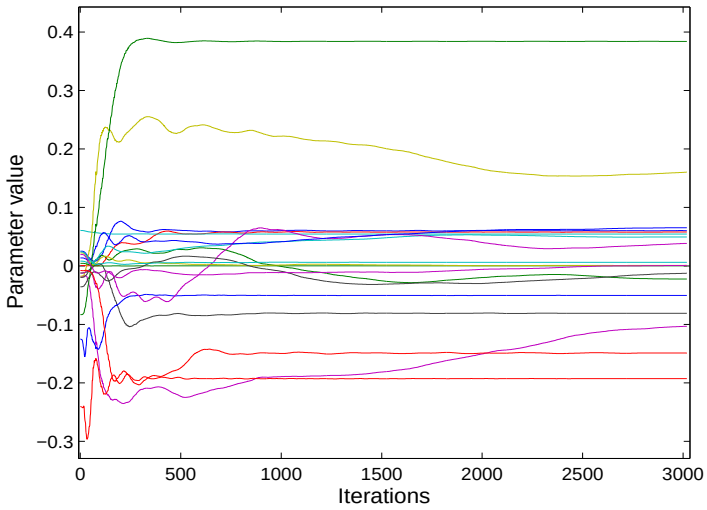
The approximate inverse Hessian B_0^{-1} plays a crucial role in the L-BFGS algorithm. At every iteration it is used as a base for the final inverse Hessian approximation. A better base, means a better end result. The B_0^{-1} currently in use is a simple scaled identity matrix. As a first step of improving that base it was decided to study the actual Hessian of eq. (14).

Figure 7 shows a heatmap of the logarithm of the absolute values of the Hessian for the Small-Real data set. The logarithm was applied in order to even out the large variations in the matrix. The Hessian was calculated numerically after 70 iterations of L-BFGS had run, and the original matrix consisted of 18,986 $\times$ 18,986 elements. Handling and plotting such a huge matrix is very cumbersome, so a reduction of the matrix dimensions was performed. A new matrix was formed by summing up the absolute value of elements in 22×22 squares in the original matrix, forming a local average, and using this as a corresponding element in the new matrix. The resulting picture is thus a 863×863 approximation of the original Hessian.

There are a few things to note in Figure 7. Firstly, as can be seen the diagonal has a high amplitude throughout. This is not surprising seeing as the diagonal gets contributions from every parameter in every observation. The empty squares along the diagonal, marked by A, B and C, correspond to the intervals of different model effects. Thus there are seven such empty squares in total, although four of them are small and hard to see. The elements inside the squares are all zero, since each observation contains only one of each covariate. Hence there can be no contribution to a Hessian position corresponding to for instance two different materials. The three easily visible squares correspond to the most numerous parameters, namely material effect parameters (A), placement effect parameters (B) and material-placement interaction parameters (C).

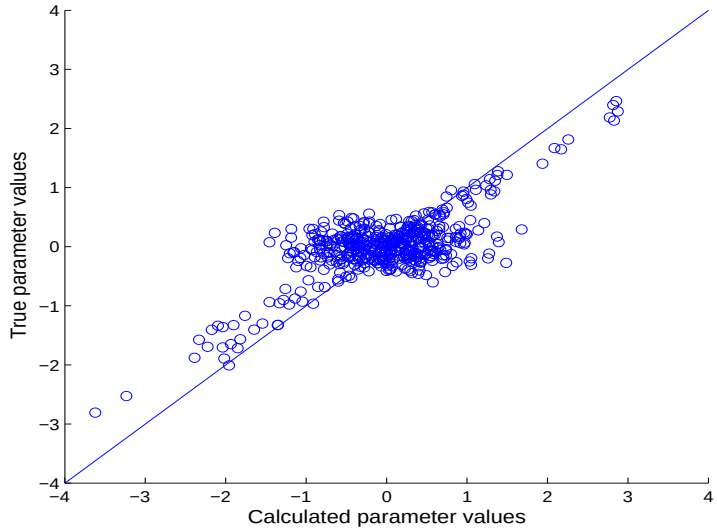


(a) Hyperparameters $\Psi_I = 1$ and $\Psi_S = 0.2$. The parameter converging to -8 is the intercept parameter.

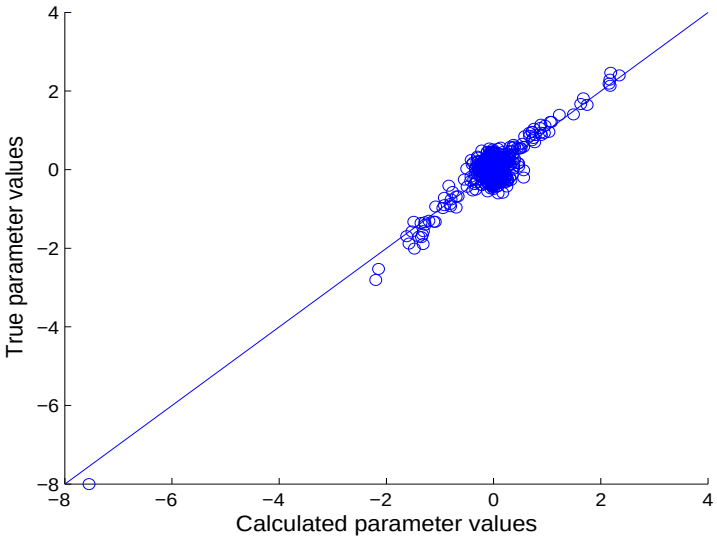


(b) Hyperparameters $\Psi_I = 0.1$ and $\Psi_S = 0.02$.

Figure 5: Two plots of the evolution of parameter values against iteration count, for two different runs of CUDA-L-BFGS on the Large-Real data set. Figure 5b is centered closer around zero and uses smaller hyperparameters, but the behaviour of the two plots seem to be roughly similar over all.



(a) Hyperparameters $\Psi_S = 1$ and $\Psi_I = 0.2$.



(b) Hyperparameters $\Psi_S = 0.1$ and $\Psi_I = 0.02$.

Figure 6: Parameters returned by CUDA-L-BFGS are plotted against the true parameters of m_T. The data set used is Large-Simulated, with different hyperparameters. As can be seen in Figure 6a there appears to be a trend in the calculated parameter values of being slightly to large in absolute value, whereas for Figure 6b, the trend in the calculated data seems to have disappeared. If anything the absolute of the calculated values are somewhat small. The two plots concern the same parameters, but the lower plot also includes the intercept parameter, which explains the different zoom.

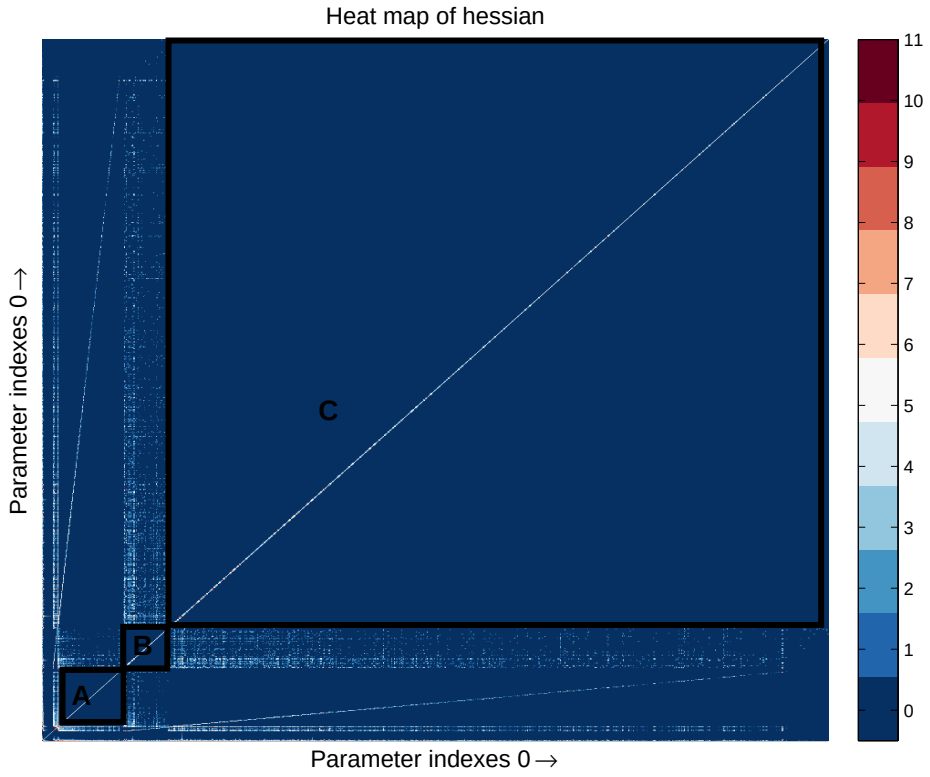


Figure 7: A heat map of the Hessian for test set Small-Real. As can be seen, the diagonal has large values. Along the diagonal are square regions (marked A, B, C) with elements equal to zero. These correspond to the different model effects.

7 Discussion and future work

From Figure 2 and Table 4, it is apparent that CUDA-L-BFGS performs better than SGD for the problem studied in this thesis. The SGD has a hard time reaching the same low function values found by the L-BFGS. This is not surprising, considering that the SGD uses such a harsh approximation of the gradient. The L-BFGS steadily, though rather slowly, converges toward some local minimum.

The best function value found for the Large-Real data set by CUDA-L-BFGS at convergence, is 0.03% lower than the best value found by the SGD; for the Large-Simulated set the corresponding difference is 0.06%. However, as can be seen from Table 5, the lower function value does not seem to have much of an effect on the model's predictive power. This may be due to the use of incorrect hyperparameters. The striking similarity of Figure 4c and Figure 4d also suggests that the difference between the fit obtained by the CUDA-L-BFGS and the SGD is not great. Over all, the performance differences between the two algorithms were not as great as had been expected, given Admeta's previous experience with SGD.

The strength of the SGD is its relatively constant run time; it scales well with the number of observations and parameters. The biggest drawback is its lack of a line search. Although this lack does make single iterations exceptionally quick, it also implies that the algorithm has difficulties reaching the smallest function values, as seen in Figure 2.

It is unclear exactly what boosts to the model performance are obtained by finding a lower function value. As mentioned in the previous paragraph the findings in this thesis suggest that the boost is small. Nevertheless, it might be that, for instance, parameters that have few observations associated with them, and therefore a small impact on the function value, are better fit using CUDA-L-BFGS. This aspect can be critical for Admeta, as it is important for them to quickly be able to make correct performance estimates for new advertisements. These naturally have few observations associated with them. In the simulated data, used for testing the properties of fitted models, observations were distributed evenly amongst the parameters, hence the above effect cannot be properly detected by the *rmse* score stated here.

It is possible that a better approach for evaluating *rmse* scores would have been to do so on a separate test data set, as is standard in statistical analysis. In such an approach a specific data set is used for fitting the data (the training set) and another data set (the test set) used for evaluating fit. Usually, such an approach is taken in order to reduce the effect of overfitting. If the model is overfitted to the training set, then it will do bad predictions for the test set. It is possible that the small differences in *rmse* seen here

between the SGD and CUDA-L-BFGS is due to them both being overfitted on the training data. Nonetheless, since the main aim of this thesis was not to investigate the model performance in general, the approach of using a training and test set was not applied.

If it is the case that CUDA-L-BFGS overfits data, a way of reducing this could be to include the *rmse* as a termination criteria. However, there is no way of calculating the *rmse* for real data, so this convergence criteria is not useful in practice. A more realistic approach at making sure that a better function value actually yields a better fit, is to make use of a multi-objective optimization, where apart from maximizing the MAP, some measure of the fit is also optimized. A solution which strikes a good balance between the two objectives can then be selected. However, such an approach requires the definition of some measure of convergence applicable also to real data. Furthermore, the optimization becomes more complicated.

The simplest way of making sure that good function values also correspond to good fits, is to adjust the model hyperparameters. That way, the optimization algorithm can focus on one thing and the model on one thing.

Selecting a good initial step length for the SGD and adjusting it accordingly is very problem dependent. Perhaps if the step length had been reduced further, the SGD would have been able to reach the same solution as the L-BFGS. Admeta needs to solve a number of optimization problems that are constantly changing, so finding an optimal SGD step length is difficult. On the other hand, m, the number of previous gradient and function values stored in the L-BFGS, is quite easy to decide upon. A value such as $m = 10$ works reasonably well for most problems. This is another advantage L-BFGS has over SGD.

Looking at Figure 2, both the SGD and L-BFGS experience rapid convergence during the first 100-200 iterations. Naturally this depends a lot on the starting position, but, the slow convergence thereafter ($r \approx 0.996$), as seen in section 6.2, suggests that the objective function is quite flat around the minimum.

This observation corresponds well to the results obtained for tests with smaller hyperparameters (see Table 4). Lower hyperparameter values correspond to an increased regularization of the parameter values. In a way, this is equivalent to the optimization problem, eq. (14), becoming less flat around the minimum since the prior contribution, θ^2/Ψ, is increased as Ψ is reduced. This yields quicker convergence.

As seen in Figure 4, Table 5 and Figure 6, the predictive power of the model also depends heavily on the hyperparameter values. Using $\Psi_S = 1$, $\Psi_I = 0.2$, caused a very bad fit of the model, which meant that the predictions were actually better when CUDA-L-BFGS was terminated after only

50 iterations compared to when letting it converge (after 22000 iterations). Lower hyperparameters, $\Psi_S = 0.1$ and $\Psi_I = 0.02$, not only yielded faster convergence of CUDA-L-BFGS, but also resulted in a lot better predictions. Reducing the hyperparameters even more makes the model too rigid, decreasing the predictive capabilities, but reducing convergence time to 2 minutes.

The choice of hyperparameters is thus critical both for the performance of prediction and the speed of convergence. From the results obtained in this thesis, it seems that using slightly too small hyperparameters is preferable to using too large. At least the rigid model obtained with small hyperparameters is made up for by quick convergence speed of CUDA-L-BFGS.

From Figure 2, one can deduce that CUDA-L-BFGS is about 3 times faster than L-BFGS on real data, but only 2 times as fast on simulated data. This difference is due mainly to the Large-Real data set having about 10 times as many material and placement effect parameters as the other effect parameters combined, and that the distribution of covariates is far from uniform. In the Large-Real data set some covariates occur a lot more frequently than others. Therefore, the sorting done of the observation set, to speed up the parallelized gradient and function evaluation, has much greater effect in increasing cache hits for the Large-Real data set. This is in contrast to the Large-Simulated data set, where the different types of covariates occur in equal numbers.

One way of improving the performance of the L-BFGS is to improve the starting inverse Hessian approximation B_0^{-1} used in every iteration. The method is called preconditioning and Veerse et al [5] and Lianjun [29] have both applied it to great effect. Most commonly, the preconditioner is some approximation of the actual inverse Hessian.

Studying Figure 7 of the Hessian, does not immediately bring to mind any good methods for approximating the inverse. The Hessian, although quite sparse, is dense in some parts. Lianjun in [29] uses a Cholesky decomposition, which requires the calculation of the Hessian every iteration. This, however, would be far too time consuming to consider for this problem. Furthermore, for problems with a lot of parameters, such as Large-Simulated, the Hessian consists of tens of billions of parameters. Handling such large matrices is difficult. Therefore, the most promising way of obtaining a better preconditioner, lies in constructing some heuristic approximation algorithm. Such a heuristic might start with diagonal matrices, but must, to truly be successful, probably go beyond that. Constructing such a preconditioner will require further studies of the Hessian and its inverse.

Future work directed toward improving the speed of the algorithm could focus on either implementing a preconditioner, as discussed in the previous paragraph, or in parallelizing the linear algebra involved. As the number of

parameters grows, multiplying and adding parameter vectors take up more of the algorithm time. One way of reducing this time is to utilize the GPU also for this purpose.

Another area, relevant to Admeta, is to investigate how to best handle small changes in the observation set, since Admeta receives new observations continuously. Can previously stored function and gradient values be utilized for quicker convergence?

Admeta are happy with the results of this thesis. They have found that the L-BFGS is easier to use than the SGD, yet there are still several things to investigate. One such thing, regarding Figure 5, is to do research on which parameters take a long time to stabilize. Related to this is the question of how much of a difference the lower function value obtained by CUDA-L-BFGS, compared to the SGD, does for prediction.

8 Conclusion

This thesis has shown that the MAP problem obtained from a large-scale Bayesian Poisson model with log link is smooth, unconstrained and can be solved using L-BFGS. By evaluating the objective function and gradient on the GPU a parallel algorithm, CUDA-L-BFGS, was obtained. Due to the iterative nature of most Large-Scale optimization algorithms this was found to be the most suitable way of parallelizing the algorithm. CUDA-L-BFGS was shown to perform about 3 times quicker than the ordinary L-BFGS implementation on real industry data.

Comparing the performance of L-BFGS to SGD is in some sense ill defined as they converge to different values. L-BFGS performs better, achieving a mean reduction of 0.05% in function value compared to the SGD, for the two data sets. Time-wise the CUDA-L-BFGS also has an edge, but if only an approximate solution is desired then the SGD can be said to perform similarly. The thesis goal of achieving a convergence speed ten times quicker than the SGD has been met (see Figure 2). CUDA-L-BFGS finds a solution, with a function value equal to the final function value found by the SGD, in approximately $\frac{1}{75}$th of the time it takes the SGD to find the solution.

L-BFGS finds a better solution than the SGD, in the sense that the objective function value is lower, however, the predicted expected values are very similar, slightly favouring the SGD. The quality of the predictions depend heavily on the hyperparameter values and it is likely that the optimal hyperparameters were not used. It was found that, for the reduced model used in the thesis, the hyperparameters $\Psi_S = 0.1$ and $\Psi_I = 0.02$, worked reasonably well. An $rmse$ of 0.0302 was obtained using CUDA-L-BFGS in conjunction with these hyperparameters, compared with 0.0298 for the SGD. This implies that the expected values obtained using SGD deviated slightly less from the true expected values than the ones obtained using CUDA-L-BFGS.

Based on the above conclusions I recommend that Admeta henceforth use the L-BFGS algorithm to fit their model to data. However, future research must also be done in order to find proper hyperparameters of the various effects. Only then will the speed improvement, and better optimization, also yield a better performing model.

References

[1] T. L. Lai, "Statistical modeling: Applications and recent advances," 2009. Seminar notes, Stanford University, Stanford, California.

[2] N. R. Draper and H. Smith, *Applied Regression Analysis*. Hoboken, New Jersey: Wiley, third ed., 1998.

[3] S. N. Wood, *Generalized Additive Models: An Introduction with R*. Boca Raton, Florida: Chapman & Hall/CRC, first ed., 2006.

[4] B. Léon, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)* (Y. Lechevallier and G. Saporta, eds.), (Paris, France), pp. 177–187, Springer, August 2010.

[5] F. Veersé, D. Auroux, and M. Fisher, "Limited-memory BFGS diagonal preconditioners for a data assimilation problem in meteorology," *Optimization and Engineering*, vol. 1, no. 3, pp. 323–339, 2000.

[6] J. Morales, "A numerical study of limited memory BFGS methods," *Applied Mathematics Letters*, vol. 15, no. 4, pp. 481–487, 2002.

[7] B. Das, H. Meirovitch, and I. M. Navon, "Performance of hybrid methods for large-scale unconstrained optimization as applied to models of proteins," *Journal of Computational Chemistry*, vol. 24, no. 10, pp. 1222–1231, 2003.

[8] N. Andréasson, A. Evgrafov, and M. Patriksson, *An Introduction to Continuous Optimization*. Studentlitteratur, first ed., 2005.

[9] D. P. Bertsekas, *Nonlinear Programming*. Belmont, Massachusetts: Athena Scientific, second ed., 2008.

[10] W. Sun and Y.-X. Yuan, *Optimization Theory and Methods: Nonlinear Programming*. New York, NY 10013, USA: Spinger, 2006.

[11] J. F. Bonnans, J. C. Gilbert, C. Lemaréchal, and C. A. Sagastizábal, *Numerical Optimization: Theoretical and Practical Aspects*. Berlin Heidelberg: Springer, second ed., 2006.

[12] C. Broyden, "The convergence of a class of double-rank minimization algorithms," *Journal of the Institue of Mathematics and its Applications*, vol. 6, pp. 76–90, 1970.

[13] R. Fletcher, "A new approach to variable metric algorithms," *The Computer Journal*, vol. 13, no. 3, pp. 317–322, 1970.

[14] D. Goldfarb, "A family of variable-metric methods derived by variational means," *Mathematics of Computation*, vol. 23, pp. 23–26, 1970.

[15] D. F. Shanno, "Conditioning of quasi-Newton methods for function minimization," *Mathematics of Computation*, vol. 24, no. 111, pp. 647–656, 1970.

[16] A. Perry, "A class of conjugate gradient algorithms with a two-step variable metric memory," tech. rep., Graduate School of Management, Northwestern University, Evanston, Illinois, 60201, 1977.

[17] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Mathematical Programming*, vol. 45, pp. 503–528, Dec. 1989.

[18] N. Gould, D. Orban, and Ph. Toint, "Numerical methods for large-scale nonlinear optimization," *Acta Numerica*, vol. 14, pp. 299–361, 2005.

[19] A. Lindbäck, "Efficient GPU implementation of parameter estimation of a statistical model for online advertisement optimization." Master's thesis, 2012.

[20] F. Lootsma and K. Ragsdell, "State-of-the-art in parallel nonlinear optimization," *Parallel Computing*, vol. 6, no. 2, pp. 133–155, 1988.

[21] P. K.-H. Phua, W. Fan, and Y. Zeng, "Parallel algorithms for large-scale nonlinear optimization.," *International Transactions in Operational Research*, vol. 5, no. 1, pp. 67–77, 1998.

[22] A. Migdalas, G. Toraldo, and V. Kumar, "Nonlinear optimization and parallel computing," *Parallel Computing*, vol. 29, no. 4, pp. 375–391, 2003.

[23] R. B. Schnabel, "A view of the limitations, opportunities, and challenges in parallel nonlinear optimization," *Parallel Computing*, vol. 21, pp. 875–905, 1995.

[24] V. Galiano, H. Migallón, V. Migallón, and J. Penadós, "GPU-based parallel algorithms for sparse nonlinear systems," *Journal of Parallel and Distributed Computing*, 2011.

[25] Y. Censor and S. A. Zenios, *Parallel Optimization Theory, Algorithms, and Applications*. USA: Oxford University Press, first ed., 1997.

[26] T. A. Straeter, "A parallel variable metric optimization algorithm," tech. rep., NASA, Langley Research Center, Hampton, Virginia, 23665, 1973.

[27] P. Jog, J. Y. Suh, and D. van Gucht, "Parallel genetic algorithms applied to the traveling salesman problem," *SIAM Journal of Optimization*, vol. 1, pp. 515–529, 1991.

[28] V. T. J. E. Dennis JR., "Direct search methods on parallel machines," *Siam Journal of Optimization*, vol. 1, pp. 448–474, 1991.

[29] L. Jiang, *Preconditioning the limited-memory BFGS algorithm*. PhD thesis, University of Colorado at Boulder, Boulder, CO, USA, 2006.

A Further optimization theory

A.1 Convergence rate

One way of ranking algorithms is through theoretical measures of convergence rate. There are several ways of defining convergence rate for an optimization algorithm. The route taken in most mathematical texts is the path of local convergence analysis, which concerns the behaviour of the algorithm close to a solution. Another possibility is to consider the number of operations needed until convergence, or the number of function and gradient calls required.

To quantify the rate of convergence in local convergence analysis $\|x - x^*\|$ or $|f(x) - f(x^*)|$ is compared to known sequences. For instance an algorithm satisfying

$$\|x^k - x^*\| < q\beta^k, \tag{41}$$

for some $\beta \in (0,1)$ and $q > 0$ is said to converge at Q-linear speed. The distance to the solution is reduced by β at each step. If, on the other hand, iterates x^k of an algorithm satisfy (41) for any $\beta > 0$, then it is said to converge superlinearly. In practice, alternative characterisations are used to classify different modes of convergence. They are stated below for Q-convergence (quotient-convergence) along with a definition of R-linear convergence speed for reference.

Definition 23 (Characterisation of Q-linear convergence rate)**.** An algorithm converges Q-linearly if

$$\lim_{k\to\infty} \sup \frac{\|x^{k+1} - x^*\|}{\|x^k - x^*\|} = \beta, \tag{42}$$

for some $\beta \in (0,1)$.

Definition 24 (Characterisation of Q-super-linear convergence rate)**.** An algorithm converges Q-super-linearly if

$$\lim_{k\to\infty} \sup \frac{\|x^{k+1} - x^*\|}{\|x^k - x^*\|} = 0. \tag{43}$$

Definition 25 (Characterisation of Q-quadratic convergence rate)**.** An algorithm converges Q-quadratically if

$$\lim_{k\to\infty} \sup \frac{\|x^{k+1} - x^*\|}{\|x^k - x^*\|^2} = \beta, \tag{44}$$

for some $\beta \in (0,1)$. A Q-quadratic convergence is roughly equivalent to the number of correct digits being doubled at every iteration.

Definition 26 (Characterisation of R-linear convergence rate)**.** An algorithm converges R-linearly if there exists a sequence of nonnegative scalars q_k such that,

$$\|x^k - x^*\| \leq q_k \, \forall k, \tag{45}$$

where q_k converges Q-linearly to 0. Hence $q_k < Cr^k$, where C is some constant. Note that R-linear convergence is weaker than Q-linear convergence.