

Scalable Logistic Regression

Shuo Zhou, PhD

COM6012 Scalable Machine Learning
20.02.2023



Contents

Logistic regression

Introduction

Optimisation

Regularisation

Efficient computation

Quasi-Newton methods

Stochastic gradient descent

Distributed and parallel computation

Logistic regression in PySpark

Acknowledgment

References and recommended reading

Probabilistic classifier

- A logistic regression model is an example of a probabilistic classifier.
- Let $\mathbf{x} \in \mathbb{R}^p$ represents a feature vector and y the target value.
- $y \in \{0, 1\}$ or $y \in \{-1, +1\}$ for a binary classification problem.
- We model the relationship between y , and \mathbf{x} using a Bernoulli distribution.

Bernoulli distribution

- A Bernoulli random variable can only take two possible values.
- A Bernoulli distribution is a probability distribution for Y , expressed as

$$p(Y = y) = \text{Ber}(y|\mu) = \begin{cases} \mu & y = 1, \\ 1 - \mu & y = 0, \end{cases}$$

where $\mu = P(Y = 1)$.

- The expression above can be summarised in one line using

$$p(Y = y) = \text{Ber}(y|\mu) = \mu^y(1 - \mu)^{1-y},$$

How are y and \mathbf{x} related in logistic regression?

- The target value y follows a Bernoulli distribution

$$p(y|\mathbf{x}) = \text{Ber}(y|\mu(\mathbf{x})).$$

- Notice how the probability $\mu = P(y = 1)$ explicitly depends on \mathbf{x} .
- In logistic regression, the probability $\mu(\mathbf{x})$ is given as

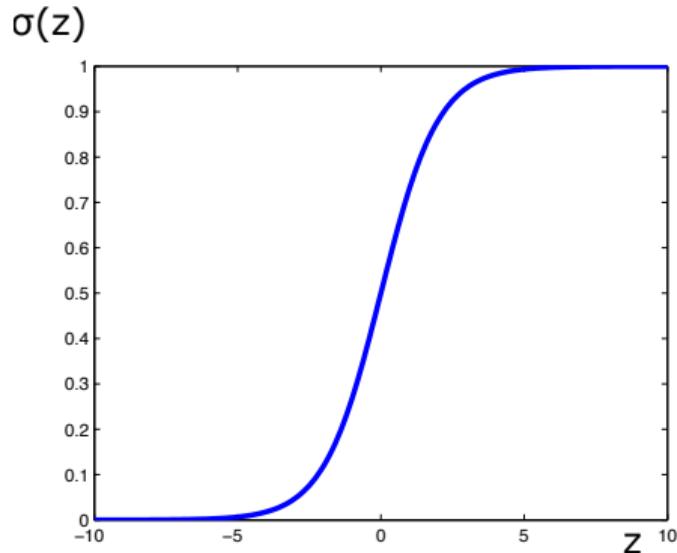
$$\mu(\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^\top \mathbf{x})} = \sigma(\mathbf{w}^\top \mathbf{x}),$$

where $\sigma(z)$ is known as the *sigmoid function*, and \mathbf{w} is the coefficient vector to learn.

- We then have

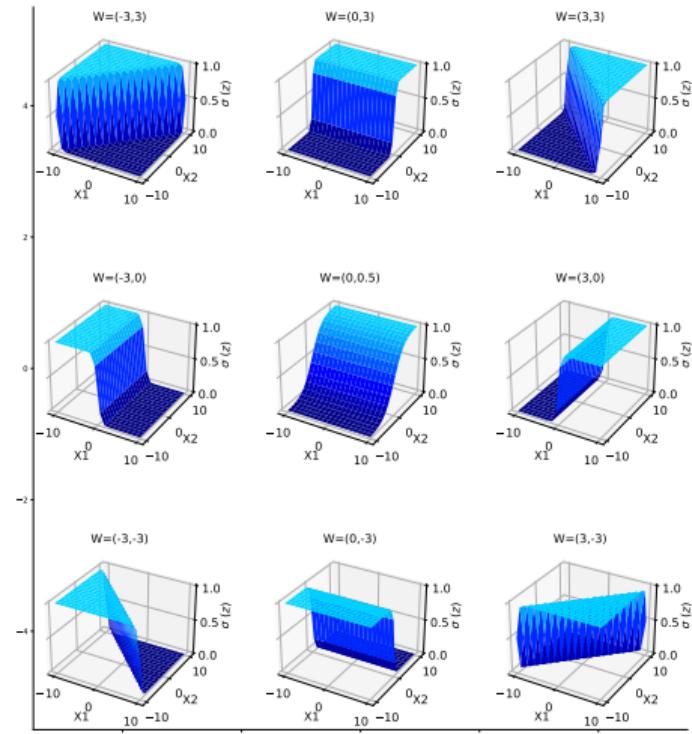
$$p(y|\mathbf{w}, \mathbf{x}) = \text{Ber}(y|\sigma(\mathbf{w}^\top \mathbf{x})).$$

The sigmoid function $\sigma(z)$



- Recall $\sigma(z) = \frac{1}{1 + \exp(-z)}$.
- If $z \rightarrow \infty$, $\sigma(z) = 1$. If $z \rightarrow -\infty$, $\sigma(z) = 0$. If $z = 0$, $\sigma(z) = 0.5$.
- Let $z = \mathbf{w}^\top \mathbf{x}$ in logistic regression.

The sigmoid function in 2d



Plot of $\sigma(w_1 x_1 + w_2 x_2)$. Here $\mathbf{w} = [w_1 \ w_2]^\top$. Generated by code at [this link](#).

Decision boundary

- After the training phase, we will have an estimation for \mathbf{w} .
- For a test input vector \mathbf{x}_* , we compute $p(y = 1 | \mathbf{w}, \mathbf{x}_*) = \sigma(\mathbf{w}^\top \mathbf{x}_*)$.
- This will give us a value between 0 and 1.
- We define a threshold of 0.5 to decide to which class we assign \mathbf{x}_* .
- With this threshold we induce a linear decision boundary in the input space.

Cross-entropy error function

- Let $\mathcal{D} = \{\mathbf{x}_n, y_n\}_{n=1}^N$.
- We write $\mathbf{X} = [\mathbf{x}_1 \cdots \mathbf{x}_N]^\top$, and $\mathbf{y} = [y_1 \cdots y_N]^\top$.
- Assuming IID observations

$$p(\mathbf{y}|\mathbf{w}, \mathbf{X}) = \prod_{n=1}^N p(y_n|\mathbf{w}, \mathbf{x}_n) = \prod_{n=1}^N \text{Ber}(y_n|\sigma(\mathbf{w}^\top \mathbf{x}_n)).$$

- The cross-entropy function or negative log-likelihood is given as

$$\begin{aligned} NLL(\mathbf{w}) &= -\log p(\mathbf{y}|\mathbf{w}, \mathbf{X}) \\ &= -\sum_{n=1}^N \{y_n \log[\sigma(\mathbf{w}^\top \mathbf{x}_n)] + (1 - y_n) \log[1 - \sigma(\mathbf{w}^\top \mathbf{x}_n)]\}, \end{aligned}$$

which can be minimised with respect to \mathbf{w} .

Contents

Logistic regression

Introduction

Optimisation

Regularisation

Efficient computation

Quasi-Newton methods

Stochastic gradient descent

Distributed and parallel computation

Logistic regression in PySpark

Acknowledgment

References and recommended reading

General problem

- We are given a function $\mathcal{L}(\mathbf{w})$, where $\mathbf{w} \in \mathbb{R}^p$.
- Aim: to find a value for \mathbf{w} that minimises $\mathcal{L}(\mathbf{w})$.
- Use an iterative procedure

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \eta \mathbf{d}_k,$$

where \mathbf{d}_k is known as the search direction in the k th step and it is such that

$$\mathcal{L}(\mathbf{w}_{k+1}) < \mathcal{L}(\mathbf{w}_k).$$

- The parameter η is known as the **step size** or **learning rate**.

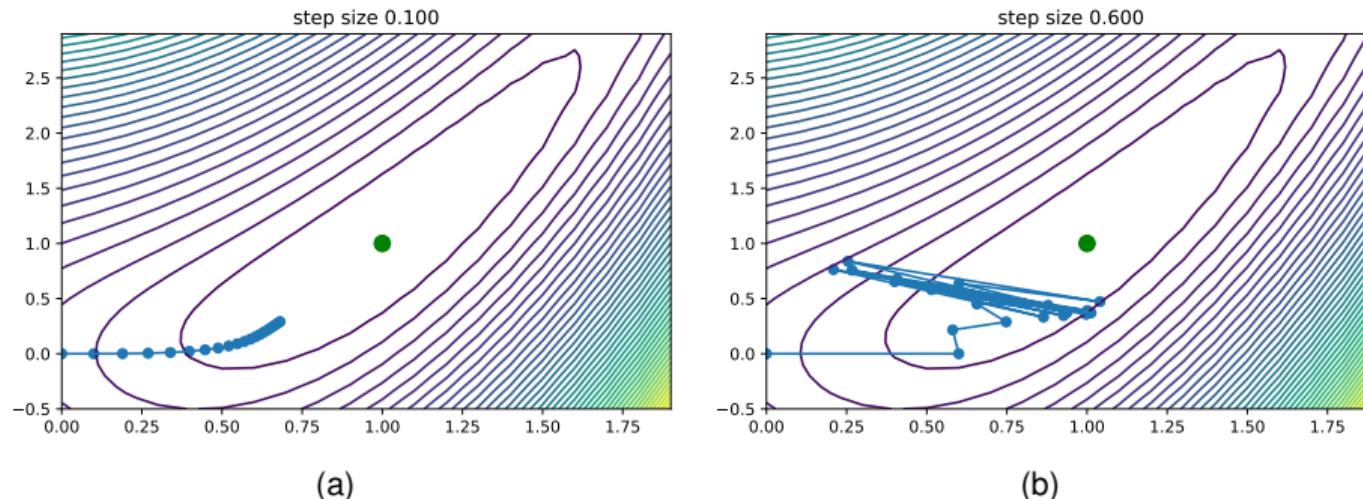
First-order method: gradient descent

- ❑ Perhaps, the simplest algorithm for unconstrained optimisation.
- ❑ It assumes that $\mathbf{d}_k = -\mathbf{g}_k$, where $\mathbf{g}_k = \nabla \mathcal{L}(\mathbf{w}_k)$.
- ❑ It can be written like

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \mathbf{g}_k.$$

Step size

- ❑ Main issue: If it is too small, convergence will be very slow. If it is too large, the method can fail to converge at all.



The function to optimise is $f(w_1, w_2) = 0.5(w_1^2 - w_2)^2 + 0.5(w_1 - 1)^2$. The minimum is at $(1, 1)$. In (a) $\eta = 0.1$. In (b) $\eta = 0.6$. Generated by code at [this link](#).

- ❑ Optimal step size can be found by **line search**.

Second-order method: Newton's method

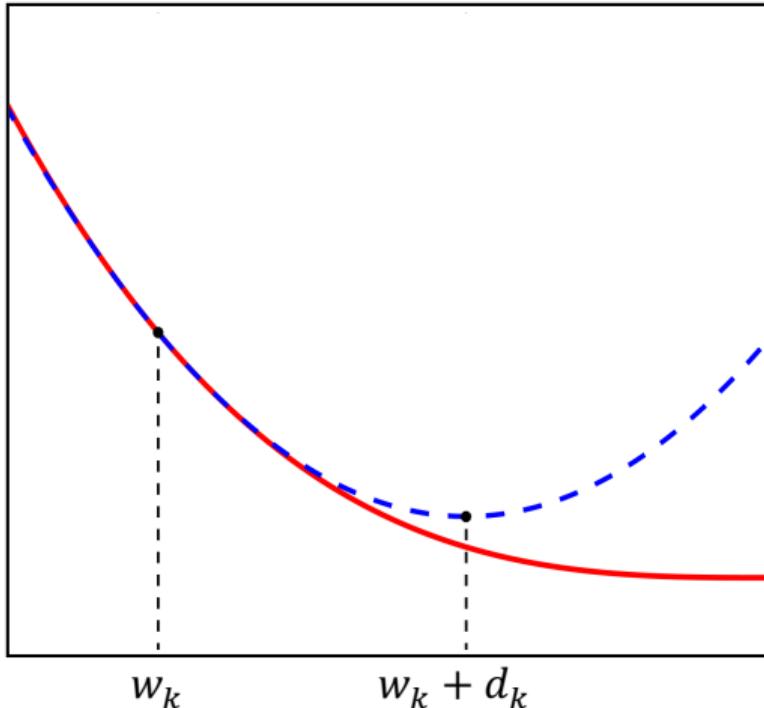
- Newton's algorithm

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \mathbf{H}_k^{-1} \mathbf{g}_k,$$

where $\mathbf{g}_k = \nabla \mathcal{L}(\mathbf{w}_k)$, and $\mathbf{H}_k = \nabla^2 \mathcal{L}(\mathbf{w}_k)$ is a Hessian matrix.

- Usually $\eta = 1$, and $\mathbf{d}_k = -\mathbf{H}_k^{-1} \mathbf{g}_k$.
- It derives a faster optimisation algorithm by taking the curvature of the space (i.e., the Hessian) into account.

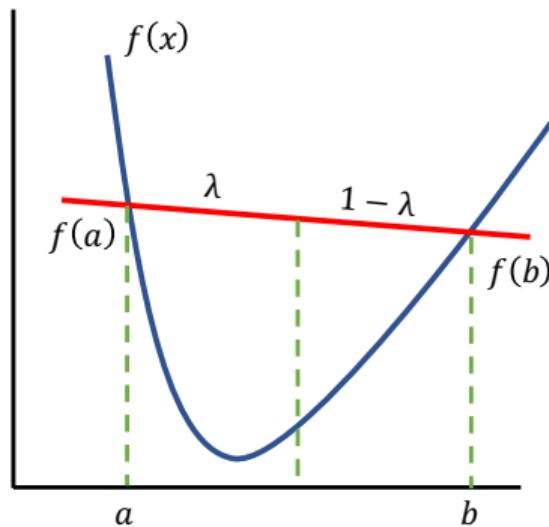
Newton's method (II)



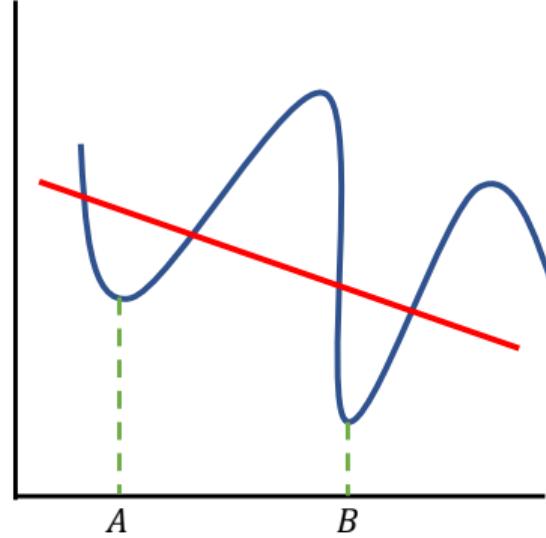
The solid curve is the function $\mathcal{L}(x)$. The dotted line $\mathcal{L}_{\text{quad}}(w)$ is its second order approximation at w_k . The Newton step d_k is what must be added to w_k to get to the minimum of $\mathcal{L}_{\text{quad}}(w)$

Newton's method and convex functions

- Newton's method requires that \mathbf{H}_k be **positive definite** (p.d.)
- The condition holds if the function is strictly convex.



(a)



(b)

In (a), an illustration of a convex function. The chord joining $(a, f(a))$ and $(b, f(b))$ lies above the function. In (b), a function that is neither convex nor concave.

Gradient and Hessian of $NLL(\mathbf{w})$

- It can be shown that the gradient $\mathbf{g}(\mathbf{w})$ of $NLL(\mathbf{w})$ is given as

$$\mathbf{g}(\mathbf{w}) = \frac{d}{d\mathbf{w}} NLL(\mathbf{w}) = \sum_{n=1}^N [\sigma(\mathbf{w}^\top \mathbf{x}_n) - y_n] \mathbf{x}_n = \mathbf{X}^\top (\boldsymbol{\sigma} - \mathbf{y}),$$

where $\boldsymbol{\sigma} = [\sigma(\mathbf{w}^\top \mathbf{x}_1) \cdots \sigma(\mathbf{w}^\top \mathbf{x}_N)]^\top$.

- It can also be shown that the Hessian $\mathbf{H}(\mathbf{w})$ of $NLL(\mathbf{w})$ is given as

$$\mathbf{H}(\mathbf{w}) = \frac{d}{d\mathbf{w}} \mathbf{g}(\mathbf{w})^\top = \sum_{n=1}^N \sigma(\mathbf{w}^\top \mathbf{x}_n) [1 - \sigma(\mathbf{w}^\top \mathbf{x}_n)] \mathbf{x}_n \mathbf{x}_n^\top = \mathbf{X}^\top \boldsymbol{\Sigma} \mathbf{X},$$

where $\boldsymbol{\Sigma} = \text{diag}([\sigma(\mathbf{w}^\top \mathbf{x}_1)[1 - \sigma(\mathbf{w}^\top \mathbf{x}_1)], \dots, \sigma(\mathbf{w}^\top \mathbf{x}_N)[1 - \sigma(\mathbf{w}^\top \mathbf{x}_N)]])$.

Contents

Logistic regression

Introduction

Optimisation

Regularisation

Efficient computation

Quasi-Newton methods

Stochastic gradient descent

Distributed and parallel computation

Logistic regression in PySpark

Acknowledgment

References and recommended reading

What is regularisation?

- It refers to a technique used for preventing overfitting in a predictive model.
- It consists in adding a term (a regulariser) to the objective function that encourages simpler solutions.
- With regularisation, the objective function for logistic regression would be

$$\mathcal{L}(\mathbf{w}) = NLL(\mathbf{w}) + \lambda R(\mathbf{w}),$$

where $R(\mathbf{w})$ is the regularisation term and λ the regularisation parameter.

- If $\lambda = 0$, we get $\mathcal{L}(\mathbf{w}) = NLL(\mathbf{w})$.

ℓ_2 regularisation

- Prior of \mathbf{w} : zero-mean normal distribution

$$f(w|0, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(w-0)^2}{2\sigma^2}\right),$$

where σ is the standard deviation of the distribution.

- Likelihood:

$$p(\mathbf{y}|\mathbf{w}, \mathbf{X})p(\mathbf{w}) = \prod_{n=1}^N p(y_n|\mathbf{w}, \mathbf{x}_n) \prod_{m=1}^p \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{w_m^2}{2\sigma^2}\right).$$

- Negative log-likelihood:

$$NLL(\mathbf{w}) + \frac{1}{2\sigma^2} \sum_{m=1}^p w_m^2.$$

- Convex, smooth

ℓ_1 regularisation

- Prior of \mathbf{w} : zero-mean Laplace distribution

$$f(\mathbf{w}|0, b) = \frac{1}{2b} \exp\left(-\frac{|\mathbf{w} - 0|}{b}\right),$$

where $b > 0$ is a scale parameter of the distribution.

- Likelihood:

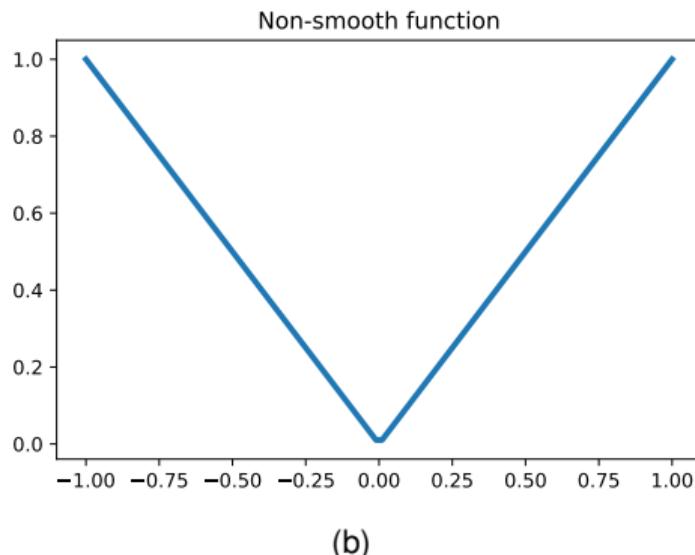
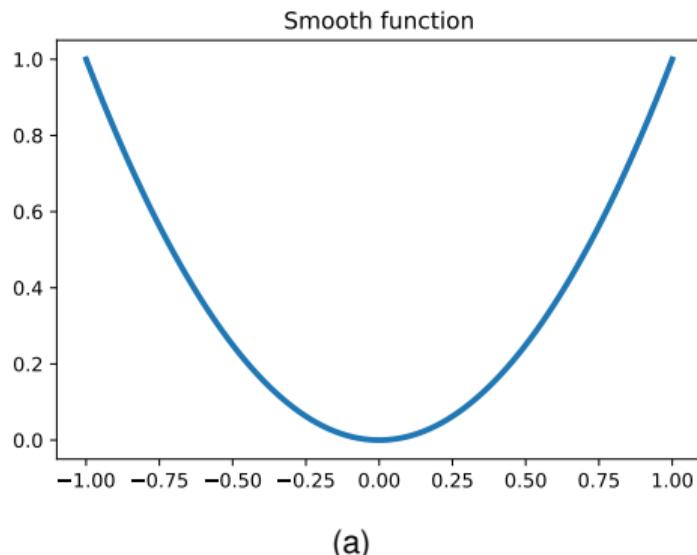
$$p(\mathbf{y}|\mathbf{w}, \mathbf{X})p(\mathbf{w}) = \prod_{n=1}^N p(y_n|\mathbf{w}, \mathbf{x}_n) \prod_{m=1}^p \frac{1}{2b} \exp\left(-\frac{|\mathbf{w}_m|}{b}\right).$$

- Negative log-likelihood:

$$NLL(\mathbf{w}) + \frac{1}{b} \sum_{m=1}^p |\mathbf{w}_m|$$

- Convex, nonsmooth, sparse

Smooth vs nonsmooth optimization



(a) Smooth 1d function, (b) Nonsmooth 1d function. Generated by code at [this link](#).

Elastic net regularisation

- The objective function for logistic regression would be

$$\mathcal{L}(\mathbf{w}) = NLL(\mathbf{w}) + \lambda R(\mathbf{w}),$$

where $R(\mathbf{w})$ follows as

$$R(\mathbf{w}) = \alpha \|\mathbf{w}\|_1 + (1 - \alpha) \frac{1}{2} \|\mathbf{w}\|_2^2,$$

where $\|\mathbf{w}\|_1 = \sum_{m=1}^p |w_m|$, and $\|\mathbf{w}\|_2^2 = \sum_{m=1}^p w_m^2$.

- If $\alpha = 1$, we get ℓ_1 regularisation.
- If $\alpha = 0$, we get ℓ_2 regularisation.
- If $0 < \alpha < 1$, we get the elastic net regularisation.

Contents

Logistic regression

Introduction

Optimisation

Regularisation

Efficient computation

Quasi-Newton methods

Stochastic gradient descent

Distributed and parallel computation

Logistic regression in PySpark

Acknowledgment

References and recommended reading

Quasi-Newton methods

- The main drawback of the Newton direction is the need for the Hessian.
- Explicit computation of this matrix of second derivatives can sometimes be a cumbersome, error-prone, and expensive process.
- Quasi-Newton search directions provide an attractive alternative to Newton's method.
- They do not require computation of the Hessian and yet still attain a faster rate of convergence.
- In place of the true Hessian \mathbf{H}_k , they use an approximation \mathbf{B}_k , which is updated after each step to take account of the additional knowledge gained during the step.

BFGS formula for \mathbf{H}_k

- In the Broyden, Fletcher, Goldfarb, and Shanno (BFGS) formula

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{\mathbf{z}_k \mathbf{z}_k^\top}{\mathbf{z}_k^\top \mathbf{s}_k} - \frac{(\mathbf{B}_k \mathbf{s}_k)(\mathbf{B}_k \mathbf{s}_k)^\top}{\mathbf{s}_k^\top \mathbf{B}_k \mathbf{s}_k},$$

where $\mathbf{s}_k = \mathbf{w}_k - \mathbf{w}_{k-1}$ and $\mathbf{z}_k = \mathbf{g}_k - \mathbf{g}_{k-1}$.

- This is a rank-two update to the matrix, and ensures that the matrix remains positive definite.
- Usually $\mathbf{B}_0 = \mathbf{I}$.
- The search direction is then $\mathbf{d}_k = -\mathbf{B}_k^{-1} \mathbf{g}_k$.

BFGS formula for \mathbf{H}_k^{-1}

- BFGS can alternatively update $\mathbf{C}_k = \mathbf{H}_k^{-1}$ using

$$\mathbf{C}_{k+1} = \left(\mathbf{I} - \frac{\mathbf{s}_k \mathbf{z}_k^\top}{\mathbf{z}_k^\top \mathbf{s}_k} \right) \mathbf{C}_k \left(\mathbf{I} - \frac{\mathbf{z}_k \mathbf{s}_k^\top}{\mathbf{z}_k^\top \mathbf{s}_k} \right) + \frac{\mathbf{s}_k \mathbf{s}_k^\top}{\mathbf{z}_k^\top \mathbf{s}_k}.$$

- The search direction is then $\mathbf{d}_k = -\mathbf{C}_k \mathbf{g}_k$.

Broyden, Fletcher, Goldfarb, Shanno



Image source: <https://aria42.com/blog/2014/12/understanding-lbfgs>

Limited memory BFGS (L-BFGS)

- Since storing the Hessian takes $O(p^2)$ space, for very large problems, one can use limited memory BFGS, or L-BFGS.
- In L-BFGS, \mathbf{H}_k or \mathbf{H}_k^{-1} is approximated by a diagonal plus low rank matrix.
- In particular, the product $\mathbf{B}_k^{-1} \mathbf{g}_k$ can be obtained by performing a sequence of inner products with \mathbf{s}_k and \mathbf{z}_k , using only the m most recent $(\mathbf{s}_k, \mathbf{z}_k)$ pairs, and ignoring older information.
- The storage requirements are therefore $O(mp)$. Typically $m \sim 20$ suffices for good performance.

Contents

Logistic regression

Introduction

Optimisation

Regularisation

Efficient computation

Quasi-Newton methods

Stochastic gradient descent

Distributed and parallel computation

Logistic regression in PySpark

Acknowledgment

References and recommended reading

Stochastic gradient descent (I)

- Traditionally in machine learning, the gradient \mathbf{g}_k and the Hessian \mathbf{H}_k are computed using the whole dataset $\mathcal{D} = \{\mathbf{x}_n, y_n\}_{n=1}^N$.
- Large datasets:** computing the exact values for \mathbf{g}_k and \mathbf{H}_k would be expensive.
- Stochastic gradient descent (SGD):**

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \nabla \mathcal{L}(\mathbf{w}_k, \mathbf{x}_k) = \mathbf{w}_k - \eta_t \mathbf{g}_k.$$

- Online learning:** the instances (\mathbf{x}_n, y_n) appear one at a time.

Stochastic gradient descent (II)

- In the stochastic setting, a better estimate can be found if the gradient is computed using

$$\mathbf{g}_k \approx \frac{1}{|S_k|} \sum_{i \in S_k} \nabla \mathcal{L}_i(\mathbf{w}_k),$$

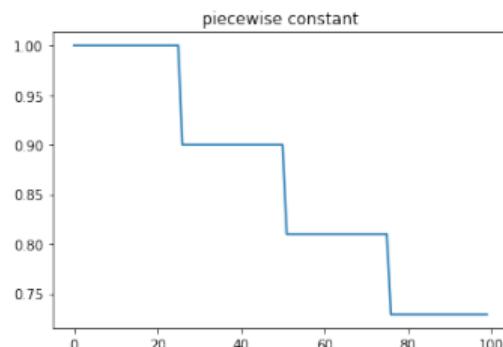
where $S_k \in \mathcal{D}$, $|S_k|$ is the cardinality of S_k , and $\nabla \mathcal{L}_i(\mathbf{w}_k)$ is the gradient at iteration k computed using the instance (\mathbf{x}_i, y_i) .

- This setting is called Mini-batch gradient descent.
- For logistic regression, $\nabla \mathcal{L}_i(\mathbf{w}_k)$ would be the gradient computed for $-\log p(y_i | \mathbf{w}, \mathbf{x}_i)$.

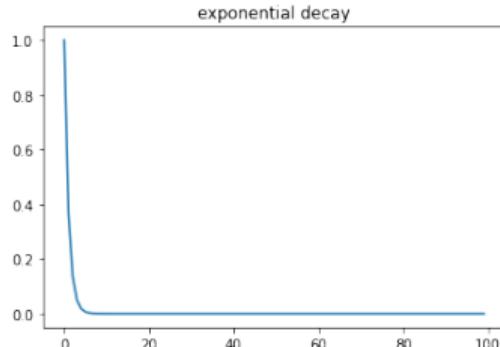
Step size in SGD

- Choosing the value of η is important in SGD since there is no easy way to compute it.
- It should follow the **Robbins-Monro** conditions

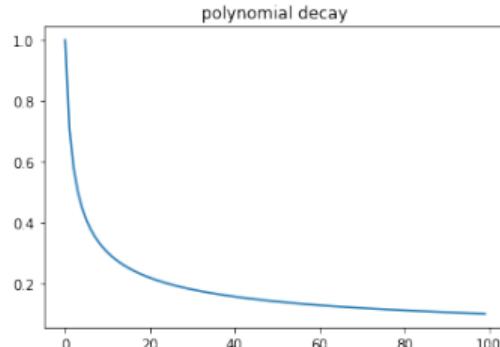
$$\eta_k \rightarrow 0, \frac{\sum_{k=1}^{\infty} \eta_k^2}{\sum_{k=1}^{\infty} \eta_k} \rightarrow 0.$$



(a)



(b)



(c)

Illustration of some common learning rate schedules. (a) Piecewise constant. (b) Exponential decay. (c) Polynomial decay. Generated by code at [this link](#).

Contents

Logistic regression

Introduction

Optimisation

Regularisation

Efficient computation

Quasi-Newton methods

Stochastic gradient descent

Distributed and parallel computation

Logistic regression in PySpark

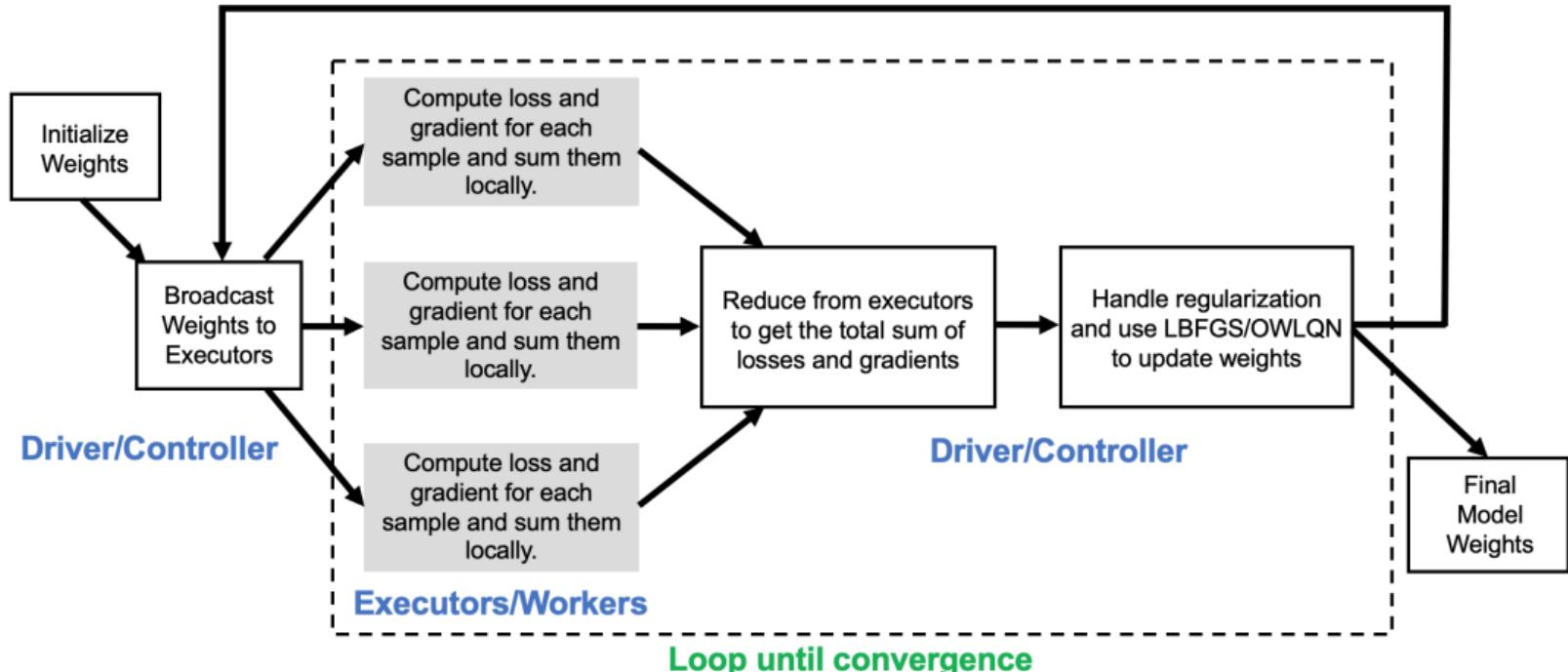
Acknowledgment

References and recommended reading

Algorithms in summation form

- When an algorithm sums over the data, we can easily distribute the calculations over multiple workers (e.g. cores).
- We just divide the dataset into as many pieces as there are workers, give each worker its share of the data to sum the equations over, and aggregate the result at the end.
- Both the gradient and Hessian in the optimisation algorithms we saw before have a summation form.
- Notice that the summation form does not change the underlying algorithm: it is not an approximation but an exact implementation.

Update of w in Apache Spark



Contents

Logistic regression

Introduction

Optimisation

Regularisation

Efficient computation

Quasi-Newton methods

Stochastic gradient descent

Distributed and parallel computation

Logistic regression in PySpark

Acknowledgment

References and recommended reading

LogisticRegression()

- ❑ LogisticRegression() works with DataFrames.
- ❑ It is possible to use regularisation ℓ_1 , ℓ_2 or Elastic Net.
- ❑ L-BFGS is used as a solver for LogisticRegression(), with ℓ_2 .
- ❑ When ℓ_1 , or Elastic Net are used, a variant of L-BFGS, known as Orthant-Wise Limited-memory Quasi-Newton (OWL-QN) is used instead.

Parameters to adjust

- ❑ **maxIter**: max number of iterations.
- ❑ **regParam**: regularization parameter (≥ 0).
- ❑ **elasticNetParam**: the ElasticNet mixing parameter, in range [0, 1]. For $\alpha = 0$, the penalty is an ℓ_2 penalty. For $\alpha = 1$, it is an ℓ_1 penalty.
- ❑ **family**: the name of family which is a description of the label distribution to be used in the model. Supported options: auto, binomial, multinomial.
- ❑ **standardization**: whether to standardize the training features before fitting the model. It can be true or false.

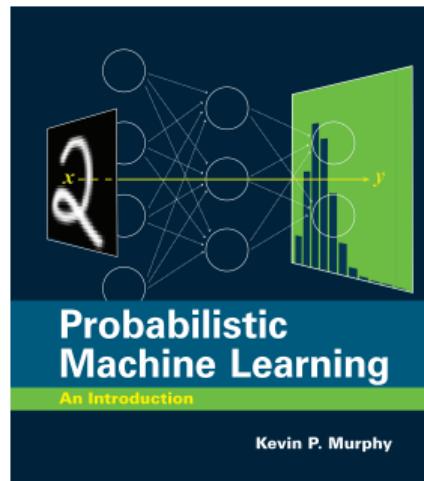
Acknowledgment

Thanks to [Dr. Mauricio Alvarez](#) for his contribution to this module from 2017 to 2022

References and recommended reading

Book: *Probabilistic Machine Learning: An Introduction* by Kevin P Murphy, 2022.

- Chapter 8 Optimization pp.217 -292
- Chapter 10 Logistic regression pp. 335-355



References and recommended reading

Website: *Spark Python API Documentation*.



Q. Search the docs ...

Spark SQL

Pandas API on Spark

Structured Streaming

Mlib (DataFrame-based)

Spark Streaming

Mlib (RDD-based)

Spark Core

Resource Management

API Reference

This page lists an overview of all public PySpark modules, classes, functions and methods.

Pandas API on Spark follows the API specifications of pandas 1.3.

- **Spark SQL**

- Core Classes
- Spark Session
- Configuration
- Input/Output
- DataFrame
- Column
- Data Types
- Row
- Functions
- Window
- Grouping
- Catalog
- Observation
- Avro

- **Pandas API on Spark**

- Input/Output
- General Functions
- Series
- DataFrame
- Index objects
- Window
- GroupBy
- Machine Learning utilities
- Extensions

- **Structured Streaming**

- Core Classes
- Input/Output
- Query Management

- **Mlib (DataFrame-based)**

- Pipeline APIs
- Parameters
- Feature
- Classification
- Clustering
- Functions
- Vector and Matrix
- Recommendation
- Regression
- Statistics
- Timing
- Evaluation
- Frequency Pattern Mining
- Image
- Utilities

- **Spark Streaming**

- Core Classes
- Streaming Management
- Input and Output
- Transformations and Actions
- Kinesis

- **Mlib (RDD-based)**

References used in this lecture

Paper: *Map-Reduce for Machine Learning on Multicore* by C-T Chu et al. (2006).

Map-Reduce for Machine Learning on Multicore

Cheng-Tao Chu *
chengtao@stanford.edu

Sang Kyun Kim *
skkim38@stanford.edu

Yi-An Lin *
ianl@stanford.edu

YuanYuan Yu *
yuanyuan@stanford.edu

Gary Bradski †
garybradski@gmail.com

Andrew Y. Ng *
ang@cs.stanford.edu

Kunle Olukotun *
kunle@cs.stanford.edu

* CS. Department, Stanford University 353 Serra Mall,
Stanford University, Stanford CA 94305-9025.

† Rexee Inc.

Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speed up. In this paper, we develop a broadly applicable parallel programming method, one that is easily applied to *many* different learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to speed up a *single* algorithm at a time. Specifically, we show that algorithms that fit the Statistical Query model [15] can be written in a certain “summation form,” which allows them to be easily parallelized on multicore computers. We adapt Google’s map-reduce [7] paradigm to demonstrate this parallel speed up technique on a variety of learning algorithms including locally weighted linear regression (LWLR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.

References and recommended reading

Paper: *Optimization Methods for Large Scale Machine Learning* by L. Bottou et al. (2018).

- Chapter 3-4 pp. 235 -258
- Chapter 6 pp. 270-284
- Chapter 8 pp. 293-298

SIAM REVIEW
Vol. 60, No. 2, pp. 223–311

© 2018 Society for Industrial and Applied Mathematics

Optimization Methods for Large-Scale Machine Learning*

Léon Bottou[†]
Frank E. Curtis[‡]
Jorge Nocedal[§]

Abstract. This paper provides a review and commentary on the past, present, and future of numerical optimization algorithms in the context of machine learning applications. Through case studies on text classification and the training of deep neural networks, we discuss how optimization problems arise in machine learning and what makes them challenging. A major theme of our study is that large-scale machine learning represents a distinctive setting in which the stochastic gradient (SG) method has traditionally played a central role while conventional gradient-based nonlinear optimization techniques typically falter. Based on this viewpoint, we present a comprehensive theory of a straightforward, yet versatile SG algorithm, discuss its practical behavior, and highlight opportunities for designing algorithms with improved performance. This leads to a discussion about the next generation of optimization methods for large-scale machine learning, including an investigation of two main streams of research on techniques that diminish noise in the stochastic directions and methods that make use of second-order derivative approximations.

Key words. numerical optimization, machine learning, stochastic gradient methods, algorithm complexity analysis, noise reduction methods, second-order methods