机器学习（L04）：线性回归

授课教师：王贝伦 / 助教：张嘉琦，周嘉莹，黄旭

1 梯度下降法

我们在上一章介绍了线性回归的基本概念和其正规方程解法。其损失函数为

$$\begin{aligned} J(\theta) &= \frac{1}{2}\sum_{i=1}^{n}(x_i^T\theta - y_i)^2 \\ &= \frac{1}{2}(X\theta - y)^T(X\theta - y) \end{aligned} \tag{1}$$

使用正规方程法，我们可以求得当损失函数取最小值时，

$$\theta^* = (X^TX)^{-1}X^Ty \tag{2}$$

虽然可以直接求得最优解，正规方程法却有限制条件，那就是方阵 X^TX 必须可逆才能求得。此外，计算 X^TX 的逆矩阵的代价也较大，计算时间复杂度为 $O(n^3)$，对于特征数量 n 较大的模型计算逆矩阵耗时较多。因此我们这里选择一种使用场景更多的优化方法——梯度下降法。

梯度下降法（Gradient descent）是一个一阶最优化算法。要使用梯度下降法找到一个函数的极小值，必须向函数上当前点对应梯度的反方向的规定步长距离点进行迭代搜索。

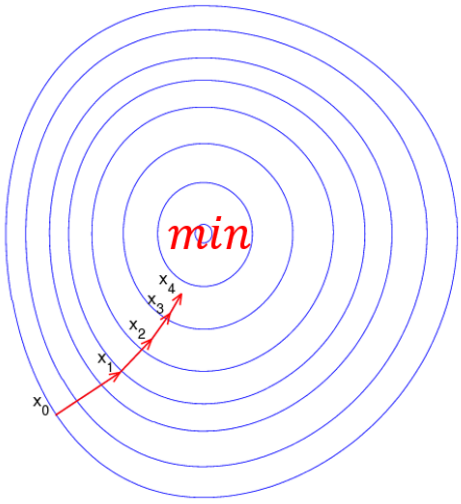

Figure 1: 梯度下降法示意图

梯度下降法基于以下观察：如果实函数 $F(x)$ 在点 a 处可微且有定义，那么函数 $F(x)$ 在 a 点沿着梯度相反的方向 $-\nabla F(a)$ 下降最多。初始化 $k=0$ 和 x_0（随机生成或手动设置），当 $k<k_{max}$ 时进行迭代：

$$x_k = x_{k-1} - \alpha\nabla_x F(x_{k-1}) \tag{3}$$

其中 α 称为学习率（learning rate），注意每次迭代 α 的值是可以改变的。

例 2. 我们使用梯度下降法来函数 $F(x)=x^2-3$ 取最小值时 x 的值，$\nabla_x F(x) = F'(x) = 2x$。假设 $x_0=3$，$\alpha=0.6$，那么我们可以得到如下迭代过程：

(1) $x_1 = x_0 - \alpha\cdot 2x_0 = 3 - 0.6\times 2\times 3 = -0.6$

(2) $x_2 = x_1 - \alpha\cdot 2x_1 = -0.6 - 0.6\times 2\times(-0.6) = 0.12$

(3) $x_3 = x_2 - \alpha\cdot 2x_2 = 0.12 - 0.6\times 2\times 0.12 = -0.024$

(4) $\cdots\cdots$

最终我们可以得到一个无限接近于 0 的值，在一定精度范围内即可认为是最优解。

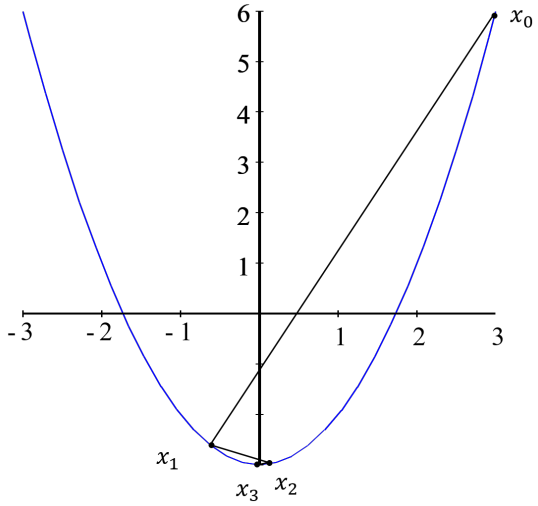

Figure 2: 迭代过程示意图

使用梯度下降法时，有三个注意点：

(1) 学习率

不同的学习率将决定训练结果的质量。如果学习率过小，那么收敛过程会很慢，如果学习率过大，那么可能会导致不收敛，甚至远离。还是以 Figure 2 为例，若初始值设为 $x_0=1$，$\alpha=1.5$，那么计算得 $x_1=-2$，这样子反而 $F(x_1)>F(x_0)$ 了，与我们训练的目标相悖。

(2) 初始点

初始点的选取也将影响结果。初始点选取的不好则容易陷入局部最优解，选取的好则能很快找到全局最优解。

(3) 目标函数

目标函数的性质也很重要。例如有的目标函数的最小值附近梯度很小，比如 $f(x)=\frac{1}{100}x^2$，会导致收敛的速度非常慢。有的目标函数甚至不可导，无法运用梯度下降法。另外如果函数在一个维度上很“平缓”，而在另一个维度上很“陡峭”，那么计算得到的梯度就不平衡，也会导致梯度下降失效（如图3所示）。

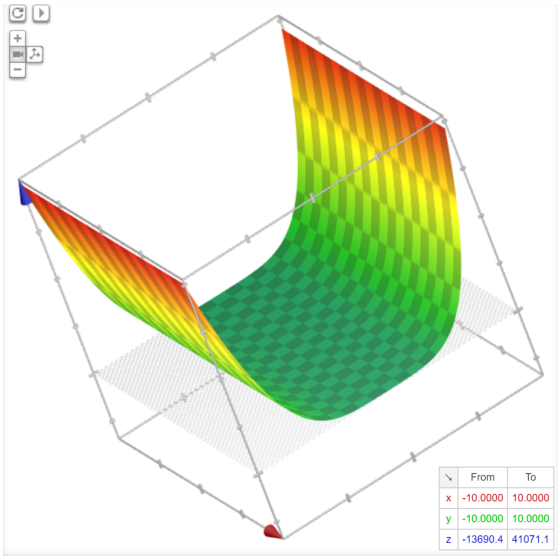

Figure 3: 使梯度下降失效的目标函数。

对于一个复杂点的函数可能有多个局部极小值，为了避免陷入局部最优解，可以再随机初始化，从另一位置重启迭代算法。

3 线性回归中的梯度下降

对于线性回归中的损失函数

$$\begin{aligned} J(\theta) &= \frac{1}{2}(X\theta - y)^T(X\theta - y) \\ &= \frac{1}{2}(\theta^T X^T X\theta - \theta^T X^T y - y^T X\theta + y^T y) \end{aligned} \tag{4}$$

其梯度为

$$\nabla_\theta J(\theta) = X^T X\theta - X^T y \tag{5}$$

因此，使用梯度下降法的迭代方程为

$$\begin{aligned} \theta^{t+1} &= \theta^t - \alpha\nabla_\theta J(\theta^t) \\ &= \theta^t + \alpha X^T(y - X\theta^t) \end{aligned} \tag{6}$$

其中 θ^t 表示经过第 t 轮迭代得到的 θ 值。由于每一次迭代，θ 都会根据所有样本的值进行更新，所以这种方法也被称为批次梯度下降法。假设有 n 个样本，p 个特征，那么每轮迭代的时间复杂度为 $O(np)$，若经过 l 轮达到收敛，则总的复杂度为 $O(npl)$。

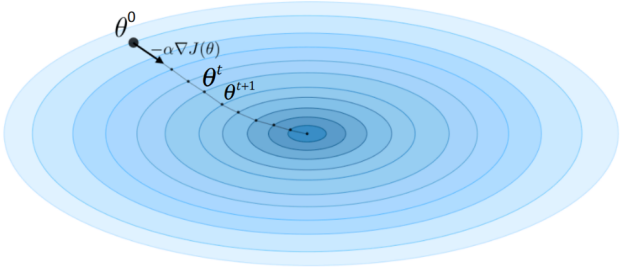

Figure 4: 线性回归中的梯度下降示意图

如何选择学习率 α 的值是一个重要的问题。如果 α 取值太小，那么靠近局部最小值的速度会很慢；如果 α 取值太大，那么可能难以足够接近局部最小值。所以在实际运用中，需要依靠经验或者实验来选取 α 的值。

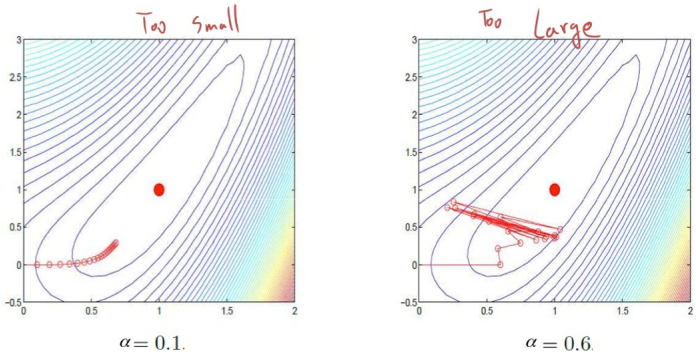

Figure 5: α 取值的影响

Algorithm 1: 梯度下降算法 (GD)

1 **输入：样本矩阵** $x \in \mathbb{R}^{n\times p}$**，样本对应的标签** $y \in \mathbb{R}^n$**，最大迭代次数** T **和学习率** $\alpha > 0$**。随机初始化系数向量** θ**;**
2 **for** $i \leftarrow 1$ *to* T **do**
3 $\quad\theta \leftarrow \theta - \alpha x^T(x\theta - y)$ **;**
4 $\quad$**if** 满足停机准则 **then**
5 $\quad\quad$**break;**
6 $\quad$**end**
7 **end**
8 **输出：预测系数** θ。

4 随机梯度下降 (Stochastic Gradient Descent)

之前介绍的梯度下降算法（GD）最大的问题是**运算负担很重**，因为梯度的计算需要用到正个训练数据集。对于大规模的数据集，运算开销会非常大。因此，为了快速解决线性回归问题，研究者们提出了随机梯度下降（SGD）算法。

4.1 随机梯度下降的形式

回顾线性回归，我们需要最小化一个损失函数：

$$J(\theta) = \sum_{i=1}^{n} J_i(\theta) \quad \text{其中} \quad J_i(\theta) = \frac{1}{2}(y_i - x_i^\top\theta)^2 \tag{7}$$

之前的批量梯度下降 (Batch Gradient Descent) 中，我们将 $J(\theta)$ 作为一个整体，每一次迭代时计算这个整体的梯度 $\nabla_\theta J(\theta) = (y - \theta^\top x)x$。显然，当样本数量 n 较大的时候，计算梯度的时间开销会很大。因此，在随机梯度下降中，我们每一次迭代更新 θ，都只考虑其中一个样本对应的损失函数 $J_i(\theta)$。通过这种方法，我们用 $\nabla_\theta J_i(\theta) = (y_i - x_i^\top\theta)x_i$ 来近似 $\nabla_\theta J(\theta)$。随机梯度下降算法与批量梯度下降算法很相似，算法伪代码如算法 2 所示。在每一次迭代中，算法随机遍历每一个样本，计算对应的梯度 $\nabla_\theta J_i(\theta)$ 以更

新系数 θ。

Algorithm 2: 随机梯度下降算法 (SGD)

输入：**样本矩阵** $x \in \mathbb{R}^{n\times p}$，**样本对应的标签** $y \in \mathbb{R}^n$，**最大迭代次数** T **和学习率** $\alpha > 0$。**随机初始化系数向量** w;
for $t \leftarrow 1$ *to* T **do**
  随机打乱 n **个样本的顺序;**
  for $i \leftarrow 1$ *to* n **do**
    $\theta \leftarrow \theta - \alpha(y_i - x_i^\top \theta)x_i$;
  end
  if 满足停机准则 **then**
    break;
  end
end
输出：**预测系数** θ。

图6体现了批量梯度下降与随机梯度下降的梯度更新的区别。显然随机梯度下降的曲线波动更大，因为在每一步只考虑了一个样本，然而整体的趋势与批量梯度下降相同。除此之外，由于随机梯度下降每一次更新的幅度较大，算法比较容易避开局部最优值，所以随机梯度下降最终的预测结果可能会更准确。

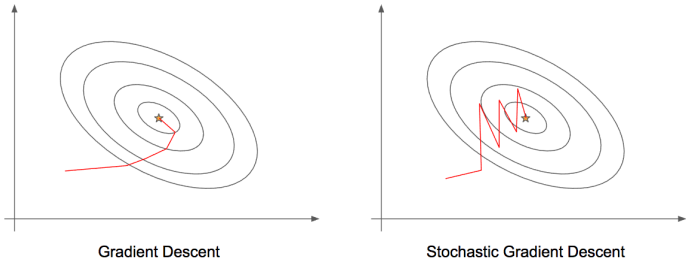

Figure 6: 批量梯度下降和随机梯度下降的对比。

4.2 停机准则 (Stop Criteria)

在实际应用中，如下的几种停机准则是比较常用的：

(1) 最大迭代数：对于算法设置一个最大的迭代次数（如算法2中的 T），当算法的迭代代数达到这个最大值的时候，算法会停止。

(2) 改进低于阈值：提前设定一个阈值 γ，当算法中在新的一次迭代后，系数与前一代相比的改进低于这个阈值时，算法就会提前停止。如何计算系数改进程度需要按照实际要求来选择。比如，对于系数 θ，用 $||\theta^t - \theta^{t-1}||_2 \leq \gamma$ 来判断是否提前停止算法。

(3) 损失函数值：利用损失函数值来作为停机准则。因为我们的目的是最小化损失函数值，那么当损失函数值很接近于 0 的时候，算法就可以停止了。比如针对第 t 代更新后的系数 θ^t，如果 $J(\theta^t) \leq 10^{-5}$ 就停止算法。

上述的三种停机准则不只用于线性回归，可以通用于任何迭代的机器学习算法。但是针对具体的模型或算法，需要进行特别的设计。

4.3 小批量梯度下降 (Mini-Batch GD)

随机梯度下降存在的一个问题是不稳定（图6）。这是因为我们每次更新系数时只考虑一个样本，这种“短视”的做法会让我们的算法过分关注一个样本，而不是考虑多个样本的平均效应。最原始的批量梯度下降从整体上考虑了所有数据的信息，然而它的运算效率比较低。因此为了在运算效率和算法稳定性之间作出权衡，我们可以使用小批量梯度下降。

我们将数据分为一共 B 个块 $\{x^{(1)}, x^{(2),\cdots,x^{(L)}}\}$，其中每一块的样本数量为 $n_1, n_2, \cdots, n_L$。然后在更新系数时综合考虑一个块内的数据。这时损失函数可以表示为

$$J(\theta) = \sum_{l=1}^{L} \sum_{i=1}^{n_l} J_i(\theta) \tag{8}$$

这样一来，在迭代中的每一代，算法遍历每一个数据块 $x^{(l)} \in \mathbb{R}^{n_l \times p}$，利用这一块数据来计算梯度。这样系数的更新公式就变为了

$$\theta \leftarrow \theta - \alpha\ (y^{(l)} - x^{(l)\top}\theta)x^{(l)} \tag{9}$$

其中 $y^{(l)}$ 是样本块 $x^{(l)}$ 中样本对应的标签。小批量梯度下降算法的伪代码如算法3所示。

Algorithm 3: 小批量梯度下降算法 (Mini-Batch SGD)

输入：**样本矩阵** $x \in \mathbb{R}^{n\times p}$，**样本对应的标签** $y \in \mathbb{R}^n$，**最大迭代次数** T **和学习率** $\alpha > 0$。**随机初始化系数向量** w;
for $t \leftarrow 1$ *to* T **do**
  for $l \leftarrow 1$ *to* L **do**
    $\theta \leftarrow \theta - \alpha\ (y^{(l)} - x^{(l)\top}\theta)x^{(l)}$;
  end
  if 满足停机准则 **then**
    break;
  end
end
输出：**预测系数** θ。

一般来说，在实际使用小批量梯度下降时，我们选择使得每一块的样本数都相同，即 $n_1 = n_2 = \cdots = n_L = B$。特别地，当 $n = 1$ 时就是随机梯度下降算法。数据块的大小 B 是实现小批量梯度下降时所需要调整的参数。如果 B 较小，那么算法会收敛很快，但是不稳定；反之如果 B 较大，那么算法的收敛会比较慢但是会更稳定。因此 B 的选择是在算法效率和稳定性之间的权衡。为了提高 CPU 或者 GPU 运算的效率，块大小 B 经常设置为 2 的次方，比如 32,64,128$\cdots\cdots$。

4.4 复杂度分析

梯度下降的计算主要集中在更新 θ 时的梯度计算。先考虑随机梯度下降算法。对于随机梯度下降，梯度的计算只需要向量乘法运算，因此每一次更新 θ 只需要 $O(p)$ 的时间复杂度。因此随机梯度下降算法总体的时间复杂度为 $O(Tnp)$

然后考虑小批量梯度下降这种一般情况。针对每一块样本 $x^{(l)} \in \mathbb{R}^{B\times p}$，计算梯度所需的是 B 乘法运算。这一步所需的时间复杂度为 $O(Bp^2)$。在每一次迭代中要针对总共 L 个样本块进行上述的系数更新。因此小批量梯度下降算法的时间复杂度为 $O(TLBp^2)$。因为 $LB = n$，小批量梯度下降和批量梯度下降算法的时间复杂度其实是相同的。但是如果我们把更新一次 θ 当做算法的一步，那么小批量梯度下降算法在每一步中的时间成本要少于批量梯度下降。这种特性使得小批量梯度下降更适用于样本数量不断变化的场景，比如在线学习（下一节中会详细解释）。

5 在线学习 (Online Learning)

之前我们考虑的情况都是以获得了全部样本为前提的：利用获得的所有样本 x 来预测系数 θ。然而有些应用中，我们可能暂时无法一次性获得所有样本，或者样本数量是无限的。比如说对于股票预测，每一天都会有新的数据产生而且理论上来说数据量是无限的。这种情况下数据往往以数据流的形式存在，因此当每次获得新的样本时，我们需要对模型参数（比如线性回归中的 θ）进行调整。对于在线学习，批量梯度下降并不是一个好的选择，因为每当有新的数据输入，我们就需要将新的数据合并入现有的数据集，并用合并后的数据集来重新计算梯度以更新系数。随着时间的增长，数据集会越变越大，每次有新的数据输入时，更新系数的时间成本会非常大。相比较而言，小批量梯度下降更适合在线学习。针对一个提前设定的数据块大小 B，如果新加入的数据数量达到 B，我们就用这些数据来更新系数。因此，根据数据流来每次调整系数只需要用到一小部分数据，而不是现有的整个数据集。

6 梯度下降收敛性分析

梯度下降算法在每一次迭代中针对损失函数 $J(\theta)$ 求梯度并更新 θ 来最小化 $J(\theta)$。对于线性回归，我们通常选择 $J(\theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - x_i^\top\theta)^2$ 作为损失函数。然而梯度下降并不只适用于这一种损失函数。下面我们会证明对于一个任意的满足简单约束的函数 $J(\theta)$，梯度下降算法都会令这个函数收敛。

假设对于函数 $J(\theta)$，存在一个 θ^* 使得 $J(\theta^*) \leq J(\theta)$，即 θ^* 能使得函数取到最小值。假设函数 $J(\theta)$ 是凸函数且可导，同时假设它利普希茨连续（Lipschitz Continuous）。我们先来解释相关的几个约束条件。

定义：凸函数

假设对于一个凸集 $\mathbf{X}$，有一个函数 $J: \mathbf{X} \mapsto \mathbb{R}$，那么我们称 J 是凸函数如果

$$\forall x, y \in \mathbf{X},\ \forall t \in [0,1]: \quad J(tx+(1-t)y) \leq tJ(x)+(1-t)J(y) \tag{10}$$

上面这个式子表示对于横坐标上的任意两个点，它们平均值的函数值不大于它们函数值的平均值。从几何的角度来讲，如果一个函数是凸函数，那么这个函数上任意两个点之间的连线一定在函数上部包围的空间之内。

图7是一个凸函数例子，图中显然函数曲线上任意两点的连线在函数的内部。

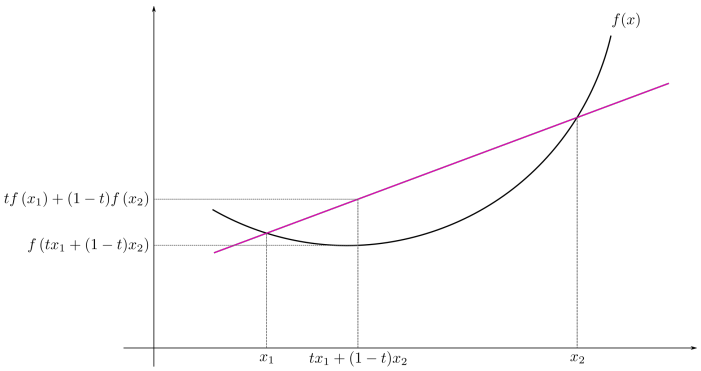

Figure 7: 凸函数示例

凸函数并非都可导。比如对于函数 $J(x) = |x|$ 来说，它在 $x = 0$ 处不可导。因此为了在梯度下降算法中能够计算梯度值，我们需要假设损失函数可导。然而的确会出现选择的损失函数不可导的情况，这种情况我们需要计算次梯度 (subgradient) 或者利用近端算子 (proximal operator)。这部分在之后的课程中会有详细的解释。

定义：利普希茨连续

对于一个函数 $J: \mathbb{R}^p \mapsto \mathbb{R}$，我们称它是利普希茨连续的如果

$$\forall x, y \quad ||J(x) - J(y)||_2 \leq K||x - y||_2 \tag{11}$$

其中 $K > 0$ 是利普希茨常数。

利普希茨连续的条件可以转化为 $\frac{||\nabla J(x) - \nabla J(y)||_2}{||x - y||_2} \leq K$。也就是说函数上两点连线的斜率有一个上界：利普希茨常数 K。这个常数限制了函数 J 的“陡峭”程度。

下面我们就可以来证明梯度下降的收敛性质。

定理 6.1. 假设损失函数 $J: \mathbb{R}^p \mapsto \mathbb{R}$ 是一个可导的凸函数，并且它的梯度是利普希茨连续的，即 $||\nabla J(\theta_1) - \nabla J(\theta_2)||_2 \leq K||\theta_1 - \theta_2||_2$ 对于任何 $\theta_1, \theta_2 \in \mathbb{R}^p$ 成立。如果设置一个固定不变的学习率 $\alpha \leq \frac{1}{K}$，那么在梯度下降算法中经过 t 代的迭代之后，损失函数值满足

$$J(\theta^t) - J(\theta^*) \leq \frac{||\theta^0 - \theta^*||_2^2}{2t\alpha} \tag{12}$$

其中 θ^* 是系数在解空间的最优值，使得损失函数取得最小值。

定理6.1的具体证明参见引用中的参考资料 [9]。公式 (12) 说明对于满足条件的损失函数，梯度下降一定会收敛，并且收敛速率为 $O(\frac{1}{t})$。这说明如果设置一个阈值 ε，那么为了达到 $J(\theta^t) - J(\theta^*) \leq \varepsilon$，我们需要经过 $t = O(\frac{1}{\varepsilon})$ 次迭代。

7 牛顿法（Newton's Method）

在凸优化中，牛顿法用迭代和不断逼近的思想来求解一个比较复杂的方程的根。牛顿法的核心思想是对函数的一阶泰勒展开求解。假设我们有一个函数 $f(x)$，我们需要求解 $f(x) = 0$。我们在点 x_0 处将函数进行一阶展开，得到

$$f(x) = f(x_0) + \nabla f(x)(x - x_0) \tag{13}$$

将 $f(x) = 0$ 代入可得

$$x = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{14}$$

我们这里计算得到的 x 只是对方程根的近似，但是 x 比 x_0 要更接近方程根。因此我们可以通过迭代的方式，在这个近似解 x 处一阶展开然后更新近似解的值。这样我们拥有一个迭代解方程根的公式

$$x^{t+1} = x^t - \frac{f(x^t)}{f'(x^t)} \tag{15}$$

我们已经知道了牛顿法的根本思路：利用迭代方法逼近最优解以解 $f(x) = 0$。回顾线性回归，我们希望预测一个系数 θ 来最小化函数 $J(\theta)$。而损失函数的最小值出现在 $\nabla J(\theta) = 0$ 的位置。所以对于线性回归，我们可以求牛顿法求方程 $\nabla J(\theta)$ 的根。令 $x = \theta$，$f(x) = \nabla J(\theta)$，类比公式 (15)，我们可以得到用牛顿法解线性回归问题的系数迭代公式

$$\theta^{t+1} = \theta^t - [\nabla^2 J(\theta^t)]^{-1} \nabla J(\theta^t) \tag{16}$$

其中

$$\nabla^2 J(\theta) = \begin{bmatrix} \frac{\partial^2 J(\theta)}{\partial \theta_1^2} & \frac{\partial^2 J(\theta)}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 J(\theta)}{\partial \theta_1 \partial \theta_p} \\ \frac{\partial^2 J(\theta)}{\partial \theta_2^2 \partial \theta_1} & \frac{\partial^2 J(\theta)}{\partial \theta_2^2} & \cdots & \frac{\partial^2 J(\theta)}{\partial \theta_2 \partial \theta_p} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 J(\theta)}{\partial \theta_p \partial \theta_1} & \frac{\partial^2 J(\theta)}{\partial \theta_p \partial \theta_2} & \cdots & \frac{\partial^2 J(\theta)}{\partial \theta_p^2} \end{bmatrix} \tag{17}$$

是二阶导数矩阵，也被称为黑塞矩阵（Hessian matrix）。在用牛顿法解线性回归时，我们不仅需要求黑塞矩阵，还要对黑塞矩阵求逆。当数据量很大的时候，运算速度会受到很大影响，因此实际应用中牛顿法并不如梯度下降法常用。

7.1 牛顿法 vs. 梯度下降法

牛顿法和梯度下降都是迭代求解线性回归问题的算法，但它们有很多本质上的不同之处：

- 梯度下降只考虑了一阶导数，而牛顿法考虑了二阶梯度。直观来解释的话，梯度下降在更新时考虑当前位置“坡度”最大的方向，而牛顿法不仅考虑了当前位置的“坡度”，还考虑了走了一步之后“坡度”是否会变的更大。可以说牛顿法比梯度下降**看得更远**。因此一般来说，牛顿法都比梯度下降要收敛得快。

- 梯度下降需要计算梯度，而牛顿法需要计算二阶导数矩阵及其逆。因此牛顿法的时间复杂度更高。并且牛顿法中计算的二阶导数矩阵需要保存在内存中，因此牛顿法的内存复杂度也很高。

- 梯度下降算法中需要设置学习率 α。而牛顿法中没有任何参数，节省了调整参数的时间成本。

8 基于梯度下降的的其他改进算法

为了解决批量梯度下降算法和随机梯度下降算法存在的一些问题，研究者还提出了许多其他基于梯度的算法。

8.1 动量随机梯度下降算法

在之前我们介绍的随机梯度下降和小批量梯度下降中，我们可以看到在每一次迭代时，梯度的下降并不是严格按照全局的最小方向的。虽然总体上来说下降趋势是朝着最小方向的，但是在每一步中的波动比较大，因此可能会导致收敛比较慢。动量随机梯度下降（Stochastic Gradient Descent with Momentum）就是为了减少迭代中的这种波动而提出的。动量随机梯度下降借用了物理中动量的概念：梯度下降的过程中，在某一个点的系数更新方向不仅基于当前这个位置的梯度方向，还会受到之前更新时的梯度方向的影响。如图8所示，一个点的更新方向是在当前点的梯度方向和之前点的梯度方向的共同影响下决定的。

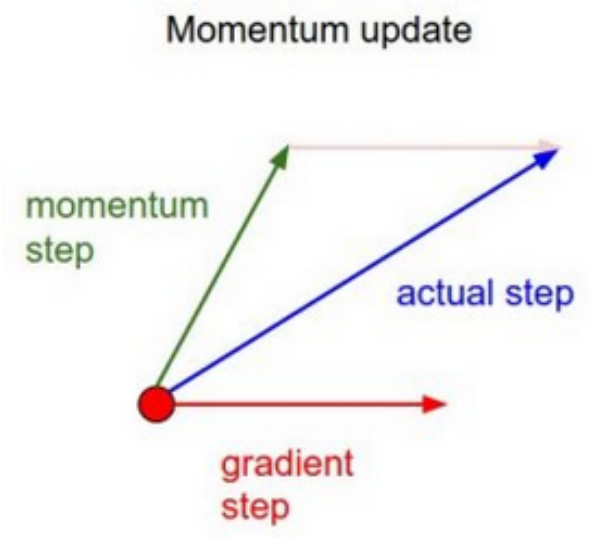

Figure 8: 梯度下降中的动量。

动量梯度下降的具体系数更新公式为

$$\begin{aligned} v^t &= \beta v^{t-1} - (1-\beta)\nabla J_i(\theta^t) \\ \theta^{t+1} &= \theta^t - \alpha v^t \end{aligned} \tag{18}$$

其中 α 是学习率，v^t 就是累计的动量，$\beta \in [0, 1)$ 是用来权衡当前梯度和累计动量的系数。我们来考虑对 v 的更新，v^{t-1} 是之前所累积下来的动量，而 ∇J 是当前位置的梯度方向。在经过用 β 加权平均之后，累积的动量和现在的梯度都被考虑了进去。而且，由于 $\beta < 1$，随着迭代代数增加，早期的一些梯度会在累计动量之中所占的比重越来越少。图9说明动量的存在的确会使梯度下降中的波动减小。

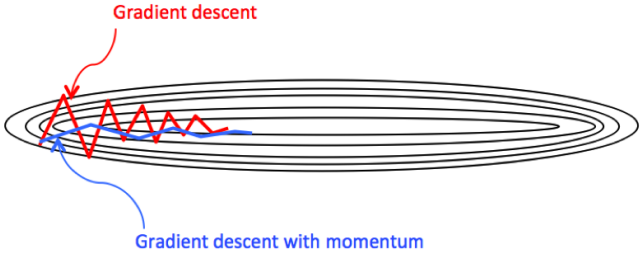

Figure 9: 动量对梯度下降的影响。

8.2 Nestrov 加速梯度下降 (Nestrov Accelerated Gradient Descent)

Nestrov 加速梯度下降是在动量梯度下降的基础上提出的。动量梯度下降存在的问题是，在某一点如果计算出梯度为 0（即满足了最

小化损失函数的目的），由于动量的存在，系数仍然会沿着动量的方向更新。这种情况会导致收敛变慢。举个例子，动量梯度下降就是当一个小球沿着坡滚向山谷的时候，在即将到达山谷的时刻小球并不会减速而是会冲到对面的斜坡上。然后小球可能会在两个斜坡上来回滚动直到最终到达谷底。而 Nestrov 加速梯度下降不仅会考虑当前点的梯度，还会考虑移动之后下一个点的梯度，也就是比动量梯度下降多考虑一步。用那个小球的例子来说就是，小球在快要滚到谷底的时候会预知到下一步将冲到对面的坡上，于是会提前刹车。

Nestrov 加速梯度下降的系数更新公式为

$$\begin{aligned} v^t &= \gamma\theta^{t-1} + \eta\nabla J(\theta - \gamma v^{t-1}) \\ \theta^{t+1} &= \theta^t - v^t \end{aligned} \tag{19}$$

其中从 v^t 的更新我们可以看出 Nestrov 加速梯度考虑了下一步的梯度 $\nabla J(\theta - \gamma v^{t-1})$。

9 解析解 vs. 迭代法

- 标准方程通过计算解析解来解线性规划问题。解析解的优点是实现简单，只需一步就可以完成。但是由于标准方程需要求矩阵的逆，对于大规模的数据，时间成本会很高。另外，如果协方差矩阵 x^Tx 不可逆，那么标准方程就不再适用。
- 迭代法虽然在实现的时候比标准方程复杂，但是时间成本相对较少。另外，迭代法比标准方程更适用于在线学习。但是迭代法最大的问题是并不一定收敛到最优值。

10 编程实现

现在我们一一实现之前介绍的三种解线性方程（标准方程，批量梯度下降，随机梯度下降）的算法。

```
# 标准方程
# theta = (X^TX)^{-1} X^Ty
def normal_equation(X,Y):
    theta = np.linalg.inv(X.T.dot(X)).dot(X.T.dot(Y))
    return theta
```

```
# 批量梯度下降法
def gd(X, Y, theta0, alpha=0.01, max_iters=10000, precision=1e-6):
# 训练样本X,样本标签Y, 初始点theta0,学习率alpha,最大迭代次数max_iters,迭代终止条件precision
    theta = theta0
    f_record = [f(X, Y, theta)]
    for i in range(max_iters):
        current_theta = theta
        theta = theta - alpha * df(X, Y, theta)
        f_record.append(f(X, Y, theta))
        if abs((f(X, Y, current_theta) - f(X, Y, theta))/f(X, Y, theta)) <= precision:
            break
    return theta
```

```
# 随机梯度下降法
def sgd(X, Y, theta0, alpha=0.01, max_iters=10000, precision=1e-8, batch_number=1):
# 训练样本X,样本标签Y, 初始点theta0,学习率alpha,最大迭代次数max_iters,迭代终止条件precision,每批采用样本个数batch_number
    theta = theta0
    f_record = [f(X, Y, theta)]
    for i in range(max_iters):
        batch = np.random.choice(n, batch_number, replace=False)
        current_theta = theta
        theta = theta - alpha * X[batch].T @ (X[batch] @ theta - Y[batch] )
        f_record.append(f(X, Y, theta))
        if abs((f(X, Y, current_theta) - f(X, Y, theta))/f(X, Y, theta)) <= precision:
            break
    return theta
```

我们随机生成一些数据，并分别用三种方法来预测系数，最后计算 MSE。

```
# 测试三种线性回归算法
import numpy as np

# 训练样本数 n; 变量数 p; 测试样本数 m
n = 100
m = 50
p = 5

# 随机生成样本，每一个单独的数据由[0,1]上的均值分布生成。
X_train = np.random.uniform(size = (n, p))
X_test = np.random.uniform(size=(m ,p))
# 随机生成系数，每一个元素由标准正态分布 N(0,1) 产生。
theta = np.random.normal(size=(p,))
# 计算标签，添加一些噪声。
Y_train = X_train @ theta + 0.02*np.random.normal(size=(n,))
Y_test = X_test @ theta + 0.02 * np.random.normal(size=(m,))

theta_nq = normal_equation(X_train, Y_train)
mse_nq = mse(X_test, Y_test, theta_nq)
print("MSE of Normal Equation: ", mse_nq)

theta0 = np.random.normal(size=(p,))
theta_gd = gd(X_train, Y_train, theta0)
mse_gd = mse(X_test, Y_test, theta_gd)
print("MSE of GD: ", mse_gd)

theta_sgd = sgd(X_train, Y_train, theta0)
mse_sgd = mse(X_test, Y_test, theta_sgd)
print("MSE of SGD: ", mse_sgd)
```

其中计算 MSE 的函数 mse 的具体实现代码为

```
# 用MSE来评估预测值的精确度
def mse(X, Y, x):
    prediction = X @ x
    error = np.linalg.norm(prediction - Y)
    MSE = error**2 / Y.size
    return MSE

```

其中一次实验的 MSE 结果对比如表格1所示。图10对比了随着迭代代数的变化，两种算法的损失函数值的变化。显然，随机梯度算法的波动更大一些。

Table 1: 三种方法的 MSE 比较

	正规方程	GD	SGD
MSE	0.000634	0.000634	0.000651

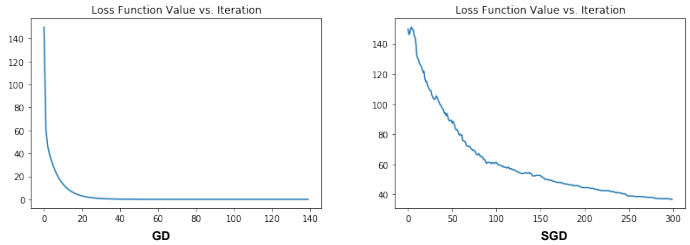

Figure 10: 梯度下降法和随机梯度下降中，损失函数值随迭代代数的变化。

11 数据预处理

最后我们介绍一些数据预处理的方式。在一些机器学习算法中，由于原始数据值的范围差异可能很大，或者因为数据的单位或量纲不同，如果不进行预处理，目标函数将无法正常工作。例如在算欧式距离时，如果其中一个特征的取值范围很大，比如 $[1000, 10000]$，而另一个特征的取值范围很小，比如 $[0, 1]$，那么该距离肯定受第一个特征支配，第二个特征对距离的影响微乎其微，因此特征值需要归一化。另一个方面，特征缩放也可以加快如梯度下降法等算法的收敛速度。本节主要介绍数据的标准化与归一化。

11.1 标准化

标准化使数据中的特征值的均值变为 0，标准差变为 1。这个方法在机器学习中经常使用。它的计算步骤为

$$x' = \frac{x - \bar{x}}{\sigma} \tag{20}$$

其中 x 是原始的特征值向量，x' 是标准化后的向量，$\bar{x}$ 是特征值向量的均值，σ 是它的标准差。

编程实现

```python
from sklearn import preprocessing
import numpy as np
# 将每一列的特征值向量分别标准化
X_train = np.array([
  [1., -1., 2.],
  [2., 0., 0.],
  [0., 1., -1.]
])
X_scaled = preprocessing.scale(X_train)
print(X_scaled)
print(X_scaled.mean(axis=0))
print(X_scaled.std(axis=0))
# Output:
# [[ 0.          -1.22474487  1.33630621]
#  [ 1.22474487  0.          -0.26726124]
#  [-1.22474487  1.22474487 -1.06904497]]
# [0. 0. 0.]
# [1. 1. 1.]
```

11.2 归一化

归一化是将数据应映射到一个范围，通常是 $[-1, 1]$ 或 $[0, 1]$。一种方法是通过最大值和最小值来缩放

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{21}$$

其中 x 是原始值，x' 是归一化后的值。若要映射到 $[a, b]$ 区间，计算公式为

$$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)} \tag{22}$$

另一种方法是通过均值来实现归一化

$$x' = \frac{x - \text{average}(x)}{\max(x) - \min(x)} \tag{23}$$

其中 x 是原始值，x' 是归一化的值。

11.3 编程实现

```python
from sklearn import preprocessing
import numpy as np
# 将每一列的特征值向量分别归一化
# 使用最大值和最小值将特征值映射到[0, 1]
# 若要映射到区间[a, b]，只需设置feature_range=(a, b)
X_train = np.array([
  [1., -1., 2.],
  [2., 0., 0.],
  [0., 1., -1.]
])
X_scaled = preprocessing.minmax_scale(X_train, feature_range
    =(0, 1))
print(X_scaled)
# Output:
# [[0.5         0.          1.        ]
#  [1.          0.5         0.33333333]
#  [0.          1.          0.        ]]

# 使用均值来归一化，没有现成的函数，需手动实现
X_scaled = (X_train - X_train.mean(axis=0)) / (X_train.max(
    axis=0) - X_train.min(axis=0))
print(X_scaled)
# Output:
# [[ 0.          -0.5          0.55555556]
#  [ 0.5          0.          -0.11111111]
#  [-0.5          0.5         -0.44444444]]
```

引用

[1] Masters, D., & Luschi, C. (2018). Revisiting small batch training for deep neural networks. arXiv preprint arXiv:1804.07612.

[2] Online Machine Learning: https://en.wikipedia.org/wiki/Online_machine_learning

[3] Hoi, S. C., Sahoo, D., Lu, J., & Zhao, P. (2018). Online learning: A comprehensive survey. arXiv preprint arXiv:1802.02871.

[4] Lipschitz continuity: https://en.wikipedia.org/wiki/Lipschitz_continuity

[5] Newton's method: https://en.wikipedia.org/wiki/Newton%27s_method

[6] Variants of Gradient Descent: https://suniljangirblog.wordpress.com/2018/12/13/variants-of-gradient-descent/

[7] Stochastic Gradient with Momentum: https://en.wikipedia.org/wiki/Stochastic_gradient_descent#Momentum

[8] An Overview of Gradient Descent Optimization Algorithms: https://ruder.io/optimizing-gradient-descent/index.html#momentum

[9] Convergence Analysis of Gradient Descent: https://www.stat.cmu.edu/~ryantibs/convexopt-F13/scribes/lec6.pdf