编号：　　级别：公开

优化基本理论与方法课程研究报告

Optimizing Unconstrained Problems with Sharpened-BFGS

（2024 年 1 月）

郑皓壬　　（3220103230）

郑俊达　　（3220103540）

李瀚轩　　（3220106039）

浙江大学计算机科学与技术学院

Contents

Abstract

近期,对拟牛顿法的非渐进分析受到一定的关注。人们已经发现经典的BFGS 算法具有超线性的收敛性质,但是在实际应用中,BFGS 算法的收敛速度往往不够快。为了解决这个问题,人们提出了一系列的改进算法,其中就包括了 Greedy-BFGS 算法。Greedy-BFGS 算法通过直接近似目标函数的 Hessian 矩阵而非牛顿法的方向,使得算法具有局部的二次收敛速率。但是,由于 Greedy-BFGS 算法直接近似了 Hessian 矩阵,而在牛顿方向上并不一定准确,因此算法需要更多步迭代才能达到局部二次收敛速率。为了进一步提高收敛速度,论文提出 Sharpened-BFGS 算法,通过结合两者的特点,实现了在更少的步数内达到局部二次收敛速率。数值实验也验证了该算法的优越性。

1 论文介绍

1.1 问题背景

拟牛顿法主要用于解决如下的无约束优化问题

$$\min_{x\in\mathbb{R}^d} f(x), \tag{1}$$

其中 $f:\mathbb{R}^d\to\mathbb{R}$ 是强凸的并且其梯度 Lipschitz 连续。我们用 x_* 表示问题 (1) 的唯一最优解。

一阶算法，即基于梯度的方法，是求解问题 (1) 最常用的方法，它以线性的速率（即误差以指数速率衰减）收敛到最优解。一阶算法的主要优势在于每次迭代的计算代价为 $\mathcal{O}(d)$，其中 d 是问题的维度。然而，这种方法的收敛速度会受目标函数的曲率影响，因此在不适定问题（条件数较大的问题）中，其收敛速度会变得很慢。为了解决这个问题，人们提出了牛顿法，即利用目标函数的 Hessian 矩阵改进曲率估计的二阶方法。牛顿法可以较好地处理不适定问题，且可以达到更快的局部收敛速率 [1][19][7][17]。在 Hessian 矩阵满足 Lipschitz 连续的条件下求解问题 (1) 时，牛顿法在局部可以达到二次收敛速率 [2, Chapter 9]，但是牛顿法在每次迭代时，都需要求解线性方程组，其代价为 $\mathcal{O}(d^3)$。

拟牛顿法是是一种介于一阶方法和二阶方法之间的方法，它比一阶的方法具有更快的收敛速率，即超线性收敛，且迭代的计算代价为 $\mathcal{O}(d^2)$，比牛顿法的 $\mathcal{O}(d^3)$ 代价更优。拟牛顿法的主要思想是，构造一个正定矩阵来近似牛顿法中目标函数的 Hessian 矩阵。因为拟牛顿法中对矩阵的更新只需要通过一系列矩阵与向量的乘法来实现，所以拟牛顿法的计算代价为 $\mathcal{O}(d^2)$。拟牛顿法有很多种，主要的区别在于如何迭代更新矩阵来近似 Hessian 矩阵，比如 SR1 方法 [6], Broyden 方法 [4][3][11], DFP 方法 [8][10], BFGS 方法 [5][9][12][23], L-BFGS 方法 [18][16]，以及 Greedy-QN 方法 [20]。

拟牛顿法的一个重要特点是，它们可以达到超线性收敛速率。其中，Rodomanov and Nesterov [20] 介绍并分析了一种新的拟牛顿法，即 Greedy-QN 方法。它基于经典的 Broyden 拟牛顿法，但是通过贪心地选择更新矩阵的方向，使得下降过程更快。Greedy-QN 方法的收敛速度在非渐进意义下是达到了 $(1-1/d\kappa)^{t^2/2}(d\kappa)^t$，其中 κ 是问题的条件数。注意到这一边界等价于 $((1-1/d\kappa)^{t/2}(d\kappa))^t$，说明二次的局部收敛速度在 $t\geq d\kappa\ln(d\kappa)$ 时可以达到。此外，相比经典的拟牛顿法，Greedy-QN 需要更

多信息，其中包括了每一次迭代中 Hessian 矩阵的对角元。Rodomanov and Nesterov [21] 证明了 BFGS 与 DFP 在内的经典拟牛顿法在非渐进意义下的超线性收敛速度。在满足目标函数强凸光滑且强自协调的条件下，它们分别以 $(d\kappa/t)^{t/2}$ 与 $(d\kappa^2/t)^{t/2}$ 的局部超线性速率收敛。而后 Rodomanov and Nesterov [22] 将两者的收敛速率分别改进为 $((d\ln\kappa)/t)^{t/2}$ 与 $((d\kappa\ln\kappa)/t)^{t/2}$。

1.2 论文贡献

如上所述，经典的 BFGS 方法旨在逼近牛顿方向，并在初始时就获得较快的收敛速度，但它无法完美逼近 Hessian 矩阵。另一方面，Greedy-BFGS 的目标是直接逼近 Hessian 矩阵，因此一开始它的收敛速度比 BFGS 慢，但在 Hessian 矩阵的近似完善后，它的收敛速度就会比 BFGS 快得多。因此论文提出了一个问题：

> 是否有一种可能的方法，可以两全其美地实现一种拟牛顿法，同时逼近牛顿方向与 Hessian 矩阵，从而用更少的迭代步数获得更快的收敛速度？

由此，论文提出了一种新颖的 Sharpened-BFGS 方法，它结合了 BFGS 和 Greedy-BFGS 的优点，通过近似牛顿方向来实现类似 BFGS 方法的初始快速收敛，同时实现类似 Greedy-BFGS 的对 Hessian 矩阵的更精确的估计，以达到二次收敛速率。论文中的这种方法在收敛速度方面优于 BFGS 和 Greedy-BFGS，同时与 Greedy-BFGS 方法相比，它以更少的迭代次数达到了超线性收敛速度。此外，Sharpened-BFGS 每次迭代的计算成本与经典 BFGS 及 Greedy-BFGS 相同。

1.3 章节组织

摘要部分概述了论文的主要贡献和结果，包括提出了一种新的 BFGS 方法，证明了它的非渐近超线性收敛速率和局部二次收敛速率，并在数值实验中验证了它的优越性。

在简介部分中，论文首先通过经典的无约束优化问题引入线性的下降法，并且由更快的收敛速率这一目标引出经典的牛顿法，并指出了牛顿法计算代价过大这一问题，引出了用正定矩阵近似目标函数的 Hessian 矩阵的方法，即拟牛顿法；随后主要介绍了三种拟牛顿法，即经典 BFGS、Greedy-BFGS 以及论文提出的 Sharpened-BFGS，并指出 Greedy-BFGS 的优越性。这一部分还提出了这篇论文的贡献以及与此相关的一些论文所做的工作。

在预备工作部分中，论文详细介绍了经典的 BFGS 算法与 Greedy-BFGS 算法，给出了两者的迭代方式，并计算了两种算法的收敛速度、计算代价、达到局部超线性速率需要的步数等。由上述的数据，论文指出了两种算法各自的优势与不足之处，并为了解决这些不足，引出了论文提出的 Sharpened-BFGS 算法。

在 Sharpened-BFGS 章节中，论文首先从 Hessian 矩阵恒定的二次规划入手，得出了一些结论，并给出了这种条件下的 Sharpened-BFGS 算法并相应地分析了经过迭代后误差的若干上界；随后论文又讨论了在更一般的条件下，即目标函数强凸光滑且 Hessian 矩阵满足 Lipschitz 连续条件的情况下，Sharpened-BFGS 算法的形式、误差上界、计算代价等，证明了 Sharpened-BFGS 算法的超线性速率。

在讨论章节中，论文通过时间 t 后的误差与最初的误差之比结合开始局部超线性收敛的时间，将三种算法进行了比较，得出了 Sharpened-BFGS 算法能够比 Greedy-BFGS 算法更快地达到局部超线性收敛速率，且 Sharpened-BFGS 算法的局部收敛速率远大于经典 BFGS 算法。

在数值实验部分，论文中采用了不同数据集的逻辑斯蒂回归问题，对比了 Sharpened-BFGS、Greedy-BFGS、经典 BFGS 以及线性的梯度下降法的对数误差曲线，并绘制了图表，对图表中的曲线简单分析，验证了 Sharpened-BFGS 算法相较于另两种拟牛顿法的优越性。

最后，在结论部分，论文总结了 Sharpened-BFGS 算法在解决无约束优化问题时的优势，并总结了论文的主要工作与内容结构。

2 相关工作

在目标函数强凸、光滑及其最优解处的 Hessian 矩阵满足 Lipschitz 连续的条件下，Jin and Mokhtari [13] 给出经典拟牛顿法的非渐近超线性收敛速度为 $(1/t)^{t/2}$。他们对自协调函数也给出了类似的结果。它们的局部收敛速度不依赖于 d 或 κ 等与问题有关的参数，但该结果要求 Hessian 矩阵逼近误差和到最优解的距离都足够小。此外，Ye et al. [24] 还明确给出了 SR1 方法的局部超线性收敛速率。Lin et al. [14] 还扩展了求解非线性方程的 Broyden 系列拟牛顿法的非渐近局部超线性收敛速率。值得注意的是，Lin et al. [15]提出了一种基于随机的 Greedy-BFGS，其收敛速度为 $(d\kappa(1-\frac{1}{d})^{\frac{t}{2}})^t$。这种随机技术也可用于论文中提出的 Sharpened BFGS 方法以改善其收敛速度对 κ 的依赖性。论文的附录中展示并分析了基于随机的 Sharpened-BFGS 方法。

3 问题描述和常用记号

这一节主要介绍论文中用到的一些记号，以及经典的 BFGS 算法与 Greedy-BFGS 算法。$x_t \in \mathbb{R}^d$ 为与时间 t 相关的迭代，$\nabla f(x_t) \in \mathbb{R}^d$ 为在 x_t 处目标函数的梯度。拟牛顿法更新的一般形式由下式给出

$$x_{t+1} = x_t - \eta_t G_t^{-1} \nabla f(x_t), \tag{2}$$

其中 $\eta_t > 0$ 是步长（学习率），$G_t \in \mathbb{R}^{d\times d}$ 是用于近似 Hessian 矩阵 $\nabla^2 f(x_t) \in \mathbb{R}^{d\times d}$ 的矩阵。一般而言，η_t 由一些线搜索算法得到，使得经过迭代后可以收敛到全局最优解。论文主要关注拟牛顿算法的局部收敛分析，因此假设 $\eta_t = 1$。在这种情况下，即可假设 $\{x_t\}_{t=1}^{\infty}$ 在 x_* 的领域内，且 $\eta_t = 1$ 总是可行的。

3.1 BFGS 算子与算法

拟牛顿法的本质是对 Hessian 的近似矩阵 G_t 的迭代更新。更新 G_t 的方法有很多种，论文中组要研究 BFGS 方法。在阐述 BFGS 方法之前，首先我们将其理解为一种近似线性算子的算法，这种观点有利于将其与其贪婪变体统一起来。令 $A \in \mathbb{R}^{d\times d}$ 为一个正定的线性算子，并假设 $G \in \mathbb{R}^{d\times d}$ 是近似 A 的一个算子并且按 BFGS 方法进行更新。BFGS 对于矩阵 G 沿方向 $u \in \mathbb{R}^d \backslash \{0\}$ 的更新为

$$BFGS(A, G, u) = G_+ := G - \frac{Guu^\top G}{u^\top Gu} + \frac{Auu^\top A}{u^\top Au}. \tag{3}$$

注意到，这次更新将 G 转移至 G_+，其中 A 与 G_+ 在方向 u 上的作用相同，即 $Au = G_+ u$。

Remark 1. 因为我们需要在每一次迭代时计算 Hessian 的近似矩阵的逆矩阵，所以在迭代中我们直接更新 Hessian 逆矩阵的近似。利用 Sherman-Morrison-Woodbury 公式可以证明 Hessian 逆近似矩阵 $H = G^{-1}$ 的更新可以写为

$$H_+ = \left(I - \frac{uu^\top A}{u^\top Au}\right) H \left(I - \frac{Auu^\top}{u^\top Au}\right) + \frac{uu^\top}{u^\top Au}. \tag{4}$$

由于其中只包含矩阵与向量的乘积，BFGS 的计算代价为 $\mathcal{O}(d^2)$。

为了最小化目标函数并且使近似的线性算子趋向于目标函数的曲率，我们令方

向向量 $u = x_{t+1} - x_t$ 并设算子为平均 Hessian 矩阵

$$A = J_t := \int_0^1 \nabla^2 f(x_t + \tau(x_{t+1} - x_t))d\tau.$$

这样可以保证新的 Hessian 近似矩阵 G_{t+1} 满足割线条件，即

$$G_{t+1}(x_{t+1} - x_t) = J_t(x_{t+1} - x_t) = \nabla f(x_{t+1}) - \nabla f(x_t),$$

定义差分 $s_t = x_{t+1} - x_t$ 以及梯度差 $y_t = \nabla f(x_{t+1}) - \nabla f(x_t)$，则经典的 BFGS 算法更新为

$$G_{t+1} = G_t - \frac{G_t s_t s_t^\top G_t}{s_t^\top G_t s_t} + \frac{y_t y_t^\top}{s_t^\top y_t}. \tag{5}$$

(5) 中 BFGS 算法更新的一个优点在于，新的 Hessian 近似矩阵 G_{t+1} 必定满足割线条件，即 $G_{t+1}s_t = y_t$。这个条件最终保证 BFGS 的下降方向 $G_t^{-1}\nabla f(x_t)$ 更接近牛顿法的方向 $\nabla^2 f(x_t)^{-1}\nabla f(x_t)$。

3.2 Greedy-BFGS 算法

经典的 BFGS 算法可以很好地逼近牛顿方向，但其 Hessian 近似矩阵可能无法逼近真正的 Hessian 矩阵。设两正定矩阵 $A, G \in \mathbb{R}^{d\times d}$ 之差为

$$\sigma(A, G) := Tr(A^{-1}G) - d, \tag{6}$$

其中，$Tr(X)$ 为矩阵 X 的迹，即 X 的对角元之和。当有 $A \preceq G$ 时，$\sigma(A, G)$ 可以作为 A 与 G 之间的距离的度量，且 $\sigma(A, G) = 0$ 当且仅当 $A = G$。用该距离函数作为度量，可以得到 BFGS 算子的 Hessian 近似误差如下：

Lemma 1. 设正定矩阵 $A, G \in \mathbb{R}^{d\times d}$ 并假设 $G_+ = BFGS(A, G, u)$ 且 $u \in \mathbb{R}^d\backslash\{0\}$。如果 $A \preceq G$，则

$$\sigma(A, G) - \sigma(A, G_+) \geq \frac{u^\top G u}{u^\top A u} - 1. \tag{7}$$

引理的证明见原论文附录。于是我们可以得到经过一步迭代后 Hessian 近似矩阵与原目标函数的 Hessian 矩阵之间的距离减小量。此外我们还可以发现，不同的方向向量 u 会影响势函数 $\sigma(A, G)$ 的下降。而 Hessian 近似矩阵对于任意的方向向量 $u \in \mathbb{R}^d\backslash\{0\}$ 并不一定能够收敛到目标函数的 Hessian 矩阵。如果我们令 $u = x^+ - x$，那同样不能保证 $\sigma(A, G)$ 可以趋于零。这又提出了一个新的问题，即如何选择方向向量 u 使得 $\sigma(A, G)$ 下降最大，并保证 $\sigma(A, G)$ 趋于零，即 G 收敛到 A。

Rodomanov and Nesterov [20] 提出了一个贪心方法来确定 u 的最佳选择。考虑一个二次问题，其中目标函数的 Hessian 矩阵是固定的并由正定矩阵 A 表示。在这种情况下，为了最大化不等式 (7) 的右侧（BFGS 更新的进度），可以选择 u 为

$$\bar{u}(A,G) := \arg\max_{u\in\{e_i\}_{i=1}^d} \frac{u^\top G u}{u^\top A u}, \tag{8}$$

其中 $\{e_i\}$ 是第 i 个元素为 1 且其余元素为 0 的向量。如果我们在 BFGS 更新 (3) 的每次迭代中选择 $u = \bar{u}(A,G)$，我们就得到 [20] 中的 Greedy-BFGS 算法。这种贪婪选择的优点是，它确保了 $\sigma(A,G)$ 严格递减并线性收敛到 0。

Lemma 2 ([20]). *设正定矩阵 $A, G \in \mathbb{R}^{d\times d}$，满足 $A \preceq G$ 且 $\mu I \preceq A \preceq LI$，其中 $0 < \mu \leq L$ 为常数。设 $\bar{G}_+ = BFGS(A, G, \bar{u}(A,G))$ 其中 $\bar{u}(A,G) \in \mathbb{R}^d$，$\bar{u}(A,G) \in \mathbb{R}^d$ 使用 (8) 中的贪心策略进行选择。可以得到*

$$\sigma(A, \bar{G}_+) \leq \left(1 - \frac{\mu}{dL}\right)\sigma(A,G). \tag{9}$$

由此可得，(6) 中的度量函数 $\sigma(.,.)$ 在遵循 Greedy-BFGS 策略更新 Hessian 近似矩阵的情况下，误差线性收敛到零，最终得到的 Hessian 近似矩阵序列趋向于目标 Hessian 矩阵。这对于非二次情况同样成立，但需要修改算法，因为平均 Hessian 矩阵 J_t 的计算成本很高，我们可以将其替换为当前的 Hessian 矩阵 $\nabla^2 f(x_t)$。

4 方法描述

Sharpened-BFGS 算法的核心思想就是同时使用经典的 BFGS 算法和贪心 BFGS 算法来更新 G_t。经典的 BFSG 算法可以提升牛顿方向的近似估计，而贪心 BFGS 算法能够提高整体 Heissan 矩阵的估计精确度。

4.1 二次规划

对于一般的优化问题 $\min_{x\in\mathbb{R}^n} f(x)$, 我们考虑目标函数为二次函数的情形，即

$$\min_{x\in\mathbb{R}^d} f(x) = \frac{1}{2}x^T A x + b^T x \tag{10}$$

其中 $A \in \mathbb{R}^{d\times d}$ 是对称正定矩阵，满足条件 $\mu I \preceq A \preceq LI, b \in \mathbb{R}^d$。

对于这类优化问题，Sharpened-BFGS 算法在每轮迭代的时候进行了两次更新。算法首先通过经典 BFGS 算法更新得到矩阵 $\overline{G}_t$，然后使用贪心 BFGS 算法选择下

降方向 $\overline{u}$, 最后由 $\overline{G}_t$ 和 $\overline{u}$ 更新得到 G_{t+1}。即 Shapened-BFGS 通过经典 BFGS 方向对贪心 BFGS 得到的 Hessian 矩阵估计进行了改进。伪代码如下：

Algorithm. 1 Sharpened-BFGS applied to quadratic programming

Require: 初始化变量 x_0, 初始化近似矩阵 $G_0 = LI$

for $t = 0, 1, 2, \cdots$ **do**

更新变量 $x_{t+1} = x_t - G_t^{-1}\nabla f(x_t)$;

计算 BFGS 方向 $s_t = x_{t+1} - x_t$;

计算 BFGS 矩阵 $\overline{G}_t = BFGS(A, G_t, s_t)$;

计算贪心 BFGS 方向 $\overline{u} = \overline{u}(A, \overline{G}_t)$;

估计 Hessian 矩阵 $G_{t+1} = BFGS(A, \overline{G}_t, \overline{u}_t)$;

end for

为了定量地描述 Sharpened-BFGS 算法的收敛速度，验证其确实融合了经典 BFGS 和贪心 BFGS 算法的优点，我们首先需要定义牛顿衰减因子：

$$\lambda_f(x) = \sqrt{\nabla f(x)^T \nabla^2 f(x)^{-1} \nabla f(x)} \tag{11}$$

其是用于衡量优化问题迭代过程中每一步近似解的改进程度的指标。它比较当前点的目标函数值和通过二阶信息进行近似预测的下降量。对于其形式的推导如下：

我们想要最小化 $f(\boldsymbol{x})$，考虑它的二阶泰勒展开：

$$f(\boldsymbol{x}) \approx f(\boldsymbol{x_0}) + (\boldsymbol{x} - \boldsymbol{x_0})^T \nabla f(\boldsymbol{x_0}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x_0})^T \nabla^2 f(\boldsymbol{x})(\boldsymbol{x} - \boldsymbol{x_0}) \tag{12}$$

由于海森矩阵 $\nabla^2 f(\boldsymbol{x_0})$ 是正定的，所以为了使 $f(\boldsymbol{x})$ 最小，我们将上式对 $\boldsymbol{x}$ 求偏导，得到：

$$\nabla f(\boldsymbol{x_0}) + \nabla^2 f(\boldsymbol{x_0})(\boldsymbol{x^*} - \boldsymbol{x_0}) = 0 \tag{13}$$

其中 $\boldsymbol{x^*}$ 是 $f(\boldsymbol{x})$ 的极小值点，所以有：

$$\boldsymbol{x^*} = \boldsymbol{x_0} - \nabla^2 f(\boldsymbol{x_0})^{-1} \nabla f(\boldsymbol{x_0}) \tag{14}$$

将 $\boldsymbol{x^*}$ 代入 $f(\boldsymbol{x})$ 的二阶泰勒展开式，得到：

$$\begin{aligned} f(\boldsymbol{x^*}) = f(\boldsymbol{x_0}) &- (\nabla^2 f(\boldsymbol{x_0})^{-1} \nabla f(\boldsymbol{x_0}))^T \nabla f(\boldsymbol{x_0}) \\ &+ \frac{1}{2}(\nabla^2 f(\boldsymbol{x_0})^{-1} \nabla f(\boldsymbol{x_0}))^T \nabla^2 f(\boldsymbol{x_0})(\nabla^2 f(\boldsymbol{x_0})^{-1} \nabla f(\boldsymbol{x_0})) \end{aligned} \tag{15}$$

因此我们得到牛顿减量的表达形式。$\lambda_f(x)$ 越小，说明当前点附近的二次模型逼近越可信，算法使用 λ_f 来判断是否终止迭代。在本节中，我们使用 $\lambda_t = \lambda_f(x_t)$ 的记号

来代表牛顿减量。下面我们介绍在二次规划问题中，Sharpened-BFGS 算法的更新结果，收敛速度（牛顿减量 λ_t 是如何收敛到零的）。

Lemma 3. 对于形如式 (1) 的二次规划问题和步长为 1 的拟牛顿法产生的迭代序列，我们有：

$$\lambda_{t+1} = \theta(A, G_t, x_{t+1} - x_t)\lambda_t \tag{16}$$

其中，

$$\theta(A, G, u) := \left(\frac{u^\top (G - A)A^{-1}(G - A)u}{u^\top GA^{-1}Gu}\right)^{\frac{1}{2}}. \tag{17}$$

该引理通过收缩因子 θ 量化了相邻两次迭代的牛顿减量之间的关系。观察收缩因子 θ 的表达式，反映了矩阵 G 和 A 在非零方向 u 上的相似程度，用于评估迭代方向上 Hessian 矩阵的改进性能。注意到连接两次牛顿减量的收缩因子表达式中的 $x_{t+1} - x_t$ 项，它强调牛顿减量收敛的因子与 $G_t(x_{t+1} - x_t), A(x_{t+1} - x_t)$ 之间的差距有关。

由于我们关心的是收敛速度问题，因此下面的定理给出 Sharpened-BFGS 算法下收缩因子的上界。

Theorem 1. 使用 Sharpened-BFGS 算法求解二次规划问题，则：

$$\theta(A, G_t, x_{t+1} - x_t) \leq 1 - \frac{\mu}{L}, \qquad \forall t \geq 0, \tag{18}$$

因此我们有：

$$\lambda_t \leq \left(1 - \frac{\mu}{L}\right)^t \lambda_0, \qquad \forall t \geq 0. \tag{19}$$

该定理指出，在每次迭代过程中收缩因子的上界与问题参数 μ, L 有关 (μ 强凸，L 光滑)，因此牛顿减量 λ_t 满足线性速率的衰减。虽然能够以线性速率收敛，但是由于这个上界仅仅由 G_t, A 的特征值都有界的性质得出，因此这个估计可能比实际的上限更加宽松。在接下来的引理和定理中，我们可以看到收缩因子序列最终收敛于零，Sharpened-BFGS 算法的收敛速度呈超线性。

Lemma 4. 考虑 Sharpened-BFGS 算法求解二次规划问题，进一步定义 $\theta_t := \theta(A, G_t, x_{t+1} - x_t)$,$\sigma_t := \sigma(A, G_t)$。则对于任意 $t \geq 0$：

$$\sigma_{t+1} \leq \left(1 - \frac{\mu}{dL}\right)\left(\sigma_t - \theta_t^2\right) \tag{20}$$

并且我们有：

$$\sum_{i=0}^{t-1} \frac{\theta_i^2}{(1 - \frac{\mu}{dL})^i} \leq \sigma_0, \qquad \forall t \geq 1. \tag{21}$$

回忆贪心 BFGS 算法中引入的函数 $\sigma(A,G)$，它可以用来衡量矩阵 A,G 的距离(相似程度)。该引理证明了 Sharpened-BFGS 算法中的 σ_t 与贪心 BFGS 算法相比具有更快的向零收敛的速率。值得注意的是，引理也给出了收缩因子随迭代轮次增加的上界变化情况，整个 θ_t 序列最终收敛到零。因此与前述定理相比，收缩因子具有更加紧密的上界。由此我们可以得到牛顿减量超线性收敛的结论，由下面的定理给出。

Theorem 2. 若使用 Sharpened-BFGS 算法求解二次规划问题，那么对于 $t \geq 1$ 的情况：

$$\lambda_t \leq \left(1-\frac{\mu}{dL}\right)^{\frac{t(t-1)}{4}} \left(\frac{dL}{t\mu}\right)^{\frac{t}{2}} \lambda_0. \tag{22}$$

观察牛顿减量收缩的表达式，其中 $\left(1-\frac{\mu}{dL}\right)^{\frac{t(t-1)}{4}}$ 线性收敛，而 $\left(\frac{dL}{t\mu}\right)^{\frac{t}{2}}$ 在 $t > d\frac{L}{\mu}$ 时 (即底数小于 1 时) 超线性收敛。因此，对于二次规划问题，Sharpened-BFGS 算法在迭代次数小于 $d\frac{L}{\mu}$ 时，以线性速度收敛，而在迭代次数大于 $d\frac{L}{\mu}$ 时，以超线性速度收敛，λ_t 以 $\mathcal{O}((1-\frac{\mu}{dL})^{t^2}(\frac{dL}{\mu t})^t)$ 的速率收敛到零。

4.2 一般的强凸光滑场景

在本节中，我们将 Sharpened-BFGS 算法扩展到一般的情形。为了建立算法的超线性收敛速率，我们需要如下的两个假设。

Assumption 4.1. 目标函数 f 二阶可微。f 是强凸函数，强凸参数为 $\mu > 0$, 并且 f 的梯度 ∇f 是 Lipschitz 连续的，Lipschitz 参数为 $L > 0$。

Assumption 4.2. 目标函数 f 是具有参数 M 的强自和谐函数。即对于任意 $x,y,z,w \in \mathbb{R}^n$, 我们有：

$$\nabla^2 f(y) - \nabla^2 f(x) \preceq M\|y-x\|_z \nabla^2 f(w) \tag{23}$$

其中 $\|y-x\|_z := \sqrt{(y-x)^\top \nabla^2 f(z)(y-x)}$

假设二中引入的强自和谐函数是为了分析拟牛顿法的二次收敛速率。观察它的表达式，$\nabla^2 f(y) - \nabla^2 f(x)$ 表示 f 在 y 和 x 处的 Hessian 矩阵的差异，即函数在这两点附近局部性质的变化。$\|y-x\|_z$ 表示向量 $y-x$ 通过 z 点 Hessian 矩阵的作用后产生变化的范数。该范数的引入确保了函数在点 x,y 之间的局部变化受到 Hessian 矩阵的适当控制。这个性质使得牛顿法在局部收敛时更为有效，可以证明算法在逼近最优解时的收敛速度至少是二次的。

下面给出一般的 Sharpened-BFGS 算法的伪代码:

Algorithm. 2 General Sharpened-BFGS

Require: 初始化变量 x_0, 初始化近似矩阵 $G_0 = LI$

for $t = 0, 1, 2, \cdots$ **do**

更新变量 $x_{t+1} = x_t - G_t^{-1}\nabla f(x_t)$;

计算 BFGS 方向 $s_t = x_{t+1} - x_t$;

计算沿 s_t 方向的平均海森矩阵, 作为 G_t 需要近似的值 $J_t = \int_0^1 \nabla^2 f(x_t + \tau s_t)d\tau$;

计算 $\bar{G}_t = BFGS(J_t, G_t, s_t)$;

计算修正项 $r_t = \|x_{t+1} - x_t\|_{x_t}$;

计算 $\hat{G}_t = (1 + Mr_t/2)^2\bar{G}_t$;

计算贪心 BFGS 方向 $\bar{u} = \bar{u}(\nabla^2 f(x_{t+1}), \hat{G}_t)$;

估计海森矩阵 $G_{t+1} = BFGS(\nabla^2 f(x_{t+1}), \hat{G}_t, \bar{u})$;

end for

算法的整体流程和 4.1 节所述的二次规划问题类似，都是在每一轮迭代中进行两次 BFGS 的更新估计，其中第一次 BFGS 得到的矩阵用来贪婪计算下降方向。与二次规划中的 Sharpened-BFGS 不同的地方在于，一般情形的算法在两次 BFGS 更新之间添加了修正项 $r_t = \|x_{t+1} - x_t\|_{x_t}$。由于在一般情形中，函数的 Hessian 矩阵并不是固定的 (二次规划中 Hessian 矩阵始终为 A)，我们并不能保证 $\nabla^2 f(x) \preceq G$ 始终成立，而只有当 $\nabla^2 f(x) \preceq G$ 时，$\sigma(\nabla^2 f(x), G)$ 才是良定义的。因此我们需要添加修正项确保在经过一次 BFGS 更新之后，新的点 x_+ 和新的 Hessian 逼近矩阵 G_+ 仍满足 $\nabla^2 f(x_+) \preceq G_+$。

Remark 2. 我们需要考虑算法的迭代计算成本是否因为修正项的增加而改变。注意到二次规划中的 Sharpened-BFGS 每轮迭代的计算成本为 $\mathcal{O}(d^2)$。而计算修正向量 r_t 和修正矩阵 $\hat{G}_t$ 的成本也是 $\mathcal{O}(d^2)$，因此修正项的引入后算法每轮迭代的计算成本并没有发生变化。

对一般情形的 Sharpened-BFGS 算法收敛速率的分析大体上与二次规划的情形类似。但是正如上文所述，一般情形下目标函数的 Hessian 矩阵会发生变化，这是我们需要考虑的地方。除此之外，需要注意的是在一般情形下，只有当初始点在最优解的局部邻域内，我们才能保证算法的收敛性。和二次规划问题相似，接下来一步

步给出一般情形下 Sharpened-BFGS 算法收敛速率的结论。首先我们给出牛顿减量的收缩因子。

Lemma 5. *考虑满足假设 4.1 和 4.2 的目标函数以及步长为 1 的拟牛顿法产生的迭代序列，我们有：*

$$\lambda_{t+1} \leq \left(1 + \frac{Mr_t}{2}\right) \theta(J_t, G_t, x_{t+1} - x_t)\lambda_t, \tag{24}$$

其中 $J_t := \int_0^1 \nabla^2 f(x_t + \tau(x_{t+1} - x_t))d\tau$, $r_t := \|x_{t+1} - x_t\|_{x_t}$。

在给出了相邻两次迭代牛顿减量的关系之后，我们需要考虑收缩因子 θ 的上界。与二次规划的步骤一致，下面的定理先给出较为宽松的上界。

Theorem 3. *考虑一般情形下的 Sharpened-BFGS 算法, 以及满足假设 4.1 和 4.2 的目标函数。假设初始点 x_0 满足：*

$$\lambda_0 \leq \frac{C_0\mu}{ML} \tag{25}$$

其中 $C_0 = \frac{1}{4}\ln\frac{3}{2}$，*则对任意* $t \geq 0$，*我们有：*

$$\theta(J_t, G_t, x_{t+1} - x_t) \leq 1 - \frac{2\mu}{3L}, \tag{26}$$

因此得出收缩因子的上界：

$$\lambda_t \leq (1 - \frac{\mu}{2L})^t \lambda_0. \tag{27}$$

由于限制了初始点的牛顿减量，因此上述定理给出的是在最优解的局部邻域内，Sharpened-BFGS 算法以 $1 - \frac{\mu}{2L}$ 的线性速率收敛。接下来的引理和定理将给出收缩因子更精确的上界，使得算法在一般情形下也能达到超线性的收敛速率。

Lemma 6. *考虑一般情形下的 Sharpened-BFGS 算法, 以及满足假设 4.1 和 4.2 的目标函数。假设初始点 x_0 满足：*

$$\lambda_0 \leq \frac{C_0\mu}{ML} \tag{28}$$

其中 $C_0 = \frac{1}{4}\ln\frac{3}{2}$。*定义* $\theta_t := \theta(\nabla^2 f(x_t), G_t, x_{t+1} - x_t)$, $\sigma_t := \sigma(\nabla^2 f(x_t), G_t)$. *则对任意* $t \geq 0$，*我们有：*

$$\sigma_{t+1} \leq (1 - \frac{\mu}{2dL}) \left[(1 + \frac{M\lambda_t}{2})^4(\sigma_t + 4Md\lambda_t) - \frac{1}{4}\theta_t^2\right]. \tag{29}$$

以及：

$$\sum_{i=0}^{t-1} \frac{\theta_i^2}{(1 - \frac{\mu}{2dL})^i} \leq 8(\sigma_0 + 4Md\lambda_0), \qquad \forall t \geq 1. \tag{30}$$

上述引理也是建立在最优解的局部邻域之内。具体而言，式 (20) 说明只要 λ_0 足够小 (初始点位于最优解的局部邻域)，那么 σ_t 可以收敛到零，即 G_t 能够很好地近似实际的 Hessian 矩阵 $\nabla^2 f(x_t)$，并且由于 θ_t^2 的存在，其收敛速率快于贪心 BFGS 算法。式 (21) 给出收缩因子 θ 更精确的上界，保证其能够以更快的速度收敛到零。有了上述的结论，我们可以得出如下的定理，给出一般情形下 Sharpened-BFGS 算法的收敛速率。

Theorem 4. *考虑一般情形下的 Sharpened-BFGS 算法, 以及满足假设 4.1 和 4.2 的目标函数。假设初始点 x_0 满足：*

$$\lambda_0 \leq \frac{C_1 \mu}{dML}, \tag{31}$$

其中 $C_1 = \frac{\ln 2}{20}$。那么对于任意的 $t \geq 1$，我们有：

$$\lambda_t \leq 2\left(1 - \frac{\mu}{2dL}\right)^{\frac{t(t-1)}{4}} \left(\frac{8dL}{t\mu}\right)^{\frac{t}{2}} \lambda_0. \tag{32}$$

类似于二次规划问题的分析，第一项 $\left(1 - \frac{\mu}{2dL}\right)^{\frac{t(t-1)}{4}}$ 线性收敛，第二项 $\left(\frac{8dL}{t\mu}\right)^{\frac{t}{2}}$ 在 $t \geq \frac{8dL}{t\mu}$ 时超线性收敛。总结来说，当迭代轮数 $t \leq \Theta(d\frac{L}{\mu})$ 时，算法具有线性收敛速率，当迭代轮数 $t \geq \Theta(d\frac{L}{\mu})$ 时，算法可以达到超线性的收敛速率。至此我们介绍了 Sharpened-BFGS 算法的具体思路方法，以及收敛性分析的推导过程。

5 理论结果

这一部分我们将 Sharpened-BFGS 的收敛结果与第三部分的经典 BFGS 和贪心 BFGS 进行比较。我们特别关注目标函数满足假设 4.1 和 4.2 的情况。为了简化比较，除了参数 μ, L, M, d 之外，其它的参数我们用常数 1 来替代，并且定义条件数 $\kappa = L/\mu \geq 1$。

Sharpened-BFGS. 根据第四部分的结果，如果我们设置初始近似矩阵 $G_0 = LI$, 并且初始点 x_0 满足：

$$\lambda_f(x_0) = \mathcal{O}\left(\frac{1}{dM\kappa}\right),$$

那么由 Sharpened-BFGS 算法产生的迭代序列满足：

$$\frac{\lambda_f(x_t)}{\lambda_f(x_0)} \leq \min\left\{\left(1-\frac{1}{\kappa}\right)^t, \left(1-\frac{1}{d\kappa}\right)^{\frac{t(t-1)}{4}}\left(\frac{d\kappa}{t}\right)^{\frac{t}{2}}\right\}.$$

从表达式可以看出，当 $t < d\kappa$ 时，第一个上界项更小，牛顿减量以线性速率 $\left(1-\frac{1}{\kappa}\right)^t$ 收敛。当 $t > d\kappa$ 时，第二个上界项更小，牛顿减量以 $\left(1-\frac{1}{d\kappa}\right)^{\frac{t(t-1)}{4}}\left(\frac{d\kappa}{t}\right)^{\frac{t}{2}}$ 的超线性速率收敛，并且可以看出其收敛速度快于二次收敛。

Greedy-BFGS. 如果我们设置初始近似矩阵 $G_0 = LI$, 并且初始点 x_0 满足：

$$\lambda_f(x_0) = \mathcal{O}\left(\frac{1}{dM\kappa}\right),$$

那么由 Greedy-BFGS 算法产生的迭代序列满足：

$$\frac{\lambda_f(x_t)}{\lambda_f(x_0)} \leq \min\left\{\left(1-\frac{1}{\kappa}\right)^t, \left(1-\frac{1}{d\kappa}\right)^{\frac{t(t-1)}{2}}\left(\frac{1}{2}\right)^t\right\}.$$

将其结果与 Sharpened-BFGS 的结果进行对比，我们可以发现：

1. Greedy-BFGS 算法的迭代轮数达到 $d\kappa \ln(d\kappa)$ 之后，收敛速度达到超线性。这是慢于 Sharpened-BFGS 算法 ($d\kappa$ 轮后达到超线性) 的。
2. 由于当 t 充分大时，除了共同的二次收敛项 $(1-\frac{1}{d\kappa})^{t^2}$ 之外，$(\frac{d\kappa}{t})^{\frac{t}{2}} \ll (\frac{1}{2})^t$, 因此 Sharpened-BFGS 算法最终的超线性收敛速率快于 Greedy-BFGS 算法。

BFGS. 如果我们设置初始近似矩阵 $G_0 = LI$, 并且初始点 x_0 满足：

$$\lambda_f(x_0) = \max\left\{\mathcal{O}\left(\frac{1}{M\kappa}\right), \mathcal{O}\left(\frac{1}{Md\ln\kappa}\right)\right\},$$

那么由 BFGS 算法产生的迭代序列满足：

$$\frac{\lambda_f(x_t)}{\lambda_f(x_0)} \leq \min\left\{\left(1-\frac{1}{\kappa}\right)^t, \left(\frac{d\ln\kappa}{t}\right)^{\frac{t}{2}}\right\}.$$

我们可以看到，BFGS 算法在经过 $d\ln\kappa$ 轮后达到超线性收敛速率，这是快于 Sharpened-BFGS 算法的。但是，当 t 充分大时我们有

$$\left(1-\frac{1}{d\kappa}\right)^{\frac{t(t-1)}{4}}\left(\frac{d\kappa}{t}\right)^{\frac{t}{2}} \ll \left(\frac{d\ln\kappa}{t}\right)^{\frac{t}{2}}.$$

因此 BFGS 算法的超线性收敛速率是慢于 Sharpened-BFGS 算法的。

6 实验结果

6.1 实验设置

在这一节中，我们将在不同数据集上展开实验，将 Sharpened-BFGS 算法的表现与经典 BFGS 算法和 Greedy-BFGS 算法进行比较。我们只关注下面这个具有 l_2 正

Dataset	N	d	μ
svmguide3	1243	21	0.01
w8a	49749	300	0.0001
colon-cancer	62	2000	0.00001

Tab. 1: 样本大小 N, 维度 d, 正则化参数 μ.

则化的逻辑斯蒂回归问题：

$$\min_{x\in\mathbb{R}^d} f(x) = \frac{1}{N}\sum_{i=1}^{N}\log(1+\exp^{-y_i z_i^\top x}) + \frac{\mu}{2}\|x\|^2 \tag{33}$$

其中，$z_i \in \mathbb{R}^d$ 是数据点，比如一张图像的一维展开，在训练前会将其标准化，使得 $\|z_i\| = 1$；$y_i \in \{-1, 1\}$ 则是这个数据点对应的标签。

这个目标函数是一个参数为 $\mu > 0$ 的强凸函数，而且由于进行了标准化，还可得到它是一个 Lipschitz 参数为 $L = 1/4 + \mu$ 的光滑函数。同时，这个逻辑斯蒂回归函数还是强和谐的，具体可见 [20] 的 5.1 节。因此，在33式中定义的目标函数 $f(x)$ 满足假设4.1和4.2。

原论文在八个数据集上进行实验，我们从中选择了三个最具代表性的，分别是 svmguide3、w8a 和 colon-canser，每个数据集的参数可见表1。

6.2 算法实现

我们研究的算法是 (i) Sharpened-BFGS, (ii) standard BFGS, (iii) Greedy-BFGS, and (iv) gradient descent (GD)。在这里，我们通过使用 PyTorch 来实现这四个算法，定义了四个继承 `torch.optim.Optimizer` 的优化器，分别为 (i) `GD`, (ii) `BFGS`, (iii) `GreedyBFGS` 和 (iv) `SharpenedBFGS`。其中，`GD` 优化器实现了梯度下降算法，`BFGS` 实现了标准的 BFGS 算法，`GreedyBFGS` 实现了贪心 BFGS 算法，`SharpenedBFGS` 实现了本文提出的 Sharpened-BFGS 算法。

跟原论文一样，我们将起始点初始化为 $x_0 = (1/d^{3/2}) * \vec{\mathbf{1}}$，其中 $\vec{\mathbf{1}} \in \mathbb{R}^d$ 是一个全为 1 的向量。对于 BFGS 算法，均将 Hessian 近似矩阵初始化为 LI，步长设置为 1；对于梯度下降算法，则将步长设置为 $1/L$。在实践中发现，对于混合方法和贪心方法，不矫正 Hessian 近似矩阵效果会更好，即在算法2中令 $\hat{G}_t = \bar{G}_t$，因此我们将 M 设置为 0。

具体的代码实现详见附录9。

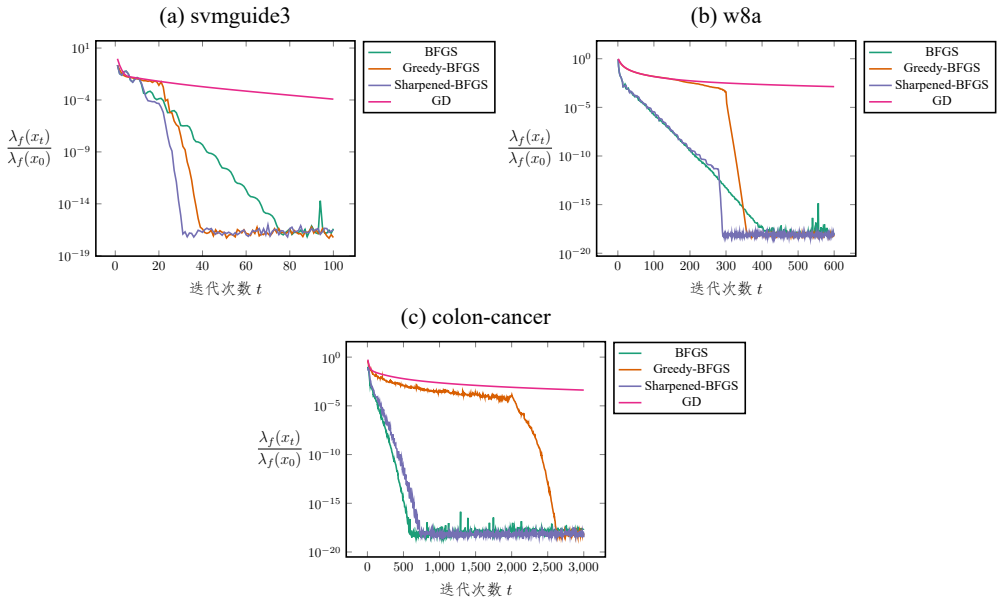

Fig. 1: 在不同数据集上 BFGS、Greedy-BFGS、Sharpened-BFGS 和 GD 的比较结果

6.3 结果分析

我们在不同的数据集上运行代码，得到了图1所示的结果。从图中可以看出，Sharpened-BFGS 算法的收敛速度确实快于经典的 BFGS 算法和 Greedy-BFGS 算法，就算稍慢也相差很小，这与理论分析的结果是一致的。更具体地说，在起始阶段，Sharpened-BFGS 利用了 BFGS 在牛顿方向的近似从而获得了较快的收敛速度，而不像 Greedy-BFGS 那样由于 Hessian 估计不够准确而收敛较慢；而一旦迭代轮数足够多，Hessian 近似矩阵足够精确，Sharpened-BFGS 又和 Greedy-BFGS 一样获得了超线性的收敛速度。上述理论结果在实验结果中得到了清晰的体现。

此外，我们还观察到了原论文没有提及的地方。在稳定性方面，可以看到，经典 BFGS 算法在达到最优点之后仍会出现较大幅度的震荡，而 Sharpened-BFGS 和 Greedy-BFGS 算法则没有这个问题，这是由于前者对 Hessian 矩阵的估计不准确，导致计算出来的搜索方向也不准确，从而偏离了最优点。

还有一个有趣的现象是，BFGS 算法在达到最优点之前以及 Sharpened-BFGS 算法在达到转折点之前，会出现较为明显的周期性震荡（这里的图表较小所以不太明显），而这一现象在 Greedy-BFGS 算法上并没有出现。这大概是也是因为 BFGS 和 Sharpened-BFGS 对于 Hessian 矩阵的估计不准确，导致搜索方向不准确，一会儿偏离正确的搜索方向，过一会儿再回到正确的搜索方向，呈现出一个动态平衡的过程，而这个问题对于 Greedy-BFGS 来说则不明显。

7 问题与挑战

虽然 Sharpened-BFGS 结合了经典 BFGS 算法和 Greedy-BFGS 算法的优点，但是它的应用场景还是有一定的局限性的。

首先，目标函数必须满足假设4.1和4.2，即目标函数必须是强凸的，其梯度必须是光滑的，并且是强自和谐的。这些假设对于一般的优化问题来说是比较严苛的，因此 Sharpened-BFGS 算法的应用场景也比较有限。

其次，在图1中也可以看到，对于较大的维数 d，Sharpened-BFGS 估计的近似矩阵很难收敛到 Hessian 矩阵，从而难以达到超线性收敛速度，实际表现还不如 BFGS 算法，却要花费更多的计算时间。由此可见，对于大规模的优化问题，Sharpened-BFGS 算法的效果可能并不好，还不如直接使用 BFGS 算法。

8 总结

在本篇论文中，作者提出了一种用于解决无约束凸优化问题新的拟牛顿方法——Sharpened-BFGS。其中目标函数具有 μ-强凸性，其梯度具有 L-光滑性，并且是 M-强自和谐的。Sharpened-BFGS 算法充分利用了经典 BFGS 算法在牛顿方向的近似和贪心 BFGS 算法的 Heissan 矩阵近似。利用这些性质，作者证明了该算法达到 $\mathcal{O}((1-\frac{\mu}{dL})^{\frac{t(t-1)}{4}}(\frac{dL}{t\mu})^{\frac{t}{2}})$ 的超线性收敛速率，并且其收敛速度快于二次收敛。作者也通过理论分析和数值实验将该方法和经典 BFGS 和 Greedy-BFGS 算法的收敛速率进行了比较，凸显了 Sharpened-BFGS 算法的优越性。

在学习这篇论文的工作时，我们体会到了不同优化方法的差异以及如何充分利用它们各自的优势得到表现更好的优化方法。同时，我们在跟着论文一步步推导并最终得到结果的过程中也学习到了如何从理论上分析算法的收敛速度，以及如何通过数值实验来验证理论结果，受益匪浅。

9 附录

GD、BFGS、Greedy-BFGS 和 Sharpened-BFGS 算法的代码实现如下：

```python
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.tensorboard import SummaryWriter
from sklearn.datasets import load_svmlight_file

configs = {
    'device': 'cuda' if torch.cuda.is_available() else 'cpu',
    'need_record': False,
    'max_iter': 3000,
    'M': 0,
    'optimizer': 'SharpenedBFGS', # GD, BFGS, GreedyBFGS, SharpenedBFGS
    'dataset': 'colon-cancer',
    'dataset_configs': {
        'svmguide3': {
            'N': 1243,
            'd': 22,
            'mu': 1e-2,
            'data_path': './dataset/svmguide3.txt'
        },
        'w8a': {
            'N': 49749,
            'd': 300,
            'mu': 1e-4,
            'data_path': './dataset/w8a.txt'
        },
        'colon-cancer': {
            'N': 62,
            'd': 2000,
            'mu': 1e-5,
            'data_path': './dataset/colon-cancer'
        },
    },
}

class Objective(nn.Module):
    def __init__(self, data, lables):
        super(Objective, self).__init__()
        self.x = nn.Parameter(torch.ones(d) / (d * np.sqrt(d)))
        self.y = torch.from_numpy(lables).to(device)
        self.z = F.normalize(torch.from_numpy(data), dim=1).to(device)
        self.z = -self.y.reshape(-1, 1) * self.z

    def forward(self):
        t = torch.log(1 + torch.exp(self.z @ self.x))
        return torch.sum(t) / N + mu / 2 * torch.dot(self.x, self.x)

    def hessian(self):
        with torch.no_grad():
            t = torch.exp(self.z @ self.x).reshape(-1, 1)
            I = torch.eye(d)
            return self.z.T @ (self.z * t / (1 + t)**2) / N + mu * I
```

```
55
56  class GD(optim.Optimizer):
57      def __init__(self, obj_func, need_record=False):
58          self.obj_func = obj_func
59          params = list(obj_func.parameters())
60          super(GD, self).__init__(params, {})
61
62          self.zero_grad()
63          loss = self.obj_func()
64          loss.backward()
65          self.x = obj_func.x
66          self.g = obj_func.x.grad.data.clone()
67
68          self.k = 0
69          self.need_record = need_record
70          self.lambda_0 = self.newton_decrement()
71
72      def step(self):
73          # compute the gradient and update the parameters
74          self.zero_grad()
75          loss = self.obj_func()
76          loss.backward()
77          self.g = self.x.grad.data.clone()
78          self.x.data = self.x.data - 1./L * self.g
79
80          self.record(loss)
81          return loss
82
83      def record(self, loss):
84          # update the iteration number and record
85          self.k += 1
86          if self.need_record:
87              ratio = self.newton_decrement() / self.lambda_0
88              writer.add_scalar('ratio', ratio, self.k)
89              writer.add_scalar('loss', loss.item(), self.k)
90
91      def newton_decrement(self):
92          with torch.no_grad():
93              g = self.g
94              H = self.obj_func.hessian()
95              return torch.sqrt(torch.dot(g, H @ g))
96
97  class BFGS(GD):
98      def __init__(self, obj_func, need_record=False):
99          super(BFGS, self).__init__(obj_func, need_record=need_record)
100         self.B = torch.eye(d) * L
101
102     def step(self):
103         s = self.update_params()
104         y, loss = self.update_grad()
105
106         # update the Hessian approximation
107         t = self.B @ s
108         self.B = self.B - torch.outer(t, t) / torch.dot(t, s) + torch.outer(y, y) /
            ↪  torch.dot(y, s)
109
110         self.record(loss)
111         return loss
112
```

```
    def update_params(self):
        # update the parameters and return the difference
        s = -torch.inverse(self.B) @ self.g
        self.x.data = self.x.data + s
        return s

    def update_grad(self):
        # update the gradient and return the difference
        self.zero_grad()
        loss = self.obj_func()
        loss.backward()
        gk = self.x.grad.data.clone()
        y = gk - self.g
        self.g = gk
        return y, loss

class GreedyBFGS(BFGS):
    def __init__(self, obj_func, need_record=False):
        super(GreedyBFGS, self).__init__(obj_func, need_record=need_record)

    def step(self):
        s = self.update_params()
        y, loss = self.update_grad()

        # update the Hessian approximation
        B_bar = self.B
        H_new = self.obj_func.hessian()
        index = 0
        max_res = B_bar[0, 0] / H_new[0, 0]
        for i in range(1, d):
            res = B_bar[i, i] / H_new[i, i]
            if res > max_res:
                max_res = res
                index = i
        t = B_bar[:, index]
        y = H_new[:, index]
        self.B = B_bar - torch.outer(t, t) / B_bar[index, index] + torch.outer(y, y) /
        ↪  H_new[index, index]

        self.record(loss)
        return loss

class SharpenedBFGS(BFGS):
    def __init__(self, obj_func, need_record=False):
        super(SharpenedBFGS, self).__init__(obj_func, need_record=need_record)

    def step(self):
        H = self.obj_func.hessian()
        s = self.update_params()
        y, loss = self.update_grad()

        # update the Hessian approximation
        t = self.B @ s
        self.B = self.B - torch.outer(t, t) / torch.dot(t, s) + torch.outer(y, y) /
        ↪  torch.dot(y, s)

        r = torch.sqrt(torch.dot(s, H @ s))
        B_bar = (1 + 0.5 * M * r) * self.B
        H_new = self.obj_func.hessian()
```

```
        index = 0
        max_res = B_bar[0, 0] / H_new[0, 0]
        for i in range(1, d):
            res = B_bar[i, i] / H_new[i, i]
            if res > max_res:
                max_res = res
                index = i
        t = B_bar[:, index]
        y = H_new[:, index]
        self.B = B_bar - torch.outer(t, t) / B_bar[index, index] + torch.outer(y, y) /
        ↪  H_new[index, index]

        self.record(loss)
        return loss

if __name__ == '__main__':
    device = configs['device']
    need_record = configs['need_record']
    max_iter = configs['max_iter']
    M = configs['M']

    dataset = configs['dataset']
    dataset_configs = configs['dataset_configs'][dataset]
    N = dataset_configs['N']
    d = dataset_configs['d']
    mu = dataset_configs['mu']
    data_path = dataset_configs['data_path']
    optim_name = configs['optimizer']

    torch.set_default_device(device)
    torch.set_default_dtype(torch.float64)

    data, labels = load_svmlight_file(data_path)
    data = data.todense()

    L = mu + 0.25
    obj_func = Objective(data, labels)

    record_path = './records/' + dataset + '/' + optim_name
    if need_record:
        writer = SummaryWriter(record_path)

    if optim_name == 'GD':
        optimizer = GD(obj_func, need_record)
    elif optim_name == 'BFGS':
        optimizer = BFGS(obj_func, need_record)
    elif optim_name == 'GreedyBFGS':
        optimizer = GreedyBFGS(obj_func, need_record)
    elif optim_name == 'SharpenedBFGS':
        optimizer = SharpenedBFGS(obj_func, need_record)
    else:
        raise NotImplementedError

    for iteration in range(max_iter):
        loss = optimizer.step()

        if iteration % 10 == 0:
            lambda_k = optimizer.newton_decrement()
            print("Iteration {}: loss = {:.6f}, lambda = {:.3e}, ratio = {:.3e}".format(
```

```
228                 iteration, loss, lambda_k, lambda_k / optimizer.lambda_0))
```

Reference

[1] BENNETT A A. Newton's method in general analysis[J]. Proceedings of the National Academy of Sciences of the United States of America, 1916, 2(10):592.

[2] BOYD S, VANDENBERGHE L. Convex optimization[M]. New York, NY, USA: Cambridge University Press, 2004.

[3] BROYDEN C G, JR. J E D, BROYDEN, et al. On the local and superlinear convergence of quasi-Newton methods[J]. IMA J. Appl. Math, 1973, 12(3):223-245.

[4] BROYDEN C G. A class of methods for solving nonlinear simultaneous equations [J]. Mathematics of computation, 1965, 19(92):577-593.

[5] BROYDEN C G. The convergence of single-rank quasi-Newton methods[J]. Mathematics of Computation, 1970, 24(110):365-382.

[6] CONN A R, GOULD N I M, TOINT P L. Convergence of quasi-Newton matrices generated by the symmetric rank one update[J]. Mathematical programming, 1991, 50(1-3):177-195.

[7] CONN A R, GOULD N I, TOINT P L. Trust region methods: volume 1[M]. Siam, 2000.

[8] DAVIDON W. Variable metric method for minimization.[R]. Argonne National Lab., Lemont, Ill., 1959.

[9] FLETCHER R. A new approach to variable metric algorithms[J]. The computer journal, 1970, 13(3):317-322.

[10] FLETCHER R, POWELL M J. A rapidly convergent descent method for minimization[J]. The computer journal, 1963, 6(2):163-168.

[11] GAY D M. Some convergence properties of Broyden's method[J]. SIAM Journal on Numerical Analysis, 1979, 16(4):623-630.

[12] GOLDFARB D. A family of variable-metric methods derived by variational means [J]. Mathematics of computation, 1970, 24(109):23-26.

[13] JIN Q, MOKHTARI A. Non-asymptotic superlinear convergence of standard quasi-newton methods[J]. arXiv preprint arXiv:2003.13607, 2020.

[14] LIN D, YE H, ZHANG Z. Explicit superlinear convergence of broyden's method in nonlinear equations[J]. arXiv preprint arXiv:2109.01974, 2021.

[15] LIN D, YE H, ZHANG Z. Greedy and random quasi-newton methods with faster explicit superlinear convergence[J]. Advances in Neural Information Processing Systems 34, 2021.

[16] LIU D C, NOCEDAL J. On the limited memory BFGS method for large scale optimization[J]. Mathematical programming, 1989, 45(1):503-528.

[17] NESTEROV Y, POLYAK B T. Cubic regularization of Newton method and its global performance[J]. Mathematical Programming, 2006, 108(1):177-205.

[18] NOCEDAL J. Updating quasi-Newton matrices with limited storage[J]. Mathematics of computation, 1980, 35(151):773-782.

[19] ORTEGA J M, RHEINBOLDT W C. Iterative solution of nonlinear equations in several variables: volume 30[M]. Siam, 1970.

[20] RODOMANOV A, NESTEROV Y. Greedy quasi-newton methods with explicit superlinear convergence[J]. SIAM Journal on Optimization, 2021, 31(1):785-811.

[21] RODOMANOV A, NESTEROV Y. Rates of superlinear convergence for classical quasi-newton methods[J]. Mathematical Programming, 2021:1-32.

[22] RODOMANOV A, NESTEROV Y. New results on superlinear convergence of classical quasi-newton methods[J]. Journal of Optimization Theory and Applications, 2021, 188(3):744-769.

[23] SHANNO D F. Conditioning of quasi-Newton methods for function minimization [J]. Mathematics of computation, 1970, 24(111):647-656.

[24] YE H, LIN D, ZHANG Z, et al. Explicit superlinear convergence rates of the sr1 algorithm[J]. arXiv preprintarXiv:2105.07162, 2021.