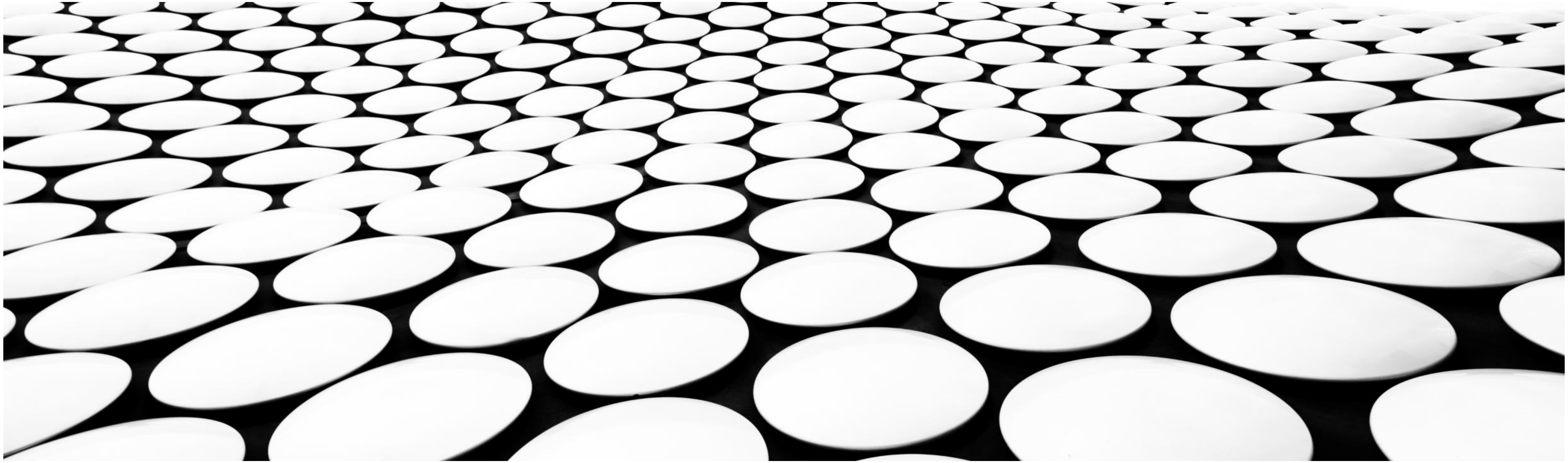
分布式计算

邱怡轩



今天的主题

- L-BFGS 优化算法
- 改进实现细节

Logistic 回归

- 假定 $Y|x \sim Bernoulli(\rho(\beta' x))$
- $\rho(x) = 1/(1 + e^{-x})$，即 Sigmoid 函数
- $\rho(\beta' x)$ 代表 Y 取1的概率

- 给定数据 (y_i, x_i)，$i = 1, \ldots, n$
- 估计 β

目标函数

- 利用极大似然准则

$$L(\beta) = -\sum_{i=1}^{n}\{y_i \log \rho_i + (1-y_i)\log(1-\rho_i)\}$$

- 其中 $\rho_i = \rho(x_i'\beta)$

- 找到一个 β 的取值，使得 $L(\beta)$ 最小

方法对比

- 梯度下降法
  - 计算简单，只需求一阶导数
  - 适合高维问题，存储 $p \times 1$ 向量
  - 收敛较慢
  - 需人工设置步长
- 牛顿法
  - 需要求二阶导数（Hessian 矩阵）
  - 存储 $p \times p$ 矩阵，计算 $O(p^3)$
  - 通常收敛很快
  - 自适应步长

方法对比

- 是否可以结合梯度下降法与牛顿法的优点?

L-BFGS 优化算法

L-BFGS

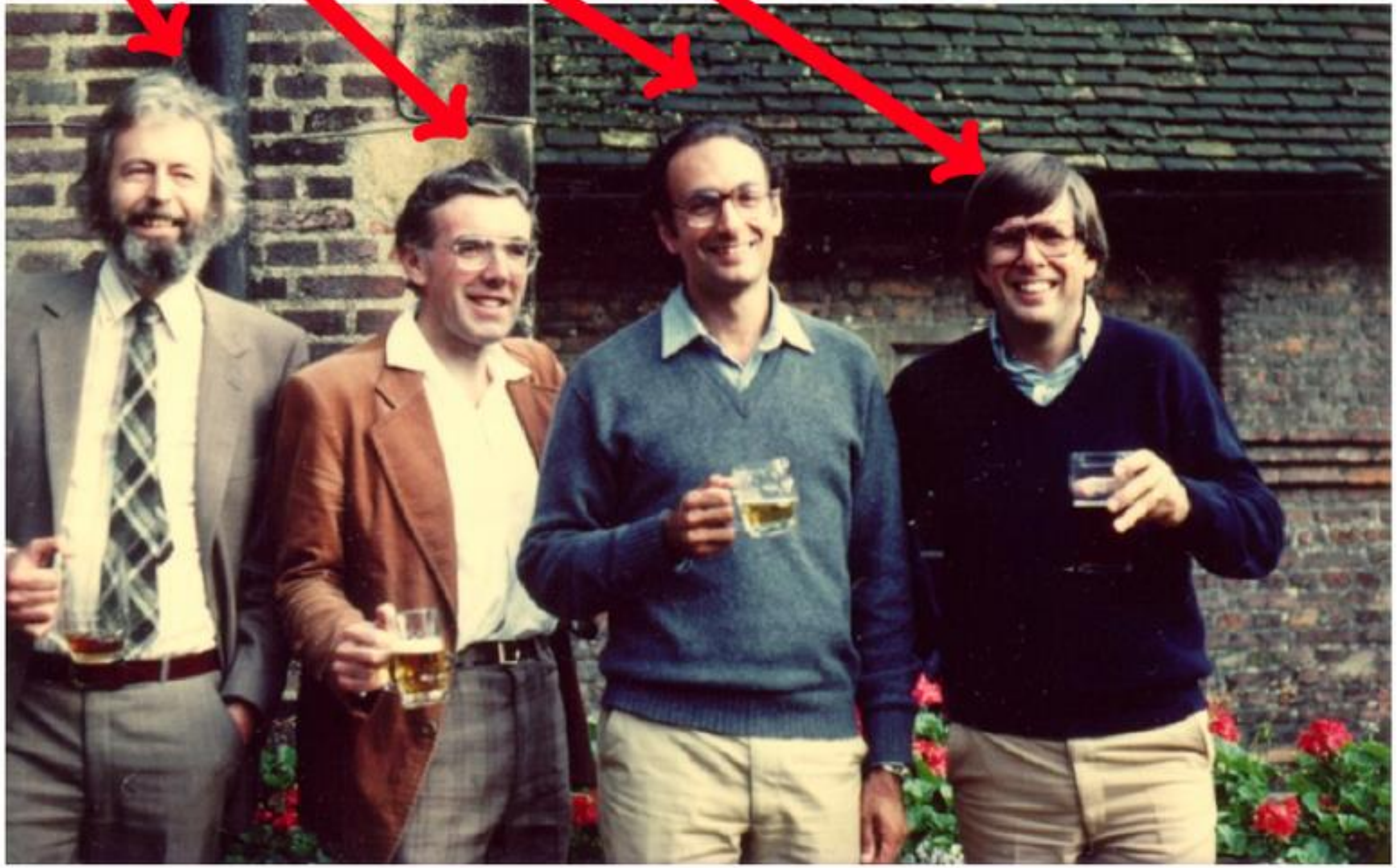
- L-BFGS 算法的全称是
- Limited-memory Broyden–Fletcher–Goldfarb–Shanno algorithm
- 是一种改进后的一阶算法

L-BFGS

- L-BFGS 是对 BFGS 算法的改进
- BFGS 由四位优化算法数学家在1970年独立提出
- L-BFGS 解决了 BFGS 的内存占用问题
- L 代表 Limited-memory

L-BFGS

Broyden, Fletcher, Goldfarb, Shanno



L-BFGS

- 具体原理可以参考其论文
- Liu, D. C., & Nocedal, J. (1989). On the limited memory BFGS method for large scale optimization. *Mathematical programming*, *45*(1), 503-528.

L-BFGS

- L-BFGS 是一个比较复杂的算法
- 但只需要“用户”提供函数值和梯度
- L-BFGS 利用一阶导数（梯度）的信息来近似二阶导数（Hessian 矩阵）
- 因此是一种拟牛顿法
- 在 Python 中由 `scipy.optimize.minimize()` 实现

实现

- `lec10-logistic-lbfgs.ipynb`

算法优点

- 结合了梯度下降和牛顿法各自的优点
- 只需推导和计算梯度，无需二阶导数
- 内存消耗不大
- 收敛速度较快
- 是当前解光滑优化问题的标准算法之一

改进实现细节

实现细节

- 数值稳定算法
- 缓存机制
- 混洗（Shuffling）机制

数值稳定算法

- 在 Logistic 回归中，我们需要计算

$$\rho(x) = \frac{1}{1 + e^{-x}}$$

- 为了计算目标函数还需要 $\log \rho(x)$ 和 $\log(1 - \rho(x))$

数值稳定算法

- $\rho(x) = \frac{1}{1+e^{-x}}$
- 问题1：当 x 很小的负值时，$e^{-x} \rightarrow +\infty$，造成 $\rho(x)$ 计算不稳定
- 问题2：当 $\rho(x)$ 接近于0或1时，$\log \rho(x)$ 或 $\log(1-\rho(x))$ 会出现 NaN

问题1

- 对于问题1，一种解决方法是对 x 的取值分类讨论

$$x \geq 0, \text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

$$x < 0, \text{sigmoid}(x) = \frac{e^{x}}{1 + e^{x}}$$

- 此时对于任意的 x，分子与分母都是稳定的取值

问题1

- scipy.special.expit() 函数进行了这样的处理
- 也可以很方便地手动实现

```
In [4]: import numpy as np
        def sigmoid(x):
            x = np.array(x)
            e = np.exp(-np.abs(x))
            numer = np.where(x >= 0.0, 1.0, e)
            denom = 1.0 + e
            return numer / denom

        sigmoid([-1000, -100, -10, 0, 10, 100, 1000])

Out[4]: array([0.00000000e+00, 3.72007598e-44, 4.53978687e-05, 5.00000000e-01,
               9.99954602e-01, 1.00000000e+00, 1.00000000e+00])
```

问题2

- 对于问题2，一种简单粗暴的方法是在log()函数中加上一个很小的正数
- 但还有一种更好的方式

问题2

- 可以先推导出 $\log\rho(x)$ 的形式

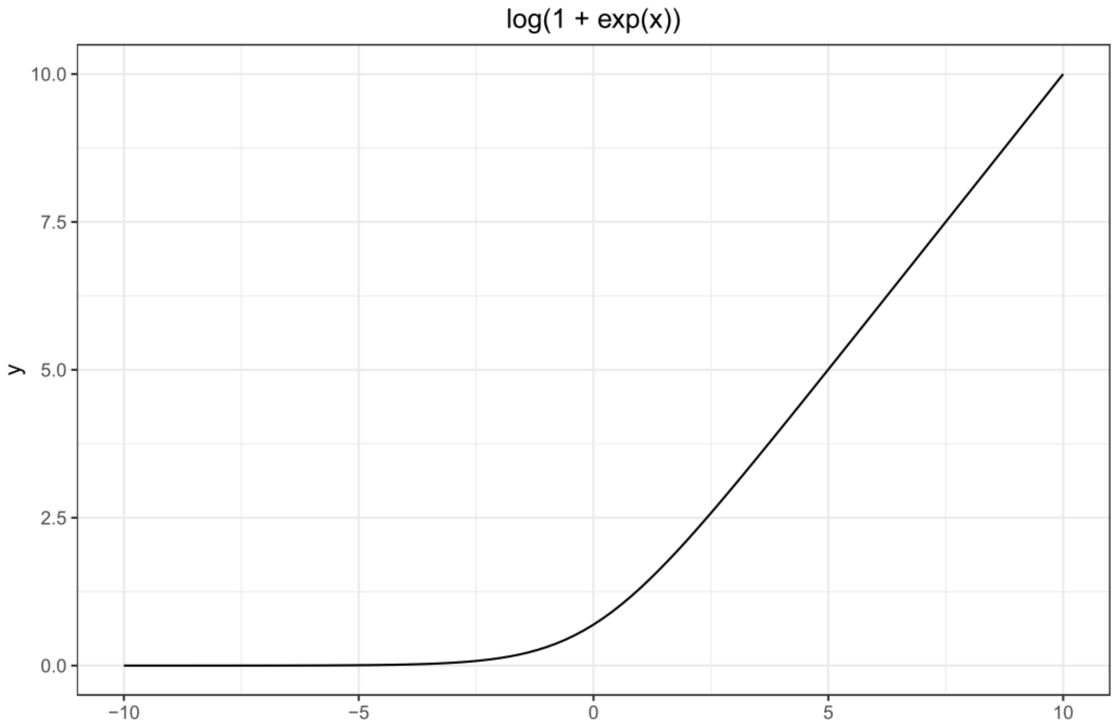
$$\log\rho(x) = x - \log(1+e^x)$$

$$\log\big(1-\rho(x)\big) = -\log(1+e^x)$$

- 核心在于计算函数 $s(x) = \log(1+e^x)$

问题2

- 事实上，$s(x)=\log(1+e^x)$ 是一个数值稳定的函数
- 但需要用特殊的计算方法



问题2

- 如果直接计算，那么 x 很大时 exp(x) 将会溢出
- 但是可以发现
  $s(x) = \log(1+e^x) = x + \log(1+e^{-x})$
  在 x 很大时与 x 是同一量级
- 因此可以分类讨论
  $$s(x) = \begin{cases} \log(1+e^x), & x < 0 \\ x + \log(1+e^{-x}), & x \geq 0 \end{cases}$$
- 思考：如何用 Numpy 进行向量化实现

缓存机制

- 在利用 Spark 对 RDD 进行操作时，通常会叠加很多次变换（map、filter 等等）
- 理论上每次取数据时都要重复整个流程
- 为了避免重复操作，可以将中间的某些结果进行缓存
- 即将变换后的计算结果存进内存
- 下次需要数据时直接从内存调取

缓存机制

- 缓存是 Spark 框架非常重要的一个机制
- 在内存允许的情况下可以显著改善计算效率
- 使用方法非常简单，`rdd.cache()`
- 但也需要注意内存的使用情况
- 必要时需要配置 PySpark，增加内存使用上限

实现

- lec10-implementation-details.ipynb

混洗

- 在回归的例子中我们发现
- MapPartitions 之后有时数据没有按原始的顺序排列
- 这是因为划分分区，即调用 `repartition()` 时触发了数据混洗（shuffling）机制

混洗

- 一方面，混洗会使数据重新进行划分，增加通信成本，从而降低运算效率
- 另一方面，要增加分区数目就必须进行混洗
- 实际使用中需要一些权衡

混洗

- 如果只需减少分区数目，可以使用 `coalesce()` 函数，避免混洗操作
- 如果一定需要增加分区数，则混洗不可避免
- 此时需要注意计算结果的顺序

排列不变性

- 许多模型和问题具有排列不变性
- 即数据的行打乱后，最终的计算结果不变

- 例：回归系数，似然函数
- 反例：观测的预测值

保留顺序

- 如果既需要增加分区数，又需要保持数据的顺序，可以在原始数据中加入索引信息
- 然后在计算结果时将分区的顺序一并返回
- 最后按索引排序
- 参见 `lec10-shuffling.ipynb`