TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Study of Gradient and Stochastic Gradient Algorithms for Logistic Regression

by

Emmanouil Limnaios

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DIPLOMA DEGREE OF

ELECTRICAL AND COMPUTER ENGINEERING

February 13, 2023

THESIS COMMITTEE

Professor Athanasios P. Liavas, *Thesis Supervisor*
Professor George N. Karystinos
Associate Professor Vasilis Samoladas

Abstract

In this Diploma thesis, we study the Logistic Regression (LR), which is a widely used method for classification. We start by presenting the regularized LR cost function and computing its gradient and Hessian. It is well known that the LR is a convex function. Our main aim is to study the performance (convergence speed and solution accuracy) of deterministic versus stochastic algorithms for the minimization of the regularized LR cost function. First, we present two variants of the deterministic (full) gradient algorithm, one with a "naive" step-size and one with backtracking line search. Next, we move to the (Nesterov-type) accelerated full gradient algorithm. Then, we present variants of the stochastic gradient descent with step-sizes computed by various methods. For example, (1) by exploiting the strong convexity property of the regularized LR, (2) by using Armijo line-search using only a subset of the data determined by the batch size, (3) by using an ad-hoc line-search based on the angle of two successive stochastic gradients, etc. We test the performance of the various algorithms by using synthetic data (linearly separable and linearly non-separable). We observe that some stochastic variants (especially the variant which exploits the strong convexity of the regularized LR) perform quite well during the first epochs, while the accelerated gradient algorithms become more accurate after the first epochs. In general, accelerated stochastic gradient-type algorithms are fast during the first epochs but not very accurate. Thus, more sophisticated accelerated stochastic algorithms must be pursued.

Acknowledgements

First of all, I would like to thank my thesis supervisor, Professor Athanasios Liavas, for introducing me to the field of Optimization through his courses. I would like to also thank him for the inspiration, guidance and support throughout this work.

I would like to thank all the glorious people that I met throughout my time as a student and am lucky to call friends. Especially, I would like to thank my very close friend Giannis V. and my best study buddies Matthaios S. and Dimitra K.. Lastly, many thanks to Eleni B. for her continuous support up to this point in my academic career.

Last, but not least, I would like to thank my family for their unquestionable support and encouragement throughout my study years.

Table of Contents

List of Figures

List of Abbreviations

LR	Logistic Regression
GD	Gradient Descent
btGD	Gradient Descent With Backtracking Line Search
NAG	Nesterov Accelerated Gradient
SGD	Stochastic Gradient Descent
SGDSC	Stochastic Gradient Descent with Strong Convexity Loss
SGD+Armijo	Stochastic Gradient Descent with Armijo Backtracking Line Search
NAG+Armijo	Nesterov Accelerated Gradient with Armijo Backtracking Line Search
RACDM	Random Adaptive Coordinate Descent Method
ADAM	Adaptive Moment Estimation Algorithm

Chapter 1

Introduction

1.1 Logistic Regression

The Logistic Regression is a supervised learning model used for classification (and not regression analysis, despite its name). It is not uncommon to see the LR model used as the activation function of a neural network end-node, to classify or infer on the final outcome [8], [9].

From an optimization perspective, we view the LR cost function as a convex function, which we must minimize [4]. Using techniques such as regularization, we augment the cost function in order to improve the model's accuracy [2], [3].

1.2 Purpose

In this thesis, we minimize the regularized Logistic Regression cost function using various algorithms. Our aim is to determine which are most suitable for the solution of the problem under different conditions. These conditions regard different categories of data (linearly separable versus linearly non-separable), regularization parameter values and mini-batch sizes. We test various line search methods and exploit cost function properties to improve the convergence of certain algorithms.

1.3 Notation

Vectors are denoted by small bold letters and matrices are denoted by capital bold letters, for example, $\mathbf{x}, \mathbf{X}$. Elements of vectors or matrices are denoted by small, non-bold letters with appropriate sub-scripts, such as $x_{i,j}$.

Sets are denoted by blackboard bold capital letters; for example, $\mathbb{R}$ denotes the set of real numbers. $\mathbb{R}^n$ denotes the n-dimensional vector space of real numbers. The identity matrix of size $n \times n$ is denoted by $\mathbf{I}_n$ (we omit the subscript n whenever it is clear from the context).

The gradient of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix}.$$

The Hessian matrix of a twice differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is defined as

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n \partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{bmatrix}.$$

The Euclidean norm of a vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is defined as

$$\|\mathbf{x}\| = \|\mathbf{x}\|_2 = \sqrt{x_1^2 + \cdots + x_n^2}.$$

1.4 Thesis Outline

This thesis is organized as follows:

- In Chapter 2, we define the Logistic Regression cost function and compute its gradient and Hessian.
- In Chapter 3, we present full gradient-type algorithms that are used for the solution of our problem.
- In Chapter 4, we present variants of stochastic gradient algorithms for the solution of our problem.
- In Chapter 5, we experimentally study the convergence characteristics of the algorithms studied.
- In Chapter 6, we make conclusions about the algorithms studied and make remarks on possible future work.

Chapter 2

Problem Formulation

In this chapter, we define the Logistic Regression cost function and compute its gradient and Hessian, which are used in the algorithms we shall study. Then, we characterize the Logistic Regression cost function in terms of convexity.

2.1 Definitions

2.1.1 Hypothesis Function

The hypothesis function used for Logistic Regression is the Logistic Function. The name is derived from its functionality, as it converts the logarithm of the odds (the log-odds) of the two possible outcomes of our problem into probability [4].

To model these outcomes, we use a binary logistic model whose possible values are "1" and "0". These values can be considered as two different classes we try to fit the given data $\mathbf{x}$ into or the probability that the given data belong in a particular class. We call these values the *labels* of our problem and are represented by the binary variable y. We show how these labels are derived, using the Logistic Function, in a following subsection [15], [4].

The Logistic Function is the binary logistic model used to model the binary dependent variable mentioned, or more accurately, to infer the probability that the data belong in a certain class [15], [4].

Definition 2.1.1. The Logistic Function has the general form:

$$h(\mathbf{x}) := \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x}+w_0)}} = \frac{e^{(\mathbf{w}^T\mathbf{x}+w_0)}}{1 + e^{(\mathbf{w}^T\mathbf{x}+w_0)}}. \tag{2.1}$$

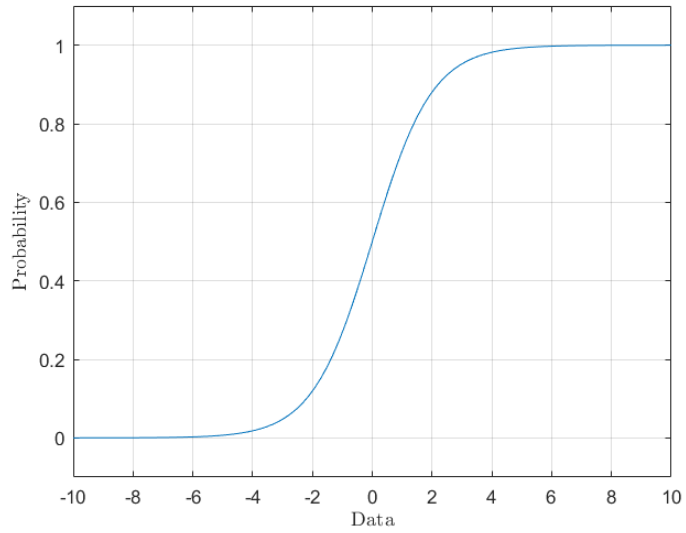


Figure 2.1: Plot of Logistic Function, where $\mathbf{w} = \mathbf{1}$ and $w_0 = 0$.
Data are represented by a 1-dimensional vector.

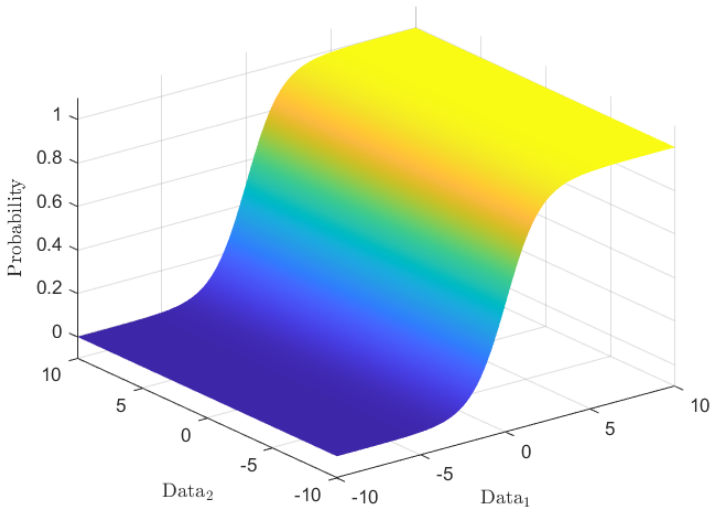


Figure 2.2: Plot of Logistic Function, where $\mathbf{w} = \mathbf{1}$ and $w_0 = 0$.
Data are represented by a 2-dimensional vector.

2.1.2 Parameters

By parameters, we refer to a vector $\boldsymbol{\theta}$ which we try to compute by minimizing the LR cost function using iterative methods, such as the optimization algorithms we are about to study.

Definition 2.1.2. We define the parameters of the problem as the augmented vector

$$\boldsymbol{\theta} := (w_0, \mathbf{w}), \tag{2.2}$$

which contains the constant intercept w_0 of the Logistic Function as well a weight vector $\mathbf{w}$. Without loss of generality, we will call vector $\boldsymbol{\theta}$ the *weight vector* of the problem [4].

In the end of the minimization procedure, the weight vector $\boldsymbol{\theta}$ will contain the *optimal solution* of the problem.

Definition 2.1.3. Having defined the weight vector $\boldsymbol{\theta}$, we define the simpler Logistic Function form we study in this thesis:

$$h_{\boldsymbol{\theta}}(\mathbf{x}) := \frac{1}{1 + e^{-(\boldsymbol{\theta}^T \mathbf{x})}}, \tag{2.3}$$

which is parameterized by $\boldsymbol{\theta}$.

The value of $h_{\boldsymbol{\theta}}(\mathbf{x})$ is considered to be the probability that the sample $\mathbf{x}$ belongs in the first binary class and the value of $1 - h_{\boldsymbol{\theta}}(\mathbf{x})$ that it belongs in the other class.

2.1.3 Linear Classifier

Logistic Regression leads to a linear classifier. Linear classifiers are the classifiers whose decision boundaries are *hyperplanes* [13].

Definition 2.1.4. The decision boundary for binary classification using LR can be computed as:

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = 1 - h_{\boldsymbol{\theta}}(\mathbf{x}), \tag{2.4}$$

meaning that the two classes have the same probability.

That leads us to

$$\frac{1}{1 + e^{-(\boldsymbol{\theta}^T \mathbf{x})}} = \frac{e^{-(\boldsymbol{\theta}^T \mathbf{x})}}{1 + e^{-(\boldsymbol{\theta}^T \mathbf{x})}} \Leftrightarrow e^{-(\boldsymbol{\theta}^T \mathbf{x})} = 1 \Leftrightarrow \boldsymbol{\theta}^T \mathbf{x} = 0. \tag{2.5}$$

Equation (2.5) defines the hyperplane that separates the two classes of our problem.

Thus, if $\boldsymbol{\theta}^T \mathbf{x} > 0$, the model classifies sample $\mathbf{x}$ in the first class, otherwise, if $\boldsymbol{\theta}^T \mathbf{x} \leq 0$, classifies it in the other class [13].

2.2 Cost Function

In the last section, we talked about the function we use for classification. In order to fit the data as accurately as possible, we must compute the optimal values of the weight vector $\boldsymbol{\theta}$.

To find these values, we must use a cost function that maximizes the probability of the odds that the samples belong in a certain class, in the sense that the probability inferred by the hypothesis function maximizes the likelihood of these odds.

2.2.1 Definition

Definition 2.2.1. We define the cost function as

$$J(\boldsymbol{\theta}) := \sum_{k=1}^{n} -y_k \boldsymbol{\theta}^T \mathbf{x}_k + \log\left(1 + e^{-\boldsymbol{\theta}^T \mathbf{x}_k}\right). \tag{2.6}$$

This cost function is called the *negative log-likelihood* cost function or, as it is called in machine learning literature, the *(Binary) Cross-Entropy* cost function [4].

The cost function in equation (2.6) is derived from the form [4]

$$J(\boldsymbol{\theta}) = \sum_{k=1}^{n} -\left\{y_k \log(h_{\boldsymbol{\theta}}(\mathbf{x}_k)) + (1 - y_k)\log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k))\right\}. \tag{2.7}$$

This form can be used to intuitively justify our choice of cost function, as shown below. This choice can be further clarified by studying its probabilistic interpretation from [4].

Since y_k takes binary values, and without loss of generality, $h_{\boldsymbol{\theta}}(\mathbf{x}_k)$ also takes binary values, we have that:

$$y_k \log(h_{\boldsymbol{\theta}}(\mathbf{x}_k)) = \begin{cases} 0, & \text{if } y_k = 1 \text{ and } h_{\boldsymbol{\theta}}(\mathbf{x}_k) = 1, \\ -\infty, & \text{if } y_k = 1 \text{ and } h_{\boldsymbol{\theta}}(\mathbf{x}_k) = 0, \end{cases}$$

$$(1 - y_k)\log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k)) = \begin{cases} 0, & \text{if } y_k = 0 \text{ and } h_{\boldsymbol{\theta}}(\mathbf{x}_k) = 0, \\ -\infty, & \text{if } y_k = 0 \text{ and } h_{\boldsymbol{\theta}}(\mathbf{x}_k) = 1. \end{cases}$$

What we can interpret from the above is that this cost function takes very large, or infinitely large, values if the inferred label $h_{\boldsymbol{\theta}}(\mathbf{x}_k)$ does not match the true label y_k. By minimizing this cost function, we are forced to find such inferred labels so that all the data samples $\mathbf{x}_k$ are as closely matched with the true labels y_k as possible [4].

2.2.2 Convexity

In this section, we prove the convexity of the cost function by proving that the two terms that make up the cost function are convex.

From equation (2.6), we observe that the term

$$-y_k \boldsymbol{\theta}^T \mathbf{x}_k \tag{2.8}$$

is linear, thus, it is convex. In order to prove convexity for the cost function, we need to further prove that the term

$$\log(1 + e^{-\boldsymbol{\theta}^T \mathbf{x}_k}) \tag{2.9}$$

is also convex. To easier prove its convexity, we reform the cost function utilizing its form in equation (2.7) as follows:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \sum_{k=1}^{n} - \{y_k \log(h_{\boldsymbol{\theta}}(\mathbf{x}_k)) + (1 - y_k) \log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k))\} \\ &= \sum_{k=1}^{n} - \left\{ y_k \left(\frac{h_{\boldsymbol{\theta}}(\mathbf{x}_k)}{1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k)} \right) + \log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k)) \right\} \\ &= \sum_{k=1}^{n} - \left\{ y_k \boldsymbol{\theta}^T \mathbf{x}_k + \log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k)) \right\}, \end{aligned} \tag{2.10}$$

so we have that $\log(1 + e^{-\boldsymbol{\theta}^T \mathbf{x}_k}) = -\log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k))$ [4].

First, we compute the gradient of the function in equation (2.9), utilizing (2.10), as

$$\begin{aligned} \sum_{k=1}^{n} \nabla[-\log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k))] &= \sum_{k=1}^{n} -\nabla \left[\log \left(1 - \frac{1}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}_k}} \right) \right] \\ &= \sum_{k=1}^{n} -\nabla \left[\log \left(\frac{e^{-\boldsymbol{\theta}^T \mathbf{x}_k}}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}_k}} \right) \right] \\ &= \sum_{k=1}^{n} -\nabla \left[-\boldsymbol{\theta}^T \mathbf{x}_k - \log(1 + e^{-\boldsymbol{\theta}^T \mathbf{x}_k}) \right] \\ &= \sum_{k=1}^{n} \mathbf{x}_k + \nabla \left[\log(1 + e^{-\boldsymbol{\theta}^T \mathbf{x}_k}) \right] \\ &= \sum_{k=1}^{n} \mathbf{x}_k + \left(\frac{-e^{-\boldsymbol{\theta}^T \mathbf{x}_k}}{1 + e^{-\boldsymbol{\theta}^T \mathbf{x}_k}} \right) \mathbf{x}_k \\ &= \sum_{k=1}^{n} h_{\boldsymbol{\theta}}(\mathbf{x}_k) \mathbf{x}_k. \end{aligned} \tag{2.11}$$

The Hessian of the function in equation (2.9) is

$$\begin{aligned}\sum_{k=1}^{n}\nabla^2[-\log(1-h_{\boldsymbol{\theta}}(\mathbf{x}_k))] &= \sum_{k=1}^{n}\nabla[h_{\boldsymbol{\theta}}(\mathbf{x}_k)\mathbf{x}_k]\\ &= \sum_{k=1}^{n}\nabla\left[\left(\frac{1}{1+e^{-\boldsymbol{\theta}^T\mathbf{x}_k}}\right)\mathbf{x}_k\right]\\ &= \sum_{k=1}^{n}\left(\frac{1}{(1+e^{-\boldsymbol{\theta}^T\mathbf{x}_k})^2}\right)\left(-e^{-\boldsymbol{\theta}^T\mathbf{x}_k}\right)\mathbf{x}_k\mathbf{x}_k^T\\ &= \sum_{k=1}^{n}\left(\frac{1}{1+e^{-\boldsymbol{\theta}^T\mathbf{x}_k}}\right)\left(1-\frac{1}{1+e^{-\boldsymbol{\theta}^T\mathbf{x}_k}}\right)\mathbf{x}_k\mathbf{x}_k^T\\ &= \sum_{k=1}^{n}h_{\boldsymbol{\theta}}(\mathbf{x}_k)[1-h_{\boldsymbol{\theta}}(\mathbf{x}_k)]\mathbf{x}_k\mathbf{x}_k^T. \end{aligned} \tag{2.12}$$

Now we must prove that the Hessian in equation (2.12) is a positive semi-definite matrix in order for the term in equation (2.9) to be convex. For any vector $\mathbf{v}\in\mathbb{R}^N$

$$\begin{aligned}&\mathbf{v}^T\nabla^2[-\log(1-h_{\boldsymbol{\theta}}(\mathbf{x}))]\mathbf{v} =\\ &\mathbf{v}^T[h_{\boldsymbol{\theta}}(\mathbf{x})[1-h_{\boldsymbol{\theta}}(\mathbf{x})]\mathbf{x}\mathbf{x}^T]\mathbf{v} =\\ &(h_{\boldsymbol{\theta}}(\mathbf{x})[1-h_{\boldsymbol{\theta}}(\mathbf{x})])\|\mathbf{v}^T\mathbf{x}\|^2\geq 0,\end{aligned} \tag{2.13}$$

where the last inequality holds because

$$0\leq h_{\boldsymbol{\theta}}(\mathbf{x})\leq 1 \Longrightarrow h_{\boldsymbol{\theta}}(\mathbf{x})[1-h_{\boldsymbol{\theta}}(\mathbf{x})]\geq 0. \tag{2.14}$$

Hence, the cost function is *convex* as a sum of convex functions.

2.2.3 Gradient of Cost Function

The gradient vector of the cost function has the following form:

$$\nabla[J(\boldsymbol{\theta})] = -\sum_{k=1}^{n}y_k\mathbf{x}_k + h_{\boldsymbol{\theta}}(\mathbf{x}_k)\mathbf{x}_k = \sum_{k=1}^{n}\mathbf{x}_k\left(h_{\boldsymbol{\theta}}(\mathbf{x}_k)-y_k\right). \tag{2.15}$$

2.2.4 Hessian of Cost Function

The Hessian matrix of the cost function has the following form:

$$\nabla^2[J(\boldsymbol{\theta})] = \sum_{k=1}^{n}\nabla[-y_k\mathbf{x}_k] + \sum_{k=1}^{n}\nabla[h_{\boldsymbol{\theta}}(\mathbf{x}_k)\mathbf{x}_k] = \sum_{k=1}^{n}h_{\boldsymbol{\theta}}(\mathbf{x}_k)[1-h_{\boldsymbol{\theta}}(\mathbf{x}_k)]\mathbf{x}_k\mathbf{x}_k^T. \tag{2.16}$$

2.3 Regularization

When minimizing a cost function, often noise is present in the data. By noise we refer to actual noise in the data or samples which do not represent the properties of the majority of samples. When using such data, the resulting weight vector dictates a more flexible model for the problem but presents an added risk that the model can be easily susceptible to errors when using less noisy data. In machine learning literature, this phenomenon is called *overfitting* [3], [6].

By adding a regularization term to our cost function, we regularize the risk of this phenomenon occuring during optimization, decreasing the flexibility of our model, by bounding its ability to missclassify data samples. Smaller values for the regularization constant λ make the model less flexible by penalizing samples that affect the model parameters negatively. Larger values for the regularization constant λ make the model more flexible, instead [6].

In our model, we use a regularization technique which uses the l_2-norm of the weight vector [6].

Definition 2.3.1. The regularization used in our model has the following form:

$$\frac{1}{2}\lambda\sum_{k=1}^{n}\theta_k^2 = \frac{1}{2}\lambda\|\boldsymbol{\theta}\|^2 = \frac{1}{2}\lambda\boldsymbol{\theta}^T\boldsymbol{\theta}. \tag{2.17}$$

2.4 Regularized Cost Function

When adding regularization to the cost function, we get the following form:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \frac{1}{n}\sum_{k=1}^{n}J(\theta_k) + \frac{1}{2}\lambda\boldsymbol{\theta}^T\boldsymbol{\theta} \\ &= \frac{1}{n}\sum_{k=1}^{n}\left(-y_k\boldsymbol{\theta}^T\mathbf{x}_k + \log\left(1+e^{-\boldsymbol{\theta}^T\mathbf{x}_k}\right)\right) + \frac{1}{2}\lambda\boldsymbol{\theta}^T\boldsymbol{\theta}. \end{aligned} \tag{2.18}$$

The gradient vector of the regularized cost function is

$$\begin{aligned} \nabla[J(\boldsymbol{\theta})] &= -\frac{1}{n}\sum_{k=1}^{n}y_k\mathbf{x}_k + h_{\boldsymbol{\theta}}(\mathbf{x}_k)\mathbf{x}_k + \lambda\boldsymbol{\theta} \\ &= \frac{1}{n}\sum_{k=1}^{n}\mathbf{x}_k(h_{\boldsymbol{\theta}}(\mathbf{x}_k) - y_k) + \lambda\boldsymbol{\theta}, \end{aligned} \tag{2.19}$$

and the Hessian matrix of the regularized cost function is

$$\begin{aligned} \nabla^2[J(\boldsymbol{\theta})] &= \frac{1}{n}\left(\sum_{k=1}^{n}\nabla[-y_k\mathbf{x}_k] + \sum_{k=1}^{n}\nabla[h_{\boldsymbol{\theta}}(\mathbf{x}_k)\mathbf{x}_k]\right) + \lambda\mathbf{I} \\ &= \frac{1}{n}\sum_{k=1}^{n}h_{\boldsymbol{\theta}}(\mathbf{x}_k)[1-h_{\boldsymbol{\theta}}(\mathbf{x}_k)]\mathbf{x}_k\mathbf{x}_k^T + \lambda\mathbf{I}. \end{aligned} \tag{2.20}$$

2.5 Strong Convexity

As proven in section 2.2.2, we know that the cost function in equation (2.6) is convex, but it may not be strongly convex. Adding the regularization term $\lambda \boldsymbol{\theta}^T \boldsymbol{\theta}$, the cost function becomes λ-strongly convex [18].

This is an important property, which we can be used to our advantage, since certain algorithms make use of that property to improve their performance.

Chapter 3

Full Gradient Algorithms

In this chapter, we present gradient descent based algorithms, where the gradient vector is computed using the whole data-set D. Such algorithms are also called full gradient algorithms. Data-set D is comprised of pairs $(\mathbf{x}_i, y_i)$, with $\mathbf{x}_i$ being a data sample and y_i its label.

We start with the Gradient Descent algorithm and continue with more advanced algorithms that make use of momentum and acceleration. We study their descent steps, derive the best step-sizes and present their convergence rates. We present a simple method to compute an upper bound for the Lipschitz constant L, which is used in the derivation of the step-size of many algorithms. Lastly, we mention the improvements in performance for these algorithms when optimizing μ-strongly convex cost functions.

3.1 Gradient Descent

Gradient Descent is one of the most notable, if not the most notable, of all optimization algorithms, as it sees big usage because of its simplicity and robustness. It is a first-order iterative method for optimizing differentiable cost functions by finding their local or global minima. The algorithm was proposed long before the era of modern computers, generally attributed to Cauchy, who first suggested it in 1847 [5].

As it was increasingly studied and used, it has led to numerous improved versions, as needs shifted to other directions in the field of optimization. We must note that it is the cornerstone of many first-order methods in optimization, as many notable algorithms are variants of GD, by using its descent step in many different forms, to solve differently formulated problems or to achieve better convergence rates.

3.1.1 Descent Step

The basic idea for the descent step of the algorithm is that we must take the steepest descent step to the minimum, when optimizing convex functions. To make the steepest possible descent in the direction of the minimum, we choose to take a step in the opposite direction of the gradient of the cost function, that is $-\nabla J(\boldsymbol{\theta})$.

Definition 3.1.1. The descent step of the Gradient Descent algorithm is as follows:

$$\boldsymbol{\theta}_{k+1} := \boldsymbol{\theta}_k - \alpha_k \nabla J(\boldsymbol{\theta}_k), \tag{3.1}$$

where α_k is the step-size and $\nabla J(\boldsymbol{\theta}_k)$ is the gradient of the cost function in iterate k [12].

3.1.2 Step-size

We can compute the optimal step-size for GD if we exploit some important properties of the cost function. By the term optimal we refer to a step-size that can achieve the best rate of convergence. We derive the following results in this subsection based on [3], [11] and [17].

First, let us define L-smoothness for convex functions.

Definition 3.1.2. A (convex) function f is called L-smooth on the sample space $\mathbb{X}$ if, for any $\mathbf{x}, \mathbf{y} \in \mathbb{X}$, it holds true that

$$\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\|. \tag{3.2}$$

If the function is twice differentiable, that is the Hessian $\nabla^2 f(\mathbf{x})$ exists on $\mathbb{X}$, then equation (3.2) is equivalent to

$$\|\nabla^2 f(\mathbf{x})\| \leq L, \quad \forall \mathbf{x} \in \mathbb{X} \Longleftrightarrow |\lambda_{\max}(\nabla^2 f(\mathbf{x}))| \leq L, \quad \forall \mathbf{x} \in \mathbb{X}. \tag{3.3}$$

If we suppose that function f is L-smooth for some L, then for any $\mathbf{x}, \mathbf{y} \in \mathbb{X}$, it holds true that

$$\begin{aligned} f(\mathbf{y}) &= f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) + \frac{1}{2}(\mathbf{y} - \mathbf{x})^T \nabla^2 f(\mathbf{w})(\mathbf{y} - \mathbf{x}) \\ &\leq f(\mathbf{x}) + \nabla f(\mathbf{x})^T(\mathbf{y} - \mathbf{x}) + \frac{L}{2}\|\mathbf{y} - \mathbf{x}\|^2, \end{aligned} \tag{3.4}$$

for some $\mathbf{w} = \lambda\mathbf{x} + (1 - \lambda)\mathbf{y}$ for $0 \leq \lambda \leq 1$.

Using (3.4) and the GD descent step in (3.1), we obtain

$$\begin{aligned} f(\mathbf{x}_{k+1}) &\leq f(\mathbf{x}_k) - \alpha_k \|\nabla f(\mathbf{x}_k)\|^2 + \frac{L\alpha_k^2}{2}\|\nabla f(\mathbf{x}_k)\|^2 \\ &= f(\mathbf{x}_k) - \alpha_k \left(1 - \frac{L\alpha_k}{2}\right) \|\nabla f(\mathbf{x}_k)\|^2. \end{aligned} \tag{3.5}$$

Thus the function decrement is

$$f(\mathbf{x}_k) - f(\mathbf{x}_{k+1}) \geq \alpha_k \left(1 - \frac{L\alpha_k}{2}\right) \|\nabla f(\mathbf{x}_k)\|^2. \tag{3.6}$$

The lower bound of the function decrement is positive if, and only if,

$$1 - \frac{L\alpha_k}{2} > 0 \Longleftrightarrow 0 < \alpha_k < \frac{2}{L}, \tag{3.7}$$

and is maximized for $\alpha = \frac{1}{L}$.

From (3.3), we have

$$|\lambda_{\max}(\nabla^2 J(\boldsymbol{\theta}))| \leq L, \tag{3.8}$$

and given the convexity of the cost functon $J(\boldsymbol{\theta})$, we get

$$\lambda_{\max}\left(\nabla^2 J(\boldsymbol{\theta})\right) \leq L \Longrightarrow \tag{3.9}$$

$$\lambda_{\max}\left(\frac{1}{n}\sum_{k=1}^{n} h_{\boldsymbol{\theta}}(\mathbf{x}_k)[1-h_{\boldsymbol{\theta}}(\mathbf{x}_k)]\mathbf{x}_k\mathbf{x}_k^T\right) + \lambda \leq L, \tag{3.10}$$

where $\lambda_{\max}$ is the maximum eigenvalue of the term in parentheses.

Thus, the optimal Lipschitz constant L_k can be computed as follows:

$$L_k = \lambda_{\max}\left(\frac{1}{n}\sum_{k=1}^{n} h_{\boldsymbol{\theta}}(\mathbf{x}_k)[1-h_{\boldsymbol{\theta}}(\mathbf{x}_k)]\mathbf{x}_k\mathbf{x}_k^T\right) + \lambda, \tag{3.11}$$

for each iterate k.

We derive the optimal step-size for each iteration of GD as follows:

$$\alpha_k = \frac{1}{L_k} = \frac{1}{\lambda_{\max}\left(\frac{1}{n}\sum_{k=1}^{n} h_{\boldsymbol{\theta}}(\mathbf{x}_k)[1-h_{\boldsymbol{\theta}}(\mathbf{x}_k)]\mathbf{x}_k\mathbf{x}_k^T\right) + \lambda}. \tag{3.12}$$

Summarizing, in order to get the optimal convergence rate for the GD algorithm, the step-size has to be the inverse of the Lipschitz constant of the function we try to minimize. The cost function in equation (2.18) is Lipschitz smooth and we use this property in order to get an optimal rate of convergence when we use GD for its minimization.

The step-size in equation (3.12) is used in other algorithms, as well, but the need to recompute its value in every iterate k deems it very inefficient for many applications. Thus, we seek to efficiently derive a less computationally intensive method to compute that step-size. In the next subsection, we examine how we can derive a simple upper bound for the Lipschitz constant L.

3.1.3 A Simple Upper Bound for the Lipschitz Constant

The data-set D remains unchanged for all iterations. Since $\mathbf{x}_k$ is a rather large vector, $\mathbf{x}_k\mathbf{x}_k^T$ is a rather large matrix to compute in each iteration. Moreover, calling $h_{\boldsymbol{\theta}}(\mathbf{x}_k)$ in order to compute $h_{\boldsymbol{\theta}}(\mathbf{x}_k)[1-h_{\boldsymbol{\theta}}(\mathbf{x}_k)]$ in each iteration, adds a large overhead. Thus, we observe that we make unnecessary, demanding, calculations to find the optimal L_k, only because the parameters of the hypothesis function (weight vector $\boldsymbol{\theta}$) change in every iteration.

We will use the formulation in equation (3.3) to derive a simple upper bound for the Lipschitz constant L of the cost functon $J(\boldsymbol{\theta})$. Using this value to compute the step-size we hope to get close to optimal convergence.

Since $0 \leq h_{\boldsymbol{\theta}}(\mathbf{x}_k) \leq 1$, we have that

$$0 \leq h_{\boldsymbol{\theta}}(\mathbf{x}_k)(1 - h_{\boldsymbol{\theta}}(\mathbf{x}_k)) \leq \frac{1}{4}. \tag{3.13}$$

Therefore we have that

$$0 \leq \nabla^2 J(\boldsymbol{\theta}_k) \leq \frac{1}{4n} \sum_{k=1}^{n} \mathbf{x}_k \mathbf{x}_k^T + \lambda \mathbf{I}. \tag{3.14}$$

Hence, we can reformulate equation (3.11) as

$$L_k = L = \lambda_{\max} \left(\frac{1}{4n} \sum_{k=1}^{n} \mathbf{x}_k \mathbf{x}_k^T \right) + \lambda \tag{3.15}$$

and the step-size can be reduced to

$$\alpha_k = \alpha = \frac{1}{L} = \frac{1}{\lambda_{\max}(\frac{1}{4n} \sum_{k=1}^{n} \mathbf{x}_k \mathbf{x}_k^T) + \lambda}. \tag{3.16}$$

Utilizing this upper bound for the Hessian matrix, we reduce the computations needed in each iteration, as we compute a simple upper bound for the constant L *only one* time before running any algorithm.

3.1.4 Accuracy and Terminating Condition

In order to set a terminating condition for the algorithm, we must first define accuracy.

With the term accuracy, or solution accuracy, we refer to the magnitude of the difference between the solution an algorithm converges to, as opposed to a predetermined solution.

Definition 3.1.3. We mathematically define accuracy at the k-th iteration as:

$$J(\boldsymbol{\theta}_k) - p_*, \tag{3.17}$$

where $\boldsymbol{\theta}_k$ is the optimal weight vector computed at the k-th iteration using any optimization algorithm and p_* is the solution to the problem.

To determine if an algorithm needs to be terminated, considering it has obtained a solution of certain accuracy, we need to define a terminating condition that takes accuracy into consideration. The target accuracy ϵ for any algorithm to attain for a problem, is set according to the application's particular requirements.

The terminating condition we used for GD is the following [12]:

$$\left\| \frac{\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1}}{\boldsymbol{\theta}_{k-1}} \right\| < \epsilon. \tag{3.18}$$

If the above condition is met, essentially, the algorithm has reached a point where the difference between two subsequent iterates of $\boldsymbol{\theta}$ is insignificant, since it is below our target accuracy ϵ. The extend in which the solution is accurate, or the solution error is small, is determined by the value of the term in (3.18).

In the algorithms that follow, unless another terminating condition is stated, the term in (3.18) *is used as the algorithm's terminating condition.*

3.1.5 Convergence Rate

It can be proved [11] that, the convergence rate of GD for a cost function with an L-smooth gradient is

$$J(\boldsymbol{\theta}_k) - J(\boldsymbol{\theta}^*) = \mathcal{O}\left(\frac{1}{k}\right). \tag{3.19}$$

A sufficient condition to attain accuracy $J(\boldsymbol{\theta}_k) - p_* \leq \epsilon$ is to perform

$$k^* = \mathcal{O}\left(\frac{1}{\epsilon}\right) \tag{3.20}$$

iterations.

This rate of convergnece is referred to as *sub-linear* convergence.

3.2 Gradient Descent With Backtracking Line Search

This algorithm is a variant of the Gradient Descent algorithm that uses a backtracking line search method to compute the step-size in each iteration.

3.2.1 Backtracking Line Search

Backtracking line search methods are used to grant a proper step-size for the descent step of an algorithm, for every iteration. Such methods are particularly helpful when there is no information available about the smoothness of a cost function or it is difficult to compute a smoothness constant [3]. When using backtracking schemes, we are guaranteed to make a descent step using the computed gradient [12].

Backtracking Condition

For this algorithm we use Armijo Backtracking Line Search [21], where the condition it seeks to satisfy is as follows:

$$J(\boldsymbol{\theta} - \eta_k \nabla J(\boldsymbol{\theta})) \leq J(\boldsymbol{\theta}) - c \cdot \eta_k \|\nabla J(\boldsymbol{\theta})\|^2, \tag{3.21}$$

where hyperparameter c is used to modify how strict the backtracking condition is and η_k is the step-size in iteration k.

We use a different notation for the step-size in this GD variant to distinguish it from the

step-size $\alpha = \frac{1}{L}$, which is used in non-backtracking algorithms and remains constant across iterations.

3.2.2 Algorithm

Algorithm 1 btGD Algorithm

repeat

  while $\left(J(\boldsymbol{\theta}_k - \eta_k \nabla J(\boldsymbol{\theta}_k)) > J(\boldsymbol{\theta}_k) - c \cdot \eta_k \|\nabla J(\boldsymbol{\theta}_k)\|^2\right)$ **do**

    $\eta_k = \beta \cdot \eta_k$

  end while

  $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta_k \nabla J(\boldsymbol{\theta}_k)$

until convergence

3.2.3 Parameters

Parameter c of the backtracking condition can be tuned to determine the aggressiveness of the backtracking condition. With smaller values, the backtracking loop iterates more times than with larger values, as in that case, the condition is easily met. Parameter c is recommended to take values in the range $(0, 0.5)$ for this algorithm [12].

Factor β determines how much the step-size is getting shrinked in every iteration. In a way, it determines the speed of the algorithm as smaller step-sizes tend to slow down convergence but are a good characteristic near the optimum, by means of accuracy. Parameter β in concordance with parameter c of the backtracking loop, can determine the speed and degree of descent for each iteration. Parameter β is recommended to receive values in the range $(0, 1)$ for this algorithm [12].

3.3 Nesterov Accelerated Gradient

Before we introduce Nesterov's algorithm, we must mention momentum-type gradient descent algorithms. The general idea is to add a momentum factor to the descent step in order to accelerate the descent itself, having confidence that the steps we choose to make are in the proper direction [2], [7], [19]. This addition has the goal to accelerate convergence, when comparing to GD, and to tackle any delays in the convergence caused by the slope of the cost function, mainly in ill-conditioned problems [22].

3.3.1 Descent Steps of Accelerated Algorithms

Informally speaking, these methods use a momentum factor that makes the algorithm take bigger descent steps in the opposite direction of the gradient of the cost function, accelerating convergence. Most momentum algorithms use an intermediate vector $\mathbf{v}$ that acts as the velocity of the current iterate, using an analogy of a ball going down a slope (iterates descending down a convex function) [7], [19], [22].

Simple descent steps for an algorithm of that type have the form

$$\mathbf{v}_{k+1} = \gamma \mathbf{v}_k - \alpha_k \nabla J(\boldsymbol{\theta}_k) \tag{3.22}$$

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \mathbf{v}_{k+1}, \tag{3.23}$$

where α_k is the step-size for iterate k and the *momentum factor* γ is a predetermined constant. Usually $\gamma < 1$, acting as a decay for the velocity vector, to mitigate for unnecessary large descent steps in later iterates [2], [19].

Equivalently, for the sake of simplicity, we describe those steps in a single descent step as follows [2]:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha_k \nabla J(\boldsymbol{\theta}_k) + \gamma(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1}), \tag{3.24}$$

where we easily observe that the descent step of equation (3.24) is that of GD augmented by the *acceleration term* $(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1})$ multiplied by the momentum factor γ. An algorithm that incorporates this descent step is referred to as the *Heavyball Method*, credited to Boris T. Polyak [5].

This formulation can further be used to describe a broader class of accelerated algorithms, with the term acceleration loosely used to describe the usage of the term $\gamma(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1})$ in their descent step.

3.3.2 Algorithm

Nesterov's Accelerated Gradient algorithm is a full gradient algorithm that incorporates a non-constant momentum factor [19]. It differs from the more basic momentum methods in the sense that it first makes a descent step and then computes the gradient vector to update weight vector $\boldsymbol{\theta}$, rather than the opposite [7].

Algorithm 2 NAG Algorithm

repeat

$t_{k+1} = \frac{1+\sqrt{4t_k^2+1}}{2}$

$\boldsymbol{\theta}_k = \tilde{\boldsymbol{\theta}}_k - \alpha \nabla J(\tilde{\boldsymbol{\theta}}_k)$

$\tilde{\boldsymbol{\theta}}_{k+1} = \boldsymbol{\theta}_k + \left(\frac{t_k - 1}{t_{k+1}}\right)(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1})$

until convergence

where $\alpha = \frac{1}{L}$ is the step-size also used in GD, t_k is the non-constant momentum factor and $\mathbf{y}$ is an intermediate vector implementing the acceleration.

3.3.3 Convergence Rate

It can be proved [2] that, the algorithm's convergence rate for a cost function with an L-smooth gradient is

$$J(\boldsymbol{\theta}_k) - J(\boldsymbol{\theta}^*) = \mathcal{O}\left(\frac{1}{k^2}\right), \tag{3.25}$$

and can be obtained in

$$k^* = \mathcal{O}\left(\frac{1}{\sqrt{\epsilon}}\right) \tag{3.26}$$

iterations.

It can be proved [2] that this is the optimal convergence rate a full gradient algorithm can attain.

3.4 Strong Convexity Variants

It can be quite computationally expensive to derive the Lipschitz smoothness constant L for the cost function's gradient when dealing with rather large data-sets, as we have already mentioned. The same is also true for the strong convexity constant μ when it is not easily available, as it needs to be computed before running each algorithm.

We have proven that the regularized form of our cost function is strongly convex, with the strong convexity constant being equal to the regularization constant, that is $\mu = \lambda$. Many full gradient algorithms use that property to further accelerate their convergence, enabling them to obtain linear rates of convergence [1].

We can compute the term

$$\frac{\sqrt{Q}-1}{\sqrt{Q}+1}, \tag{3.27}$$

where $Q = \frac{L}{\mu} = \frac{L}{\lambda}$ is the *condition number* of the problem, to use it as the momentum factor of the descent step of NAG, in order to accelerate its convergence when minimizing a strongly convex cost function [1], [19], [22].

The strong convexity constant μ has a constant value across iterations for our problem formulation, but the Lipschitz constant L can change across iterations. With the method of computing the upper bound of the optimal Lipschitz constant L, as we describe it in subsection 3.1.3, we can avoid such expensive computations. If we do not use this method, we not only need to compute L_k in every iteration, but we also need to recompute Q and the term in equation (3.27).

We now introduce the version of NAG that uses the above to obtain faster convergence, ommiting the diminishing momentum factor t_k and replacing it with the term of equation (3.27).

Algorithm 3 NAG Algorithm (Utilizing Strong Convexity)

repeat

$\boldsymbol{\theta}_k = \tilde{\boldsymbol{\theta}}_k - \alpha \nabla J(\tilde{\boldsymbol{\theta}}_k)$

$\tilde{\boldsymbol{\theta}}_{k+1} = \boldsymbol{\theta}_k + \left(\frac{\sqrt{Q}-1}{\sqrt{Q}+1}\right)(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1})$

until convergence

This variant is the one we use to gather experimental results in Chapter 5.

The GD algorithm showcases improved performance when optimizing a strongly convex cost function, without the need for any explicit changes in the algorithm.

3.4.1 Convergence Rates

Gradient Descent Utilizing Strong Convexity

The improved convergence rate of GD is

$$J(\boldsymbol{\theta}_k) - J(\boldsymbol{\theta}^*) = \mathcal{O}\left(\left(1 - \frac{1}{Q}\right)^k\right) \tag{3.28}$$

and can achieve ϵ-accuracy in

$$k^* = \mathcal{O}\left(Q \log\left(\frac{1}{\epsilon}\right)\right) \tag{3.29}$$

iterations [11] [17].

This convergence rate is *linear* and is considered better than the sub-linear rate GD can achieve for a non-strongly convex cost function.

Nesterov Accelerated Gradient Utilizing Strong Convexity

The improved convergence rate of NAG is

$$J(\boldsymbol{\theta}_k) - J(\boldsymbol{\theta}^*) = \mathcal{O}\left(\left(1 - \frac{1}{\sqrt{Q}}\right)^k\right), \tag{3.30}$$

and can achieve ϵ-accuracy in

$$k^* = \mathcal{O}\left(\sqrt{Q} \log\left(\frac{1}{\epsilon}\right)\right) \tag{3.31}$$

iterations [2].

This convergence rate is also *linear*.

Chapter 4

Stochastic Gradient Algorithms

In this chapter, we study stochastic optimization algorithms. At first, we introduce the differences between full gradient and stochastic gradient algorithms and how we approached particular methods. We form the basis of our study on stochastic methods by introducing Stochastic Gradient Descent and then we introduce variants that are based on it. We continue by studying backtracking line search variants and more advanced algorithms.

4.1 Preliminaries

4.1.1 Sampling

A typical implementation of the Stochastic Gradient Descent algorithm uses *one* random sample in each iteration. Other variants make use of *mini-batches*, which are essentially a fraction of samples from data-set D.

When referring to indexing or sampling, we mean that we get randomized indeces, thus random data samples and their corresponding labels.

Definition 4.1.1. The sampling routine we used is as follows:

- We pick random indeces $\text{mb}^k = \{i_1^k, i_2^k, \ldots, i_{\text{mb}}^k\}$ uniformly at random, from $\{1, \ldots, n\}$
- We select the data samples $\{\mathbf{x}_{i_1^k}, \mathbf{x}_{i_2^k}, \ldots, \mathbf{x}_{i_{\text{mb}}^k}\}$ from data-set D and their corresponding labels $\{y_{i_1^k}, y_{i_2^k}, \ldots, y_{i_{\text{mb}}^k}\}$

where mb refers to the size of the mini-batch and k is the current iterate.

Data sample replacement is not one of our concerns and it does not affect the performance of any algorithm.

4.1.2 Gradient Evaluations

One of the main reasons researchers shifted their interest towards stochastic algorithms is their less computationally expensive iterations. Stochastic algorithm iterations tend to be less expensive than their full gradient counterparts, due to the fact that they utilize less data in each iteration. This simplifies computations, so less time is required for certain computations, especially in rather large data-sets.

The downside is that, in order for stochastic algorithms to achieve the same accuracy as full gradient algorithms, they need to run for more iterations. Since these iterations are not

as computationally expensive, it is computationally feasible to run stochastic algorithms for many more iterations than full gradient algorithms.

The problem that arises when comparing the performance of stochastic and full gradient algorithms for a certain problem, is how to correctly calculate how many stochastic iterations are needed to make a full gradient evaluation. When referring to a full gradient evaluation, we mean the usage of the whole data-set to evaluate the cost function in an iteration. In this sense, one must correctly calculate what fraction of the full gradient a stochastic algorithm computes in each iteration, or equivalently, how many iterations are needed in order to compute a full gradient of size n.

With this in mind, a simple stochastic algorithm that only uses one sample of the data in each iteration must be run for n iterations to make a full gradient evaluation. When using mini-batches, the algorithm makes a fraction of a full gradient evaluation, namely $\frac{n}{\text{mb}}$, where mb is the size of the mini batch and n is the size of the whole data-set D.

4.1.3 Epoch

An *epoch* consists of n stochastic gradient evaluations or, in the case of mini-batching, $\frac{n}{\text{mb}}$ stochastic gradient evaluations. It is clear that, full gradient algorithms make one step in every epoch. We use this framework to solve the problem mentioned when comparing full gradient algorithms with stochastic gradient algorithms.

4.1.4 Stochastic Cost Function Form

We recall the regularized Logistic Regression cost function, in the full gradient setting:

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} J_i(\boldsymbol{\theta}) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|^2, \tag{4.1}$$

where each deterministic summand function $J_i(\boldsymbol{\theta})$ is associated with the i-th data pair $(\mathbf{x}_i, y_i)$, n refers to the number of data pairs in D and the term $\frac{\lambda}{2}\|\boldsymbol{\theta}\|^2$ refers to the regularization.

Definition 4.1.2. At the k-th iteration, the stochastic cost function is:

$$\begin{aligned} I_k(\boldsymbol{\theta}) :&= \frac{1}{\text{mb}}\sum_{i\in \text{mb}^k} J_i(\boldsymbol{\theta}) + \frac{\lambda}{2}\|\boldsymbol{\theta}\|^2 \\ &= \frac{1}{\text{mb}}\sum_{i\in \text{mb}^k} \left(-y_i\boldsymbol{\theta}^T\mathbf{x}_i + \log\left(1+e^{-\boldsymbol{\theta}^T\mathbf{x}_i}\right)\right) + \frac{1}{2}\lambda\boldsymbol{\theta}^T\boldsymbol{\theta}, \end{aligned} \tag{4.2}$$

where we sum each random realization of the cost function $J_i(\boldsymbol{\theta})$ up to the size of the mini-batch mb, and according to the selected random indeces mb^k. Lastly, we add the appropriate regularization term $\frac{1}{2}\lambda\boldsymbol{\theta}^T\boldsymbol{\theta} = \frac{\lambda}{2}\|\boldsymbol{\theta}\|^2$.

4.1.5 Unbiased Estimator

It can be proven [16] that, stochastic gradients are *unbiased estimators* of the full gradient, that is

$$\mathbb{E}[\nabla I_k(\boldsymbol{\theta})] = \nabla J(\boldsymbol{\theta}). \tag{4.3}$$

4.1.6 About Convergence Rates

We only provide convergence rates for the Stocahstic Gradient algorithm and its strongly convex counterpart. We study the convergence characteristics of the other algorithms in experiments.

4.2 Stochastic Gradient Descent

4.2.1 Algorithm

The algorithm's steps are similar to those of Gradient Descent, with the addition of the sampling routine, for the computation of the stochastic gradient vector in each iteration. Additionally, the formula for computing the step-size is different, incorporating a diminishing sequence [2].

Algorithm 4 SGD Algorithm

repeat

{ **Run Sampling Routine** }

$t_{k+1} = \frac{1}{\|\nabla I_k(\boldsymbol{\theta}_k)\|\sqrt{k+1}}$

$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - t_{k+1}\nabla I_k(\boldsymbol{\theta}_k)$

until convergence

4.2.2 Step-size

It has been proven [2] that, for a fixed step-size, SGD converges in a neighborhood of the optimal value but does not converge to the optimum. On the other hand, if the step-size is appropriately decreasing, then, in expectation, the sequence of cost function values converges to the optimum value. Thus, one can use a fixed step-size and try to converge to the optimum but after some iterations the noise in the stochastic gradients will prevent further progress.

To accommodate for this behavior, the step-size we choose for SGD has the following form [2]:

$$t_k = \frac{1}{\|\nabla I_k(\boldsymbol{\theta}_k)\|\,\sqrt{k}}. \tag{4.4}$$

4.2.3 Convergence Rate

The convergence rate of SGD is as follows:

$$\mathbb{E}[J(\boldsymbol{\theta}_k) - J(\boldsymbol{\theta}^*)] = \mathcal{O}\left(\frac{1}{\sqrt{k}}\right), \tag{4.5}$$

and accuracy ϵ can be obtained in

$$k^* = \mathcal{O}\left(\frac{1}{\epsilon^2}\right) \tag{4.6}$$

iterations.

The algorithm's convergence rate falls in the category of *sub-linear* convergence rates [2], [20], [17].

4.3 Stochastic Gradient Descent Utilizing Strong Convexity

As already mentioned, taking advantage of the property of strong convexity of a cost function can improve the performance of an algorithm. As in the full gradient setting, considering strong convexity in the design of a stochastic algorithm can positively affect its convergence rate.

4.3.1 Algorithm

The descent steps for this variant of SGD are the same as the original algorithm. The only step that changes is the update of the step-size, since this variant utilizes a different step-size update rule.

Algorithm 5 SGDSC Algorithm

repeat

{ **Run Sampling Routine** }

$t_{k+1} = \frac{2}{\mu(k+1)} = \frac{2}{\lambda(k+1)}$

$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - t_{k+1}\nabla I_k(\boldsymbol{\theta}_k)$

until convergence

4.3.2 Step-size

This variant's step-size also incorporates a diminishing sequence in order to reduce variance when near the optimum, in the same philosophy as its non-strongly convex counterpart. The norm of the gradient vector in the denominator of the term has been replaced with the strong convexity constant μ. To reiterate, in our problem the strong convexity constant is equal to the regularization constant, that is $\mu = \lambda$ [2], [20].

The step-size of SGDSC has the following form:

$$t_k = \frac{2}{\mu k} = \frac{2}{\lambda k}. \tag{4.7}$$

4.3.3 Convergence Rate

As expected, like in the full gradient setting, when utilizing the strong convexity property of the cost function, the convergence rate of the algorithm gets improved.

It can be proved [20] that, the convergence rate for the strongly convex variant of SGD is as follows:

$$\mathbb{E}[J(\boldsymbol{\theta}_k) - J(\boldsymbol{\theta}^*)] = \mathcal{O}\left(\frac{1}{k}\right), \tag{4.8}$$

and accuracy ϵ can be obtained in

$$k^* = \mathcal{O}\left(\frac{1}{\epsilon}\right) \tag{4.9}$$

iterations.

This is also a *sub-linear* convergence rate, but is faster than that of SGD. Since the function $\frac{1}{k}$ converges faster than $\frac{1}{\sqrt{k}}$ as $k \to \infty$, the strongly convex variant convergences faster than the original algorithm [2].

4.4 Stochastic Gradient Descent with Armijo Backtracking Line Search

This algorithm is a variant of Stochastic Gradient Descent. Instead of using a constant or decreasing step-size, it uses a backtracking line search routine to compute the step-size in each iteration.

The majority of the material for this algorithm is from [21].

4.4.1 Backtracking Condition

Armijo backtracking line search is mainly used in the deterministic setting [12], as we have seen it in the btGD algorithm in 3.2.1. It can be adapted to be used in the stochastic setting as well, where the condition it seeks to satisfy is as follows:

$$I_k(\boldsymbol{\theta}_k - \eta_k \nabla I_k(\boldsymbol{\theta}_k)) \leq I_k(\boldsymbol{\theta}_k) - c \cdot \eta_k \|\nabla I_k(\boldsymbol{\theta}_k)\|^2, \tag{4.10}$$

where $c < 1$ is a hyperparameter used to modify how strict the backtracking condition is and η_k is the step-size for iteration k.

The addition of the backtracking line search condition adds computational overhead to the algorithm, compared to SGD or SGDSC, in the form of reevaluating the cost function I_k, at least once and until the condition is met. It is important to note that these extra

computations are made using the mini-batch scheme and no full function evaluations are required.

4.4.2 Algorithm

Algorithm 6 SGD+Armijo Algorithm

repeat

{ **Run Sampling Routine** }

$\eta_k = \eta_{\max}$

repeat

$\eta_k = \beta \cdot \eta_k$

$\tilde{\boldsymbol{\theta}}_k = \boldsymbol{\theta}_k - \eta_k \nabla I_k(\boldsymbol{\theta}_k)$

until $I_k(\tilde{\boldsymbol{\theta}}_k) \leq I_k(\boldsymbol{\theta}_k) - c \cdot \eta_k \left\|\nabla I_k(\boldsymbol{\theta}_k)\right\|^2$

$\boldsymbol{\theta}_{k+1} = \tilde{\boldsymbol{\theta}}_k$

until convergence

4.4.3 Parameters

The initial step-size value for each iteration can be set using any method available but, by design, it is recommended to reset it to $\eta_{\max}$ in every iteration. A recommended maximum value for the step-size is $\eta_{\max} = 1$.

A recommended value for the step-size shrinkage factor is $\beta = 0.8$, for this particular algorithm.

A recommended value for the parameter of the aggressiveness of the backtracking condition in this particular algorithm is $c = 0.1$.

4.4.4 Step-size

Since this algorithm uses a backtracking line search method, the designers made some assumptions for the maximum value that the step-size can take. In this way, the step-size is constrained in a range properly set, so that the backtracking condition in (4.10) is satisfied.

Having said that, the step-size η_k is constrained to lie in the range $(0, \eta_{\max}]$ and satisfies the following inequality:

$$\eta_k \geq \min\left\{\frac{2(1-c)}{L_k}, \eta_{\max}\right\}, \tag{4.11}$$

with L_k being the Lipschitz constant of the stochastic gradient for iteration k.

When information about smoothness is available, the initial step-size can be set as the usual $\alpha = \frac{1}{L}$. In this case it is not, so with a sufficiently large $\eta_{\max}$ value and $c \leq 0.5$, the step-size appears to be at least as large as $\frac{1}{L}$.

4.5 Stochastic Nesterov Accelerated Gradient with Armijo Backtracking Line Search

In the same philosophy as we transitioned from GD with backtracking line search to its counterpart in the stochastic setting, we continue with the study of a NAG-type stochastic algorithm with Armijo backtracking line search. This is the corresponding stochastic variant of NAG we studied in the full gradient setting, inheriting all of its benefits, while extending its flexibility with the utilization of a backtracking line search scheme.

The majority of the material in this section is also from[21].

4.5.1 Algorithm

Algorithm 7 Nesterov+Armijo Algorithm

repeat

{ **Run Sampling Routine** }

$\eta_k = \eta_{\max}$

repeat

$\eta_k = \beta \cdot \eta_k$

$\tilde{\boldsymbol{\theta}}_k = \boldsymbol{\theta}_k - \eta_k \nabla I_k(\boldsymbol{\theta}_k)$

until $I_k(\tilde{\boldsymbol{\theta}}_k) \leq I_k(\boldsymbol{\theta}_k) - c \cdot \eta_k \left\|\nabla I_k(\boldsymbol{\theta}_k)\right\|^2$

$\boldsymbol{\theta}_{k+1} = (1-\tau)\tilde{\boldsymbol{\theta}}_k + \tau \boldsymbol{\theta}_k$

$t_{k+1} = \frac{\left(1+\sqrt{1+4t_k^2}\right)}{2}$

$\tau = \frac{1-t_k}{t_{k+1}}$

until convergence

The descent steps of the algorithm can be simplified in the same way as in the full gradient Nesterov algorithm, using the momentum term upper bound and by combining the terms in the descent steps, without compromising the computed terms needed for the evaluation of the backtracking condition.

4.5.2 Parameters

For this algorithm, the step-size update rule and initial value tuning, follow the same rules as in the case of SGD+Armijo. To reiterate, the initial value for the step-size is recommended to be $\eta_{\max} = 1$ and it is advised to be reset to that value in each iteration.

The shrinkage parameter of the step-size in the backtracking loop is recommended to be set as $\beta = 0.8$, like in SGD+Armijo.

The aggressiveness parameter of the backtracking condition is recommended to be set to a larger value than SGD+Armijo, in order for the backtracking loop to not have such a strong effect on the shrinkage of the step-size. What this means is that, we tend to

get faster convergence but accuracy near the optimum might suffer. This parameter is recommended to be set as $c = 0.5$.

4.6 Random Adaptive Coordinate Descent Method

This algorithm is inspired by a random coordinate descent method of Nesterov. It introduces a different backtracking scheme than the one we have studied thus far in the thesis. Albeit the algorithm's simplicity, it obtains good performance, comparable to that of the other backtracking algorithms we studied, as observed in experiments.

The majority of the material is from [14], which introduces the line-search method.

4.6.1 Algorithm

Our slightly modified version of the algorithm, is as follows:

Algorithm 8 RACDM Algorithm

repeat

{ **Run Sampling Routine** }

while $\left(\nabla I_k(\boldsymbol{\theta}_k) \cdot \nabla I_k(\tilde{\boldsymbol{\theta}}) < 0\right)$ **do**

$\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta}_k - \eta_k \cdot \nabla I_k(\boldsymbol{\theta}_k)$

$\eta_k = \frac{1}{2} \cdot \eta_k$

end while

$\boldsymbol{\theta}_{k+1} = \tilde{\boldsymbol{\theta}}$

until convergence

4.6.2 Backtracking Method

The backtracking method of RACDM, in every iteration k, searches for a gradient $\nabla I_k(\tilde{\boldsymbol{\theta}})$ that has an acute angle with $\nabla I_k(\boldsymbol{\theta}_k)$. It does so, in order to make a descent step in the correct descent direction. Intuitively, it searches for a gradient and a step-size that do not make the algorithm take a bigger descent step than needed and diverge to a point further from the optimal point $\boldsymbol{\theta}^*$.

This backtracking method provides the algorithm with the same advantageous characteristics such as other backtracking methods.

4.7 Adaptive Moment Estimation Algorithm

This algorithm is designed to work with stochastic cost functions and is based on adaptive estimates of their low-order moments to formulate its descent steps. Due to the fact that it can solve problems with unknown cost function properties and that it can handle problems with very large data-sets, it has been a very popular choice of optimizer for many machine learning problems. It requires minimal hyperparameter tuning, in contrast to the back-tracking methods mentioned above, and even with the proposed default values provides good performance even for a problem such as regularized Logistic Regression.

The majority of analysis is based on [10], which introduces the algorithm.

4.7.1 Algorithm

Algorithm 9 ADAM Algorithm

{ **Run Sampling Routine** }

$\mathbf{m}_0 = \mathbf{0}$

$\mathbf{v}_0 = \mathbf{0}$

repeat $t = 1, 2, \ldots$

$\mathbf{m}_t = \beta_1 \cdot \mathbf{m}_{t-1} + (1 - \beta_1) \cdot \nabla I_k(\boldsymbol{\theta}_{t-1})$

$\mathbf{v}_t = \beta_2 \cdot \mathbf{v}_{t-1} + (1 - \beta_2) \cdot (\nabla I_k(\boldsymbol{\theta}_{t-1}) .* \nabla I_k(\boldsymbol{\theta}_{t-1}))$

$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}$

$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}$

$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \alpha \cdot \left(\hat{\mathbf{m}}_t \,./\, (\sqrt{\hat{\mathbf{v}}_t + \epsilon_{ADAM}})\right)$

until convergence

4.7.2 Moment Estimates

The first moment estimate $\mathbf{m}$ refers to the mean. When we refer to the mean, we refer to the mean of a distribution. When optimizing with ADAM, or any other algorithm, the data distribution may not be available. Thus, we consider the distribution of the data to be the collection of all stochastic gradients at each timestep t of the optimization process. It is obvious that, we do not know the total distribution until the optimization process is done. Hence, without loss of generality, we consider the mean to be zero asymptotically, with the mean actually flactuating as it is being computed at each timestep t. Intuitively, this means that we do not value positive gradients to be more probable than negative gradients, and vice versa.

Considering the above, it seems unclear how we can compute the raw second moment estimate $\mathbf{v}$ using the mean. From the definition of the variance, we can obtain it as the expectation of the stochastic gradient at timestep t squared, since the mean equals to zero. From the formulation of stochastic algorithms, we observe that this is actually the stochastic gradient squared. Thus, the variance is nothing but the squared stochastic

gradient at timestep t and is computed without explicit knowledge of any underlying probability distribution.

4.7.3 Hyperparameters

Other than the step-size, the hyperparameters of ADAM consist of the *exponential decay rates* for the moment estimates and a constant that helps avoid division by zero. All of the default values discussed are said to be tested with machine learning problems and provide good results. Thus, we use them for solving regularized LR, as well.

The exponential decay rates that are responsible for the scaling of the moment estimates are β_1 for the mean and β_2 for the uncentered variance. The default values provided for these hyperparameters are $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Since β_1 and β_2 are in the range $(0, 1)$, the moment estimates are computed as convex combinations. During the estimation process, the estimates generate a bias, which is corrected after their updates with $\hat{\mathbf{m}}_t$ and $\hat{\mathbf{v}}_t$, respectively.

These hyperparameters play the role of decay rates for the minimization of the moment estimation terms and are parameterized by the timestep t. This is the reason that variance is introduced in later iterates in the sequence of cost function values produced by the algorithm, since the moment estimates obtain very small values and fail to influence the weight vector update (thus the cost function values), as much as in earlier stages of convergence.

Hyperparameter ϵ_{ADAM} is used as a means to avoid division by zero when the variance estimate converges to zero. The default proposed value is $\epsilon_{ADAM} = 10^{-8}$.

4.7.4 Step-size

The constant step-size α for ADAM essentially serves the same purpose as in the other algorithms we have studied, but its actual value has a more important meaning.

If we assume that the hyperparameter $\epsilon_{ADAM} = 0$ and the estimate of the second moment $\mathbf{v}$ is not zero, then the effective descent step for the algorithm is

$$\Delta_t = \alpha \cdot \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t}}. \tag{4.12}$$

Thus, the step-size has the following bounds:

$$|\Delta_t| \leq \alpha \cdot \frac{1-\beta_1}{\sqrt{1-\beta_2}}, \quad \text{when } (1-\beta_1) > \sqrt{1-\beta_2}, \tag{4.13}$$

$$|\Delta_t| \leq \alpha, \quad \text{when } (1-\beta_1) < \sqrt{1-\beta_2}, \tag{4.14}$$

where the first case appears only when the gradient has been zero in all timesteps except the current timestep, meaning cases that large sparsity exists in the gradient iterates. For less sparse cases, the step-size, thus the descent step, will be smaller.

Lastly, we have that

$$|\Delta_t| < \alpha, \quad \text{when } (1-\beta_1) = \sqrt{1-\beta_2}, \tag{4.15}$$

since $|\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t}}| < 1$.

In most cases, we have that $|\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t}}| \approx \pm 1$, since $|\frac{\mathbb{E}[\nabla I_k(\boldsymbol{\theta})]}{\sqrt{\mathbb{E}[\nabla I_k(\boldsymbol{\theta}) .* \nabla I_k(\boldsymbol{\theta})]}}| \leq 1$, hence, without loss of generality, we conclude that the effective descent step of the algorithm is bounded by the step-size as follows:

$$|\Delta_t| \lessapprox \alpha. \tag{4.16}$$

Intuitevely, the above bound offers a trust region for the current weight vector (parameter) values, in which the gradient estimate provides sufficient and useful information in order to effectively descent further. In that way, the step-size itself sets an upper bound for the magnitude of the number of descent steps in which we can reach the optimal point $\boldsymbol{\theta}^*$, from its initial values.

Therefore, it is obvious that, the proper selection of the step-size is crucial for the performance of the algorithm and offers insight for its convergence characteristics, prior to running it, more easily than the other algorithms studied. The proposed default value for the step-size is $\alpha = 0.001$.

Chapter 5

Experiments

In this chapter, we study the behavior of the algorithms presented in this thesis. As already mentioned, we are interested in the performance of the algorithms in different conditions, as well as their convergence characteristics.

Preliminaries

The size of the data-set was set to be $n \times N$, where $n = 1000$ is the number of data pairs in data-set D, and $N = 20$ their dimensions.

The initial values of the weight vector $\boldsymbol{\theta}_0$ are set as zeros, for all algorithms.

Regularization constant λ was set as $\lambda = 0.01$, as larger values give faster convergence, but with lower accuracy, and smaller values give slower convergent algorithms with better accuracy.

The target solution accuracy ϵ for our experiments is $\epsilon = 10^{-6}$, being a realistic target for the algorithms and data-set tested.

The optimal point p_* for each realization of the problem is computed by using the CVX convex optimization tool.

We keep the number of iterations performed by the Gradient Descent algorithm as a benchmark, to bound the maximum number of iterations all the algorithms will perform, in order to eliminate favoring during their performance evaluation.

Lastly, in order to make convergence plots more readable, plot averaging was used, meaning that the plot data provided were an average over 5 different realizations of the problem.

We keep all the above values constant across all experiments.

5.1 Data Types Considered

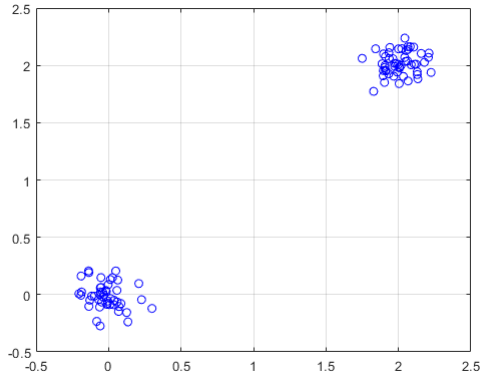
In this section, we visualize two simple cases of the problem in $\mathbb{R}^2$, for separable and non-separable data, as an introduction to the two categories of data that we considered during experimentation.

5.1.1 Separable Data

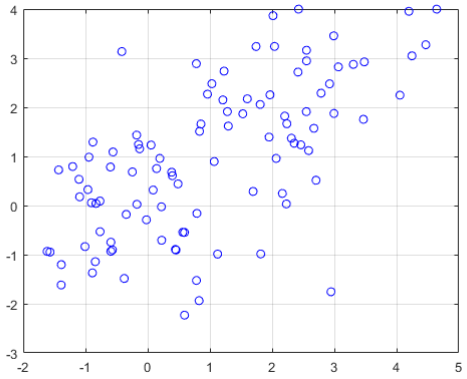
To further clarify the term (linearly) separable data, we speak of data that can be separated by a line on a two-dimensional plane. When studying problems of higher dimensions, that separating line has the form of a separating hyperplane. Such data favor the performance of linear classifiers such as LR, since they can easily infer on the hyperplane that can distinguish the underlying data classes.

5.1.2 Non-Separable Data

What differentiates separable from non-separable data is the existance of a hyperplane that can distinguish their classes. Non-separability can be an intrinsic characteristic of a data-set or the result of the existance of noise in the data. In this case, it is impossible to linearly separate the data, so misclassifications occur when data labels are inferred using a linear classifier such as LR.



A hundred separable data points ($n = 100$) on $\mathbb{R}^2$.



A hundred non-separable data points ($n = 100$) on $\mathbb{R}^2$.

5.2 Data Creation

In this section, we formulate the method used for creating our pseudorandom data-sets for each category of data considered during experimentation. To create our data-set D in each experiment, we use normally distributed pseudorandom numbers. Data and label pairs $(\mathbf{x}_i, y_i)$ are designed in a way that favors implementation, so we can make the data non-separable, from separable, with the simple addition of more noise.

We choose a vector

$$\boldsymbol{\mu} = \delta \cdot \mathbf{1}_{1\times N}, \tag{5.1}$$

where δ is an arbitrarily chosen number and N is the dimension of the problem. This vector will be *vertical* to the separating hyperplane of our binary classified data.

We create n binary labels using pseudorandom numbers from the normal distribution and the sign function as:

$$\mathbf{y} = sign(\mathcal{N}_{n\times 1}), \tag{5.2}$$

with $\mathbf{y} = \{y_1, y_2, \ldots, y_n\}$.

With that formulation, we get labels that take the values -1 and 1. With those labels, we must use a differently formulated cost function.

In order for our labels to take the values 0 and 1, which are the labels our cost function is designed for, we reformulate the label vector as:

$$\mathbf{y} = \frac{sign(\mathcal{N}_{n\times 1}) + 1}{2}. \tag{5.3}$$

Using that formulation for $\mathbf{y}$, we get labels that are compatible with our cost function.

Data-set D is created using the random labels $\mathbf{y}$, multiplied by vector $\boldsymbol{\mu}$, with the addition of random noise produced by a normal distribution as:

$$\mathbf{x}_i = y_i \cdot \boldsymbol{\mu} + \epsilon_{noise}, \tag{5.4}$$

where $\epsilon_{noise} \sim \mathcal{N}(0, \sigma^2\mathbf{I})$, with σ^2 being the noise variance.

Choosing a small value for σ^2, the data remain separable, as we can observe in subsection 5.1.1 for an example set in $\mathbb{R}^2$. Choosing a larger value makes the data non-separable, as with the existance of significant noise, the data points no longer take values close to 0 or δ, as we can observe in subsection 5.1.2, for $\delta = 2$.

5.3 Experiments with Separable Data for $\lambda = 0.01$.

5.3.1 Convergence Rates of Full Gradient Algorithms for Separable Data

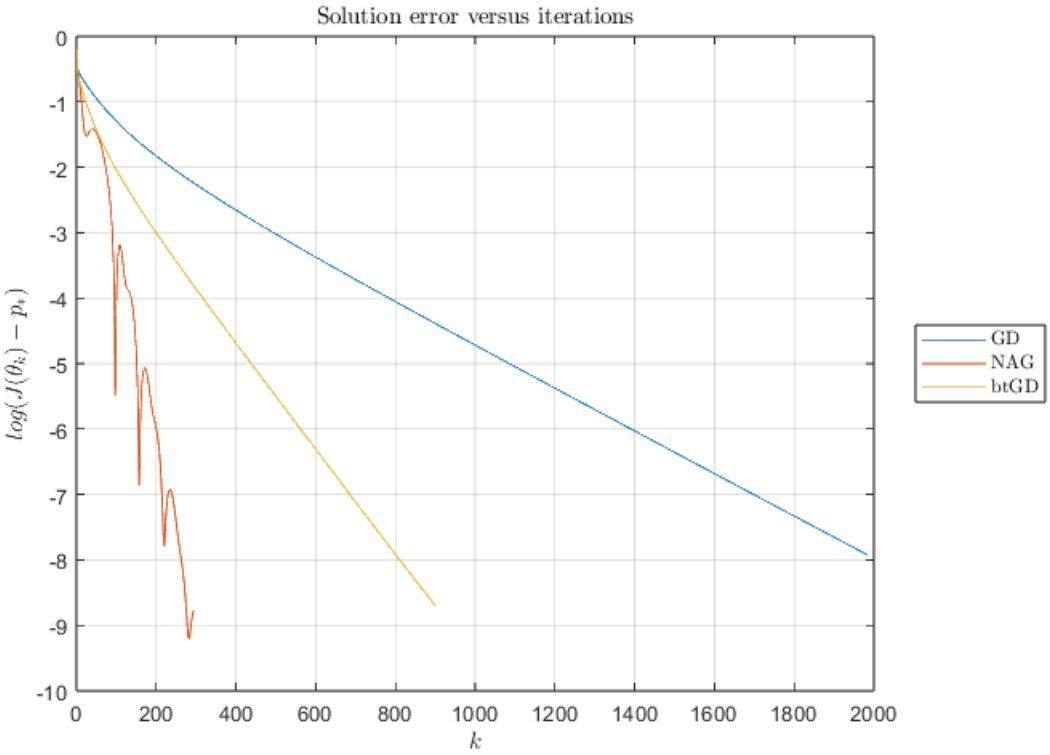


Figure 5.2: Full gradient algorithms convergence graphs for separable data.

Summary

From all the full gradient algorithms tested, we observe that NAG obtains the faster convergence, in this and subsequent experiments. More accurate results are obtained by btGD and GD, but in considerably more epochs, with GD being the slowest in our experiments.

5.3.2 Convergence Rates of Stochastic Algorithms for Separable Data and Different Mini-batch Sizes

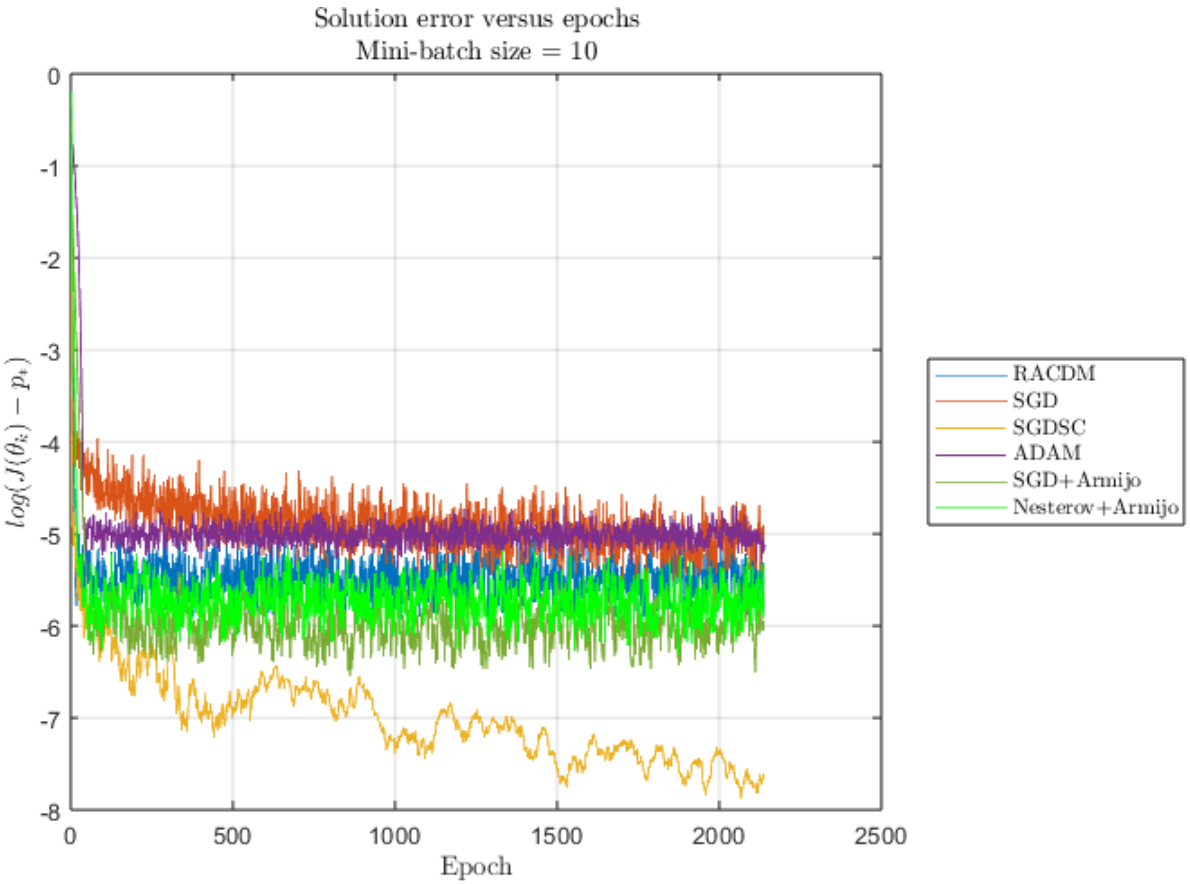


Figure 5.3: Stochastic gradient algorithms convergence graphs for separable data and mini-batch size = 10.

Summary

For this case, we observe that SGDSC has the best performance, both in terms of speed and accuracy. SGD has the worst performance, in terms of convergence speed and accuracy, compared to other algorithms. All the remaining algorithms in this graph seem to obtain similar convergence characteristics, which can be more easily observed in the next figure.

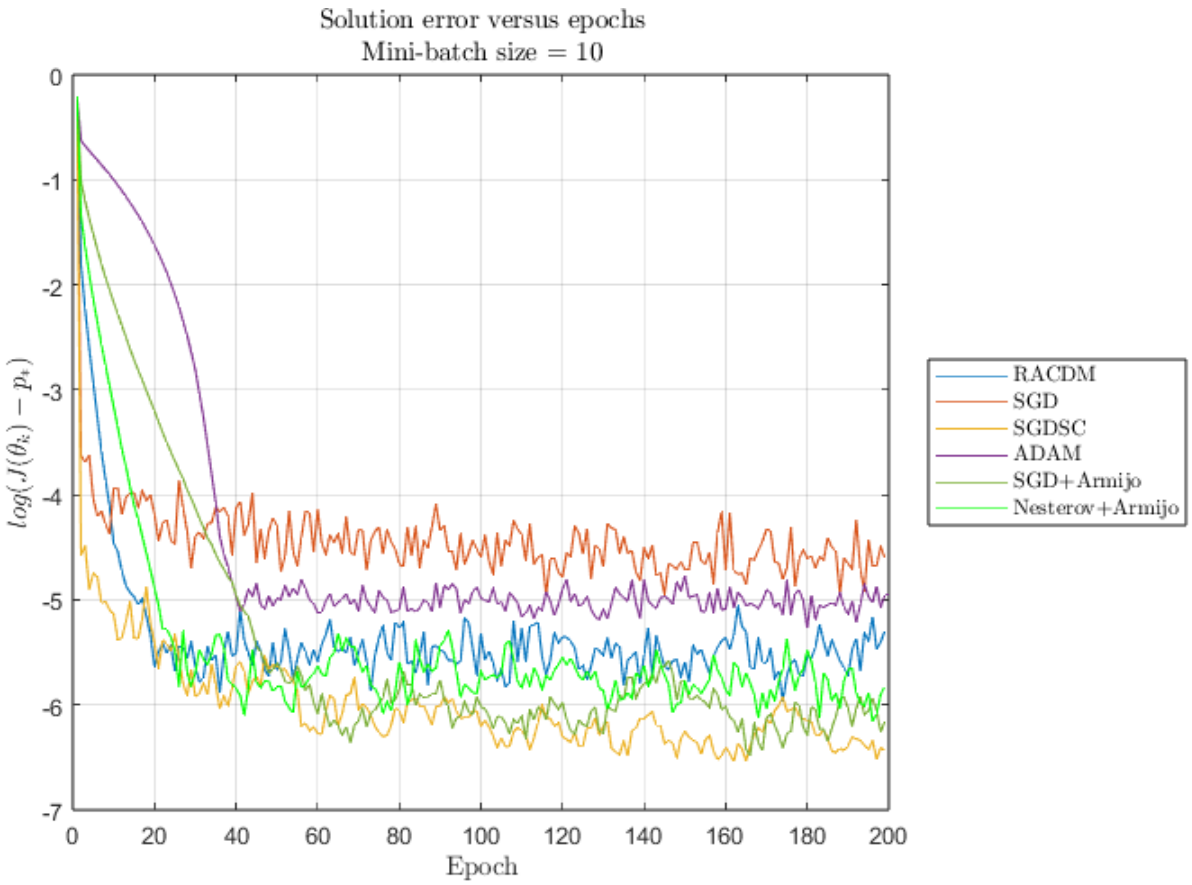


Figure 5.4: Stochastic gradient algorithms convergence graphs for separable data and mini-batch size = 10, bounded to 200 epochs.

Summary

In this graph, we focus on the first 200 epochs of the convergence of the algorithms. In this case, the above remarks still hold true, but we can observe that from all the algorithms that showcased similar performance, SGD+Armijo performance is comparable to SGDSC, with the two being the most accurate. In these earlier stages of convergence, SGD remains the least accurate and ADAM showcases slower asymptotic convergence than the other algorithms.

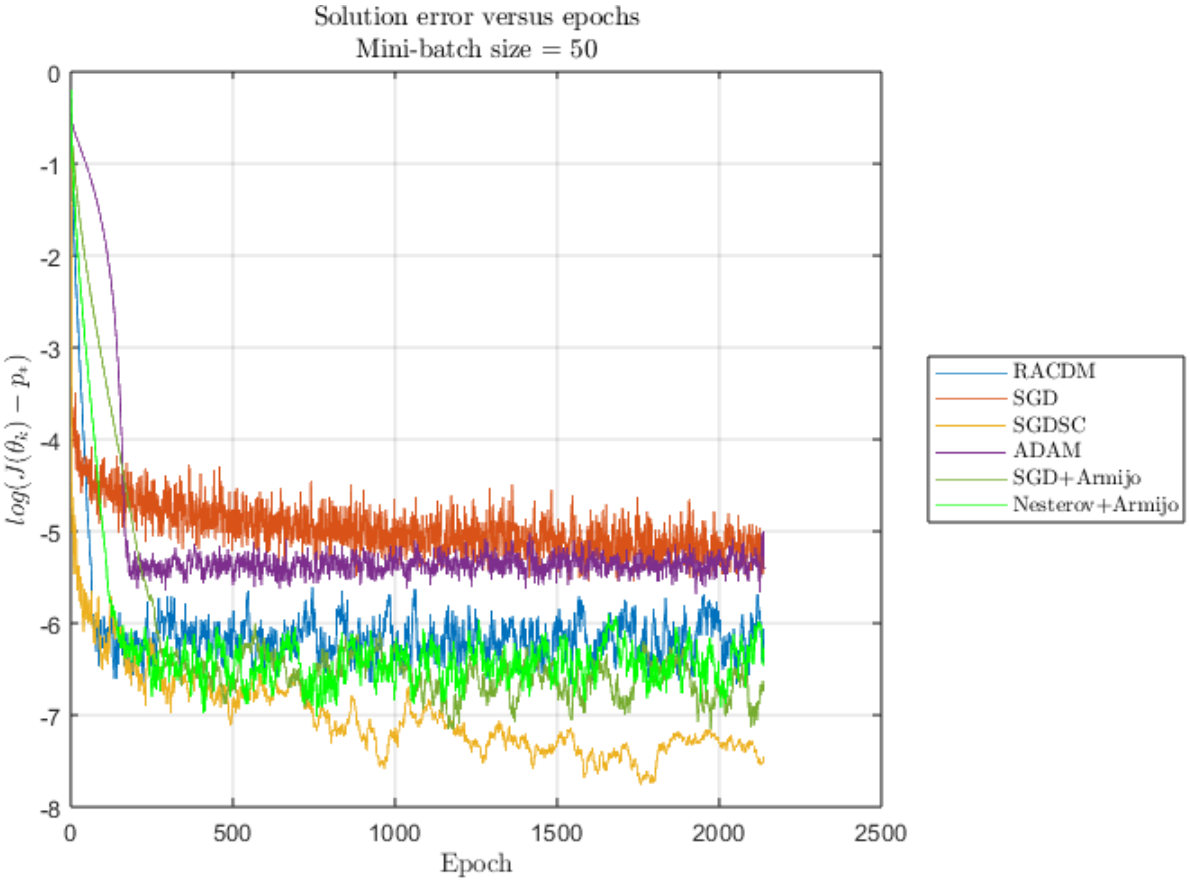


Figure 5.5: Stochastic gradient algorithms convergence graphs for separable data and mini-batch size = 50.

Summary

From this experiment, with a larger mini-batch size, we can observe that previous observations do not hold true in their entirety. Although, the algorithm with the best convergence is again the SGDSC algorithm, all the other algorithms have different behavior than with the smaller mini-batch size. SGD is observed to be the least performant algorithm in this case, too. RACDM, and the Armijo backtracking variants obtain relatively the same accuracy, but RACDM is the fastest in earlier stages of convergence. ADAM is the slowest of the algorithms in terms of asymptotic convergence, but obtains comparable accuracy to the other algorithms in later epochs.

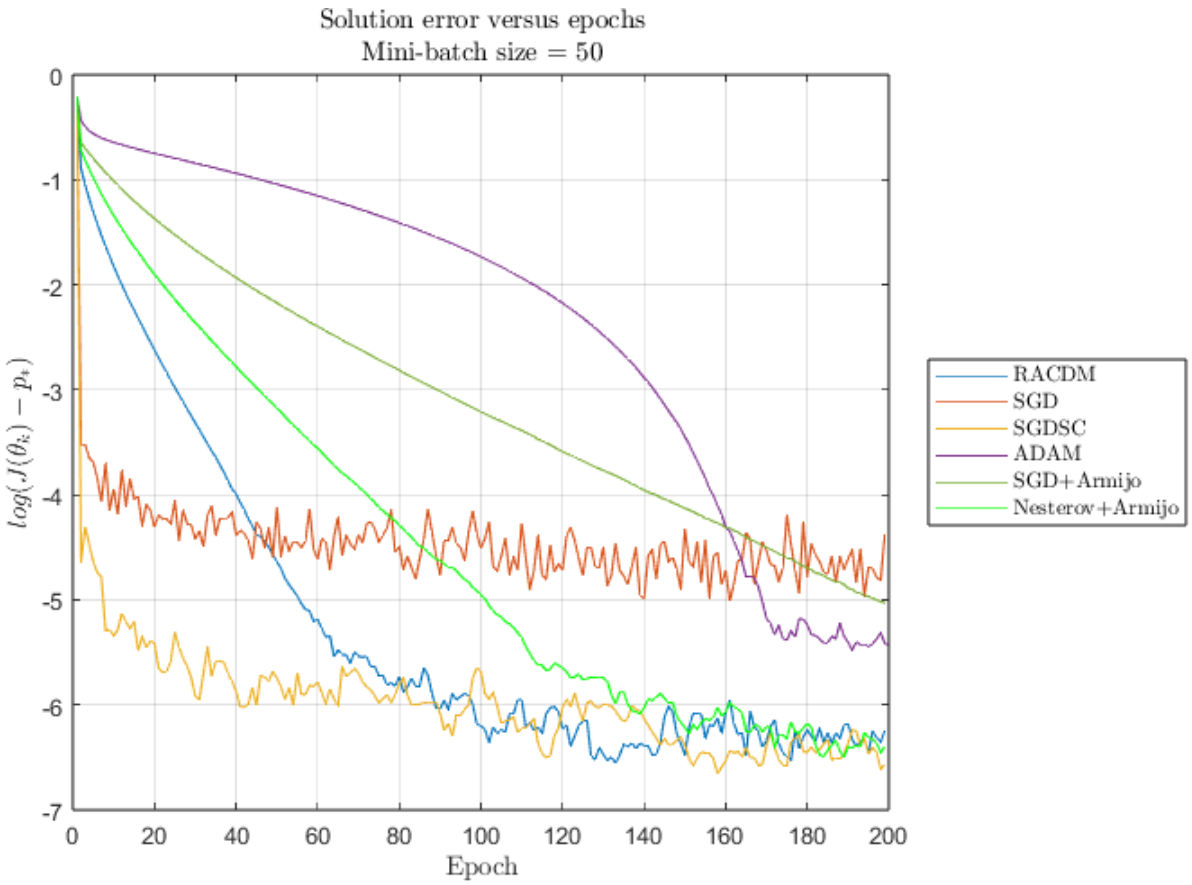


Figure 5.6: Stochastic gradient algorithms convergence graphs for separable data and mini-batch size = 50, bounded to 200 epochs.

Summary

In this graph, previous observations about the earlier stage of convergence are more prominent. We additionally observe, that, SGD albeit being the worst in terms of accuracy, achieves asymptotic convergence very fast and can be directly compared to the speed of SGDSC, in that stage of convergence. Apart from ADAM, the other algorithms demonstrate similar convergence characteristics, with RACDM being the fastest, as mentioned. This puts RACDM to an advantage over ADAM and the Armijo variants when run for 200 epochs, since it obtains comparable accuracy to that of SGDSC in that case.

5.3.3 Comparison Between Full and Stochastic Gradient Algorithms for Separable Data

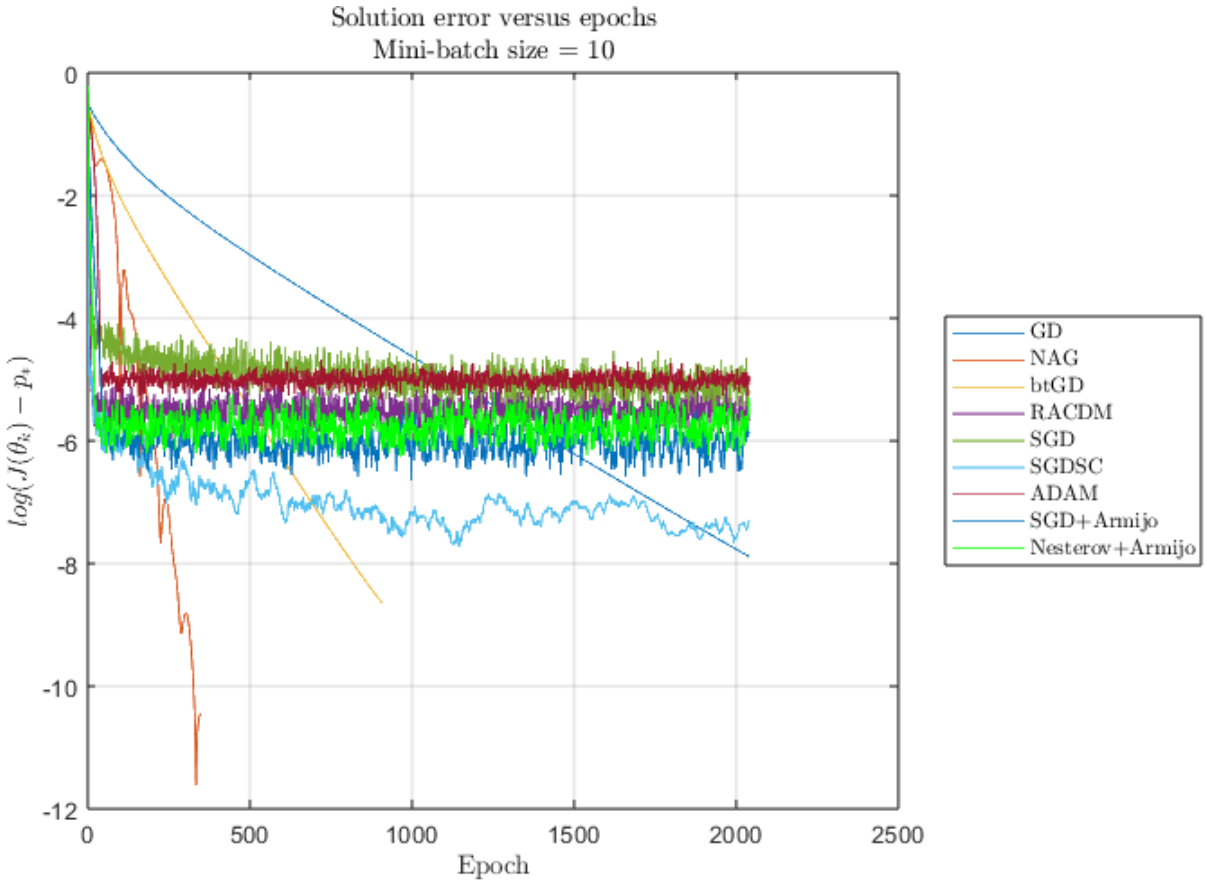


Figure 5.7: Comparison of convergence graphs of all algorithms studied for separable data and mini-batch size = 10.

Summary

Comparing full gradient algorithms with stochastic gradient algorithms in the same number of epochs, we observe that stochastic algorithms converge faster, but full gradient algorithms are more accurate. Only SGDSC obtains comparable accuracy with full gradient algorithms, except NAG. NAG achieves the best accuracy with speed comparable to that of stochastic algorithms. Although, it is not a fair comparison for the other stochastic algorithms since we utilize strong convexity in SGDSC and the full gradient variants.

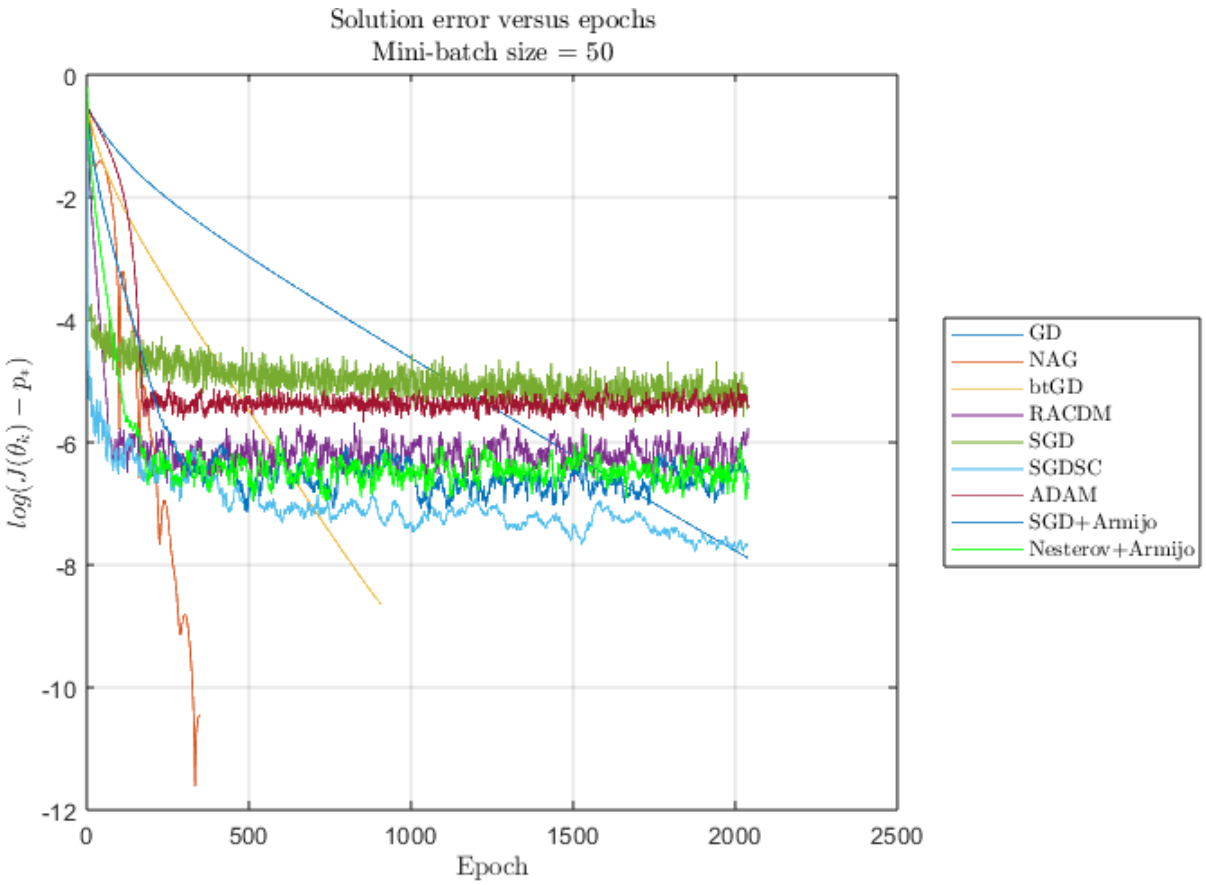


Figure 5.8: Comparison of convergence graphs of all algorithms studied for separable data and mini-batch size = 50.

Summary

Utilizing a bigger mini-batch size, the accuracy of all stochastic algorithms is comparable to that of full gradient algorithms. In addition, stochastic algorithms remain faster in this case, although, most are observed to attain convergence speed close to that of NAG in the earlier stages of convergence. Most stochastic algorithms are observed to obtain accuracy close to that of the other full gradient algorithms in later epochs. The most accurate algorithm remains NAG and obtains asymptotic convergence with comparable speed to stochastic algorithms in this case, too.

5.4 Experiments with Non-Separable Data for $\lambda = 0.01$.

5.4.1 Convergence Rates of Full Gradient Algorithms for Non-separable Data

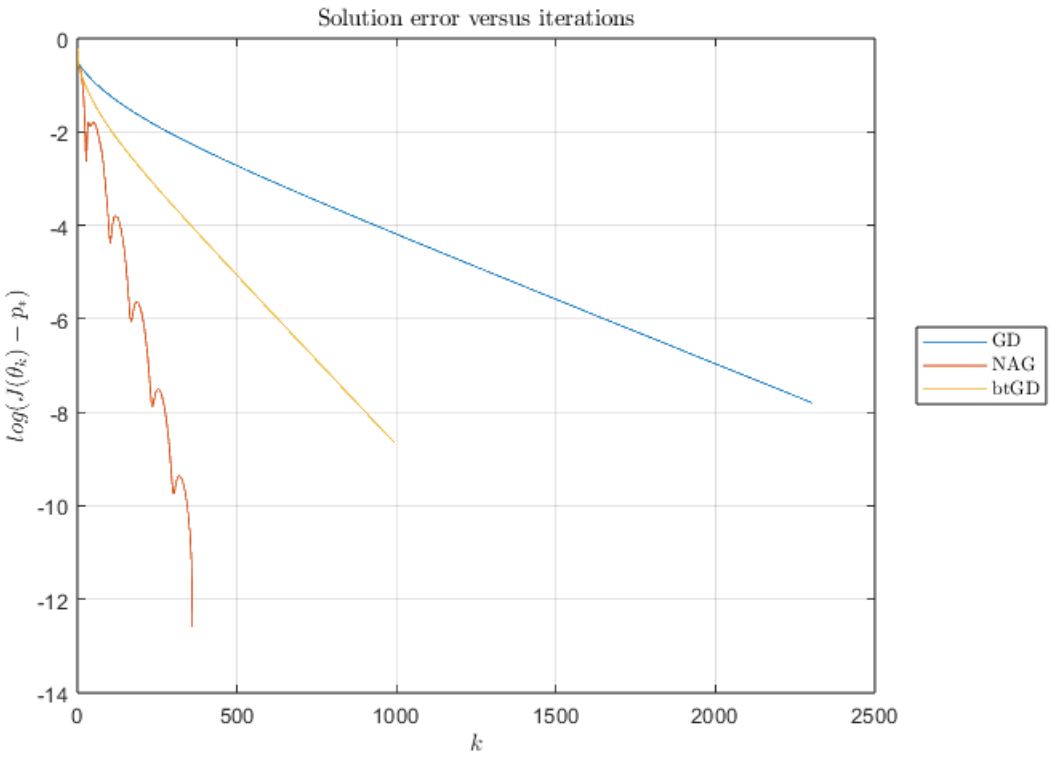


Figure 5.9: Full gradient algorithms convergence graphs for non-separable data.

Summary

In the case of non-separable data, full gradient algorithms present the same convergence characteristics, but obtain slightly less accurate results than the separable data case, and achieve those results in a larger number of epochs. The most performant algorithm is NAG, which obtains even more accurate results in this case. GD and btGD present the same convergence characteristics, as in the case of separable data.

5.4.2 Convergence Rates of Stochastic Algorithms for Non-separable Data and Different Mini-batch Sizes

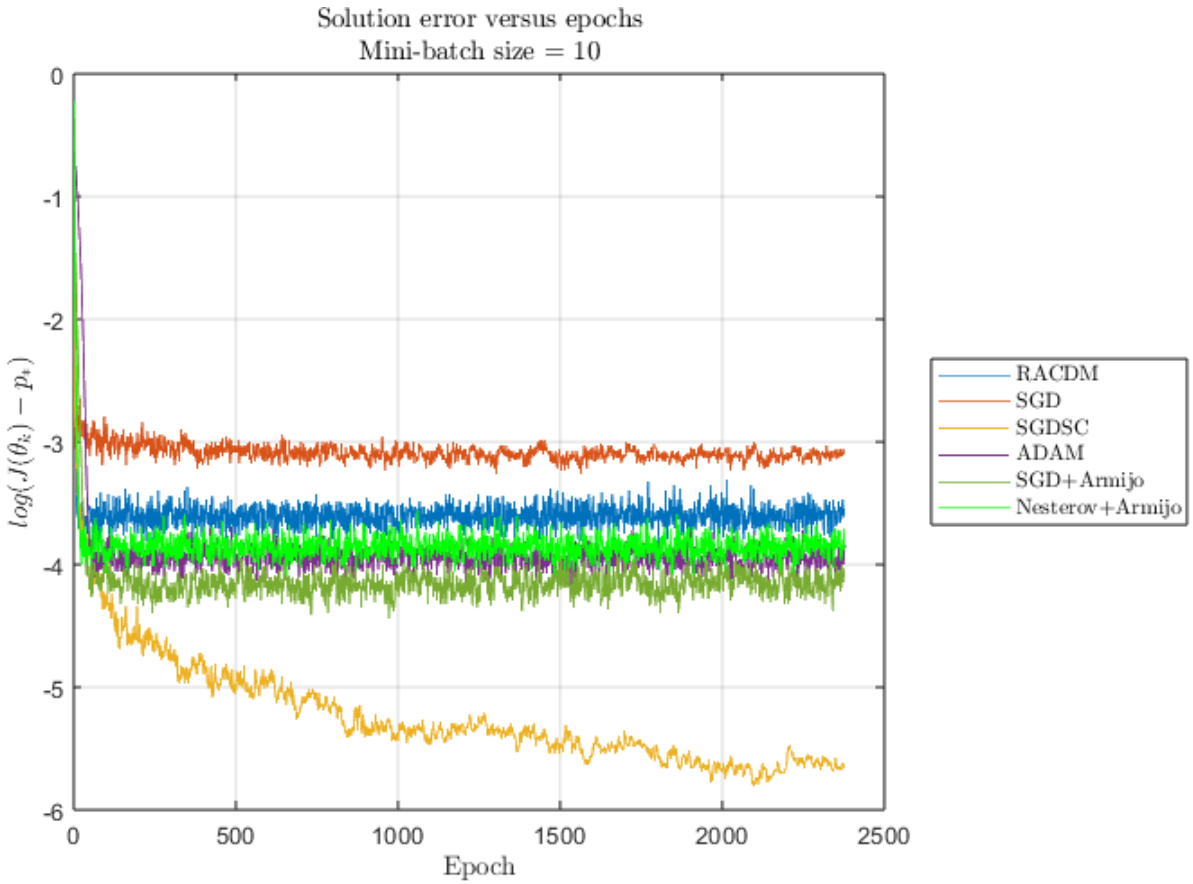


Figure 5.10: Stochastic gradient algorithms convergence graphs for non-separable data and mini-batch size = 10.

Summary

In this experiment, we observe some similar results with its separable data counterpart, but the differences in accuracy between the algorithms are prominent. The best accuracy is obtained by SGDSC and the worst from SGD. All the other algorithms have comparable performance in all stages of convergence, with SGD+Armijo showcasing the next best accuracy.

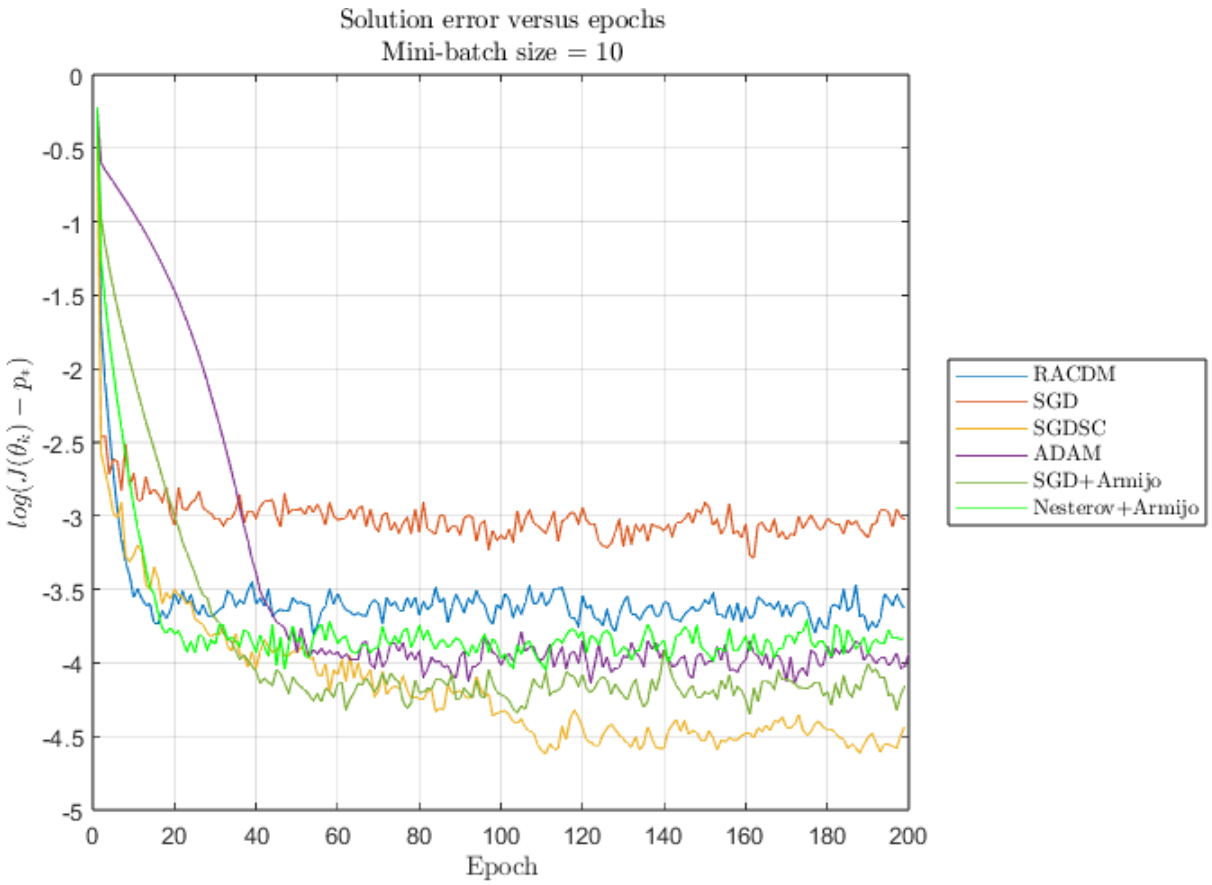


Figure 5.11: Stochastic gradient algorithms convergence graphs for non-separable data and mini-batch size = 10, bounded to 200 epochs.

Summary

Focusing on less epochs, we observe that the similarities between the algorithms' convergence characteristics previously mentioned are clearer in this plot. In this case too, most algorithms have the same convergence speed in earlier epochs, with ADAM being the slowest. The accuracy of SGDSC and SGD+Armijo in earlier epochs is directly comparable, with SGD+Armijo being slower.

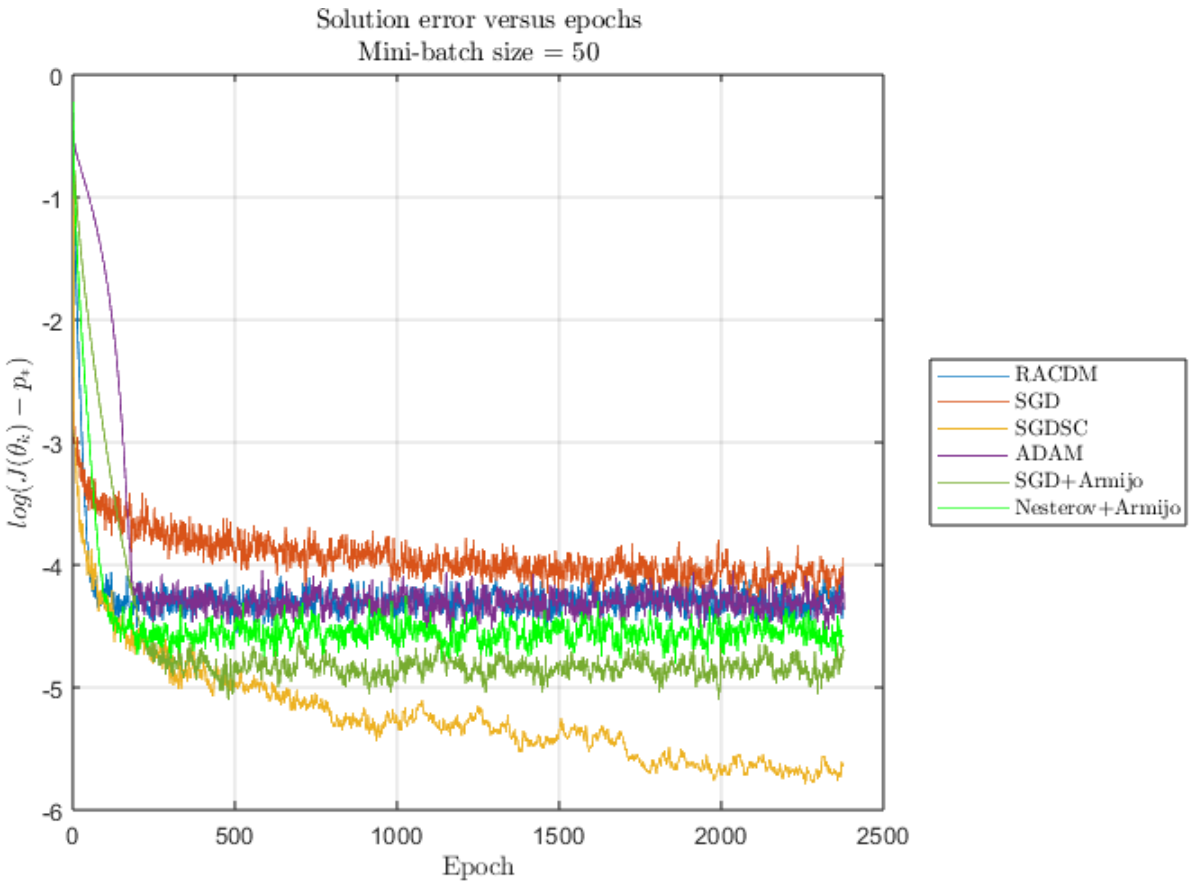


Figure 5.12: Stochastic gradient algorithms convergence graphs for non-separable data and mini-batch size = 50.

Summary

In this case, it is important to focus on the improvement in accuracy for all algorithms with the bigger mini-batch size, which is obtained in a similar number of epochs. SGD remains the least performant algorithm. SGDSC has the best performance overall, with the other algorithms obtaining comparable results.

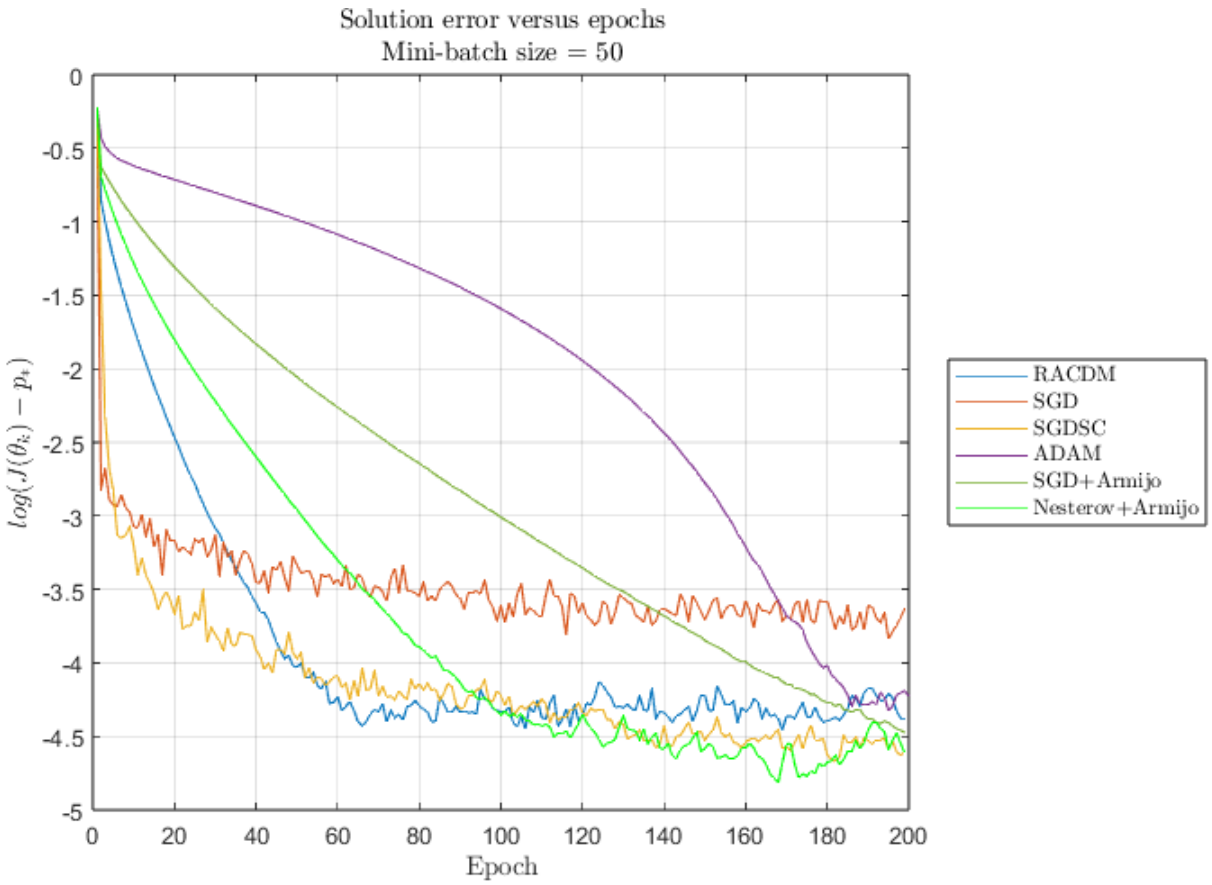


Figure 5.13: Stochastic gradient algorithms convergence graphs for non-separable data and mini-batch size = 50, bounded to 200 epochs.

Summary

Focusing in the earlier stages of convergence, we observe that the algorithms do not present the similarities they present asymptotically. SGD and its strongly convex variant SGDSC are the fastest, but in the 200 epoch mark, obtain similar accuracy with all the other algorithms, albeit them being slower in earlier epochs. ADAM remains the slowest converging algorithm in earlier epochs, in the non-separable case.

5.4.3 Comparison Between Full and Stochastic Gradient Algorithms for Non-separable Data

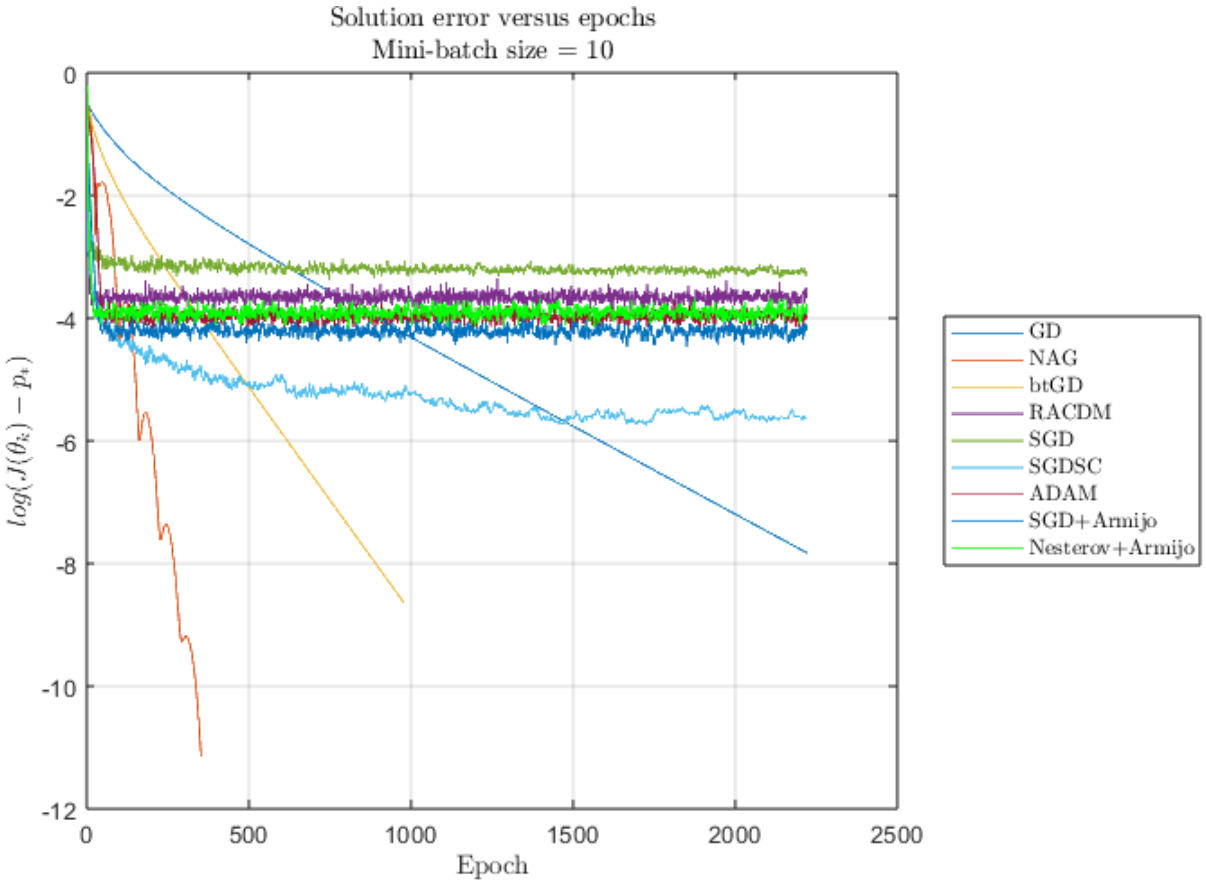


Figure 5.14: Comparison of convergence graphs of all algorithms studied for non-separable data and mini-batch size = 10.

Summary

As mentioned in the experiment with the full gradient algorithms, no algorithm obtains the same accuracy in the case of non-separable data as that in the separable data experiments. Contrary to the separable data case, we can observe that stochastic algorithms no longer attain comparable accuracy to full gradient algorithms. Although, stochastic algorithms remain faster than full gradient algorithms. The most favorable performance is that of NAG, even if not being as fast as stochastic algorithms.

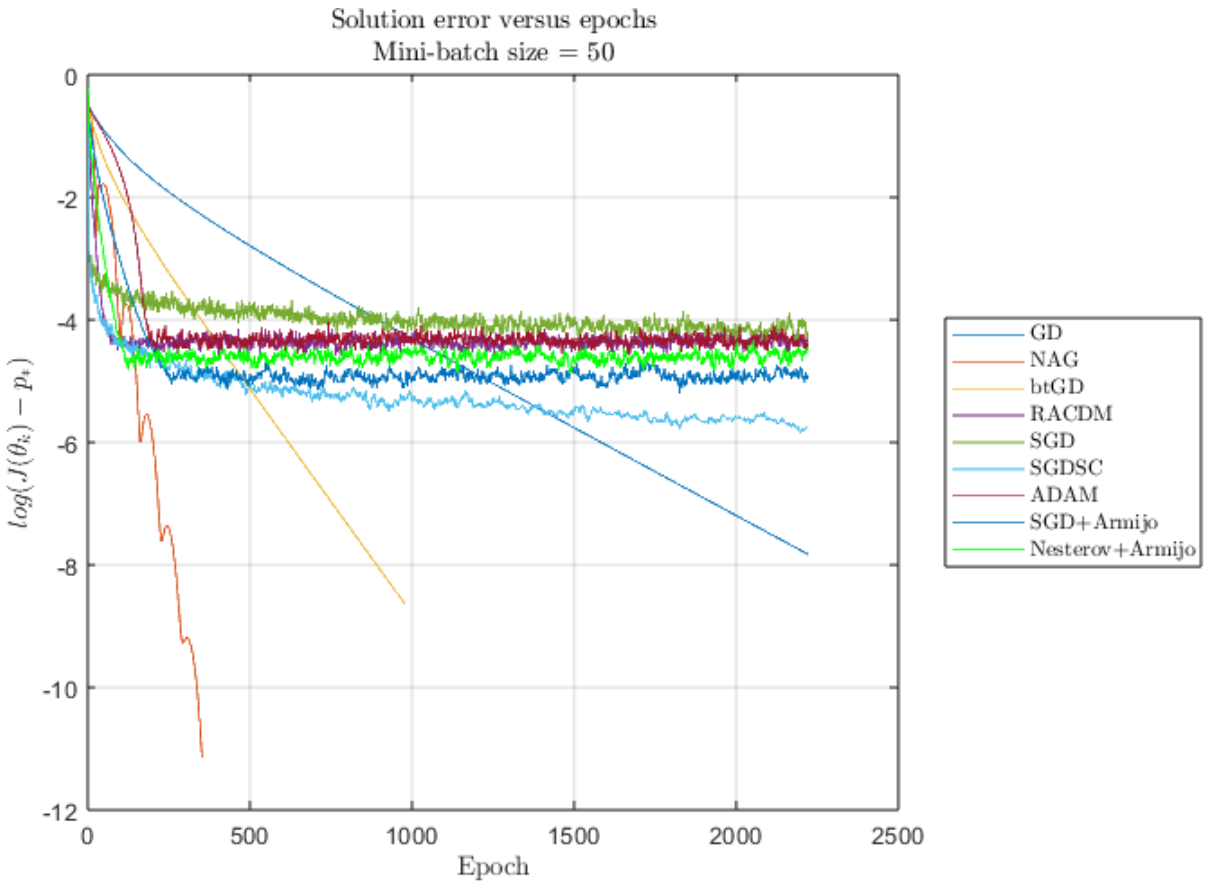


Figure 5.15: Comparison of convergence graphs of all algorithms studied for non-separable data and mini-batch size = 50.

Summary

Utilizing a larger mini-batch size, stochastic algorithms obtain better accuracy, but, in this case too, is not comparable to the accuracy of full gradient algorithms. The faster algorithms are observed to be the SGD variants and the most accurate algorithm is NAG, with the other full gradient algorithms displaying similar accuracy. We can observe that, like the separable data case, with a larger mini-batch size stochastic algorithms are slower and their speed is comparable to that of NAG, in earlier epochs.

Chapter 6

Conclusion and Future Work

In this thesis we studied various algorithms to solve the well established problem of regularized Logistic Regression. We defined the functions formulating the problem, derived the cost function's gradient and Hessian and studied its properties, in order to utilize them to improve the performance of the algorithms studied. We studied and implemented full and stochastic gradient algorithms, from the well known Gradient Descent to the more complex ADAM algorithm, in order to study the differences in the convergence of these algorithms. We performed various experiments to practically prove the theoretical results of the algorithms studied, using frameworks such as mini-batches and regularization to further illuminate the differences between the algorithms, as well as their advantages and disadvantages.

Future work could be consisted of further experimentating with hyperparameters values, other than their default values, more strict models by using smaller regularization constants and more complex algorithms to study possible gains in performance.

Regarding regularization, we could completely change the regularization scheme to observe the differences between the algorithms' performance and resulting accuracy.

Another factor that could influence performance and accuracy, could be the terminating condition in 3.1.4 we used in many algorithms. Choosing another terminating condition could lead to entirely different results in certain aspects of the algorithms' performance.

Other than tuning the algorithmic side of the problem, we could study methods such as data reformulations or augmentations, to take advantage of certain characteristics that may favor the performance of the algorithms, or even this specific formulation of the cost function, in order to eliminate data-driven limitations.

Lastly, we could rewrite some of the algorithms studied in a more efficient manner, mostly regarding the overhead that some steps may introduce to the optimization procedure.

Bibliography

[1] Amir Beck. *First-order Methods in Optimization*. SIAM, 2017.

[2] Léon Bottou, Frank E Curtis, and Jorge Nocedal. "Optimization Methods for Large-scale Machine Learning". In: *Siam Review* 60.2 (2018), pp. 223–311.

[3] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge university press, 2004.

[4] Stanley Chan. *Machine Learning Lecture Notes, Chapter 2*. Machine Learning Course , Purdue University. 2018. URL: https://engineering.purdue.edu/ChanGroup/ECE595/files/chapter2.pdf.

[5] Gradient Descent. *Gradient Descent — Wikipedia, The Free Encyclopedia*. 2022. URL: https://en.wikipedia.org/wiki/Gradient_descent.

[6] James Gareth et al. *An Introduction to Statistical Learning: with Applications in R*. Spinger, 2013.

[7] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. *Lecture Notes , Neural Networks for Machine Learning Course*. University of Toronto. 2018.

[8] IBM. *What is Logistic Regression?* URL: https://www.ibm.com/topics/logistic-regression.

[9] Daniel Jurafsky and James H Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*.

[10] Diederik P Kingma and Jimmy Ba. "ADAM: A Method for Stochastic Optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[11] Athanasios P. Liavas. *Gradient Descent Convergence Rate Notes, Convex Optimization Course*. Technical University of Crete. 2020.

[12] Athanasios P. Liavas. *Lecture Notes , Convex Optimization Course*. Technical University of Crete. 2020.

[13] A. Lindholm et al. *Machine Learning: A First Course for Engineers and Scientists*. Cambridge University Press, 2021. ISBN: 9781108919371. URL: https://books.google.gr/books?id=nIWazgEACAAJ.

[14] Yu Nesterov. "Efficiency of Coordinate Descent Methods on Huge-scale Optimization Problems". In: *SIAM Journal on Optimization* 22.2 (2012), pp. 341–362.

[15] Logistic Regression. *Logistic Regression — Wikipedia, The Free Encyclopedia*. 2022. URL: https://en.wikipedia.org/wiki/Logistic_regression.

[16] Herbert Robbins and Sutton Monro. “A Stochastic Approximation Method”. In: *The annals of mathematical statistics* (1951), pp. 400–407.

[17] Mark Schmidt. *Lecture Notes , Machine Learning Course*. University of British Columbia. 2019.

[18] Mark Schmidt, Nicolas Le Roux, and Francis Bach. “Minimizing Finite Sums with the Stochastic Average Gradient”. In: *Mathematical Programming* 162.1 (2017), pp. 83–112.

[19] Ilya Sutskever. “Training Recurrent Neural Networks”. PhD thesis. University of Toronto, 2013.

[20] Ryan Tibshirani. *Lecture Notes , Convex Optimization Course*. Carnegie Mellon University. 2012 - 2018.

[21] Sharan Vaswani et al. “Painless Stochastic Gradient: Interpolation, Line-search, and Convergence Rates”. In: *Advances in neural information processing systems* 32 (2019).

[22] Yuxuan Zhou. *Introduction to Accelerated GradientMethod*. URL: `https://github.com/ZhouYuxuanYX/Matlab-Implementation-of-Nesterov-s-Accelerated-Gradient-Method-/blob/master/Introduction%20to%20Accelerated%20Gradient%20Method.pdf`.