

梯度下降

课前准备

- 下载Anaconda软件，请点击[这里](#)进行下载。

本节要点

- 梯度的定义与性质。
- 梯度下降求解损失函数极值。
- 数据标准化对梯度的意义。

梯度

梯度的概念

梯度是一个向量，表示函数在某一点处的方向导数。

$$\nabla f(x_1, x_2, \dots, x_n) = \left(\frac{\partial f(x_1, x_2, \dots, x_n)}{\partial x_1}, \frac{\partial f(x_1, x_2, \dots, x_n)}{\partial x_2}, \dots, \frac{\partial f(x_1, x_2, \dots, x_n)}{\partial x_n} \right)$$

- $f(x_1, x_2, \dots, x_n)$: n 元函数。
- ∇ : 梯度。

由此可知，当 $n = 1$ （ f 为一元函数）时，梯度就是导数。

梯度法求解函数极值

求解示例

简单起见，我们使用一元函数，求解极小值为例。对于函数 $y = x^2$ ，该函数具有唯一的极值点（极小值）。

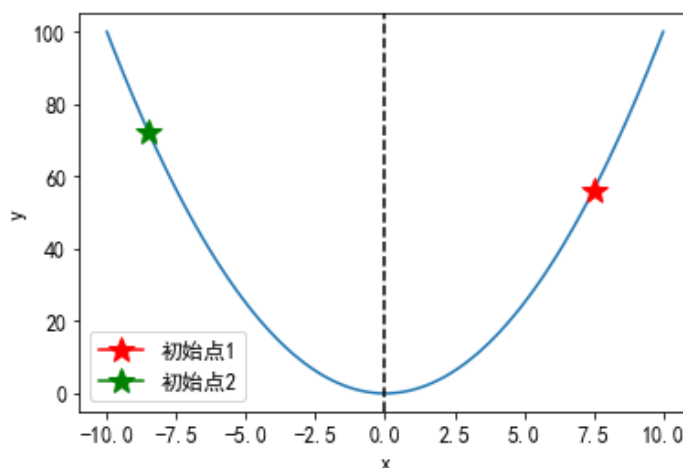
```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 plt.rcParams["font.family"] = "SimHei"
5 plt.rcParams["axes.unicode_minus"] = False
6 plt.rcParams["font.size"] = 12
7
8 def f(x):
9     """定义函数y = f(x)，根据自变量x返回函数值y。
10
11     Parameters
12     -----
13     x : array或者标量
14         自变量。
15
16     Returns
17     -----
18     y : array或者标量
19         根据自变量x，返回x所对应的y值。
20     """
```

```

21
22     return x ** 2
23
24
25 x = np.linspace(-10, 10, 100)
26 plt.plot(x, f(x))
27 # 定义两个初始点。
28 x1, x2 = 7.5, -8.5
29 plt.plot(x1, f(x1), marker="*", ms=15, color="r", label="初始点1")
30 plt.plot(x2, f(x2), marker="*", ms=15, color="g", label="初始点2")
31 plt.axvline(ls="--", c="black")
32 plt.xlabel("x")
33 plt.ylabel("y")
34 plt.legend()

```

1 | <matplotlib.legend.Legend at 0x20502294c88>



求解疑惑

在更新中，我们会遇到如下的问题：

- 初始点的选择往往是随意的，在不同的初始点下，我们该朝哪个方向更新 x 的值，才能确保函数值 y 减小？
- 每次更新的幅度是多少？是固定还是不固定的？

- 随着更新的幅度不同， x 可能会在中心线两侧反复，如果确定后续的更新方向？



梯度的性质

给定函数 $y = f(x_1, x_2, \dots, x_n)$ ，梯度具有这样的性质：在某一点处，自变量 x_1, x_2, \dots, x_n 沿着梯度的方向变化，函数值 y 变化的最快，具体体现为：

- 顺着梯度的方向更新 x ，函数值 y 上升最快。
- 逆着梯度的方向更新 x ，函数值 y 下降最快。

而当函数是一元函数时，梯度就是导数，导数具有如下性质：

1. 导函数大于0 \Leftrightarrow 函数单调递增。
2. 导函数小于0 \Leftrightarrow 函数单调递减。
3. 导函数等于0 \Leftrightarrow 函数取得极值点。

梯度求解极值

因此，我们可以通过梯度指引的方向，进而求解函数 $y = f(x_1, x_2, \dots, x_n)$ 的极值。根据求解极值的不同（极大值还是极小值），我们可以分为**梯度上升**或者**梯度下降**。在机器学习领域中，因为习惯上对损失函数求解最小值，故梯度下降的应用会更多一些。



课堂练习



如果需要求解函数的极大值，是否可以使用梯度下降来实现？

- A 可以
- B 不能
- C 不确定



我们以梯度下降求解函数极小值为例，步骤如下：

1. 设定自变量的初始值（初始点位置） (x_1, x_2, \dots, x_n) 。
2. 求解该点的梯度 (g_1, g_2, \dots, g_n) 。
3. 根据梯度值指引的方向，移动一小段距离，更新点（自变量）的坐标值。

- $x_1 = x_1 - \eta g_1$
- $x_2 = x_2 - \eta g_2$
- ...
- $x_n = x_n - \eta g_n$

4. 重复步骤2-3, 直到:

- 迭代次数达到最大迭代次数 (max_iter) 。
- 在连续若干次迭代过程中, 函数值 y 的变化 Δy 小于指定的阈值 (tol) 。

说明:

- η : 学习率, 用来控制更新幅度的大小。
- 梯度的方向不一定指向极值, 但是, 沿着梯度的方向更新可以让函数值朝着极值靠近。

课堂练习

如果通过梯度上升求解函数极大值, 我们该如何更新点的坐标值?

- A $x_n = x_n + \eta g_n$
- B $x_n = x_n - \eta g_n$
- C $x_n = x_n * \eta g_n$
- D $x_n = x_n / (\eta g_n)$



小

结

一维梯度下降

使用梯度下降法求解函数 $y = x^2 - 2x + 1$ 的最小值。

编写程序

```
1 def fun(x):
2     """定义函数y = x^2 - 2x + 1, 根据自变量x返回函数值y。
3
4     Parameters
5     -----
6     x : array或者标量
7         自变量。
8
9     Returns
10    -----
11    y : array或者标量
```

```

12     根据自变量x，返回x所对应的y值。
13     """
14     return x ** 2 - 2 * x + 1
15
16
17 def grad_f(x):
18     """定义原函数的导函数。y = 2x - 2，根据自变量x返回函数值y。
19
20     Parameters
21     -----
22     x : array或者标量
23         自变量。
24
25     Returns
26     -----
27     y : array或者标量
28         根据自变量x，返回x所对应的y值。
29     """
30     return 2 * x - 2
31
32
33 def grad_descent(f, g, x_init, eta=0.1, tol=1e-4, max_iter=100, verbose=True):
34     """通过梯度下降求解一元函数的极小值。
35
36     根据给定的函数f与初始点，通过在循环中计算梯度，并更新初始点位置，
37     从而逼近函数的极小值。
38
39     Parameters
40     -----
41     f : function
42         原函数。
43     g : function
44         梯度函数。对一元函数来说，梯度函数就是导函数。
45     x_init : float
46         自变量初始点的位置（值）。
47     eta : float, 默认值为0.1
48         学习率。控制梯度下降中x_init的更新幅度。
49     tol : float, 默认值为1e-4
50         容忍值。当多次函数值的差值小于tol时，停止迭代。
51     max_iter : int, 默认值为100
52         最大迭代次数。当达到最大迭代次数后，停止迭代。
53     verbose : bool, 默认值为True。
54         是否打印更新过程中的轨迹信息。
55
56     Returns
57     -----
58     x_list : list
59         自变量x的更新轨迹。
60     y_list : list
61         函数值y的更新轨迹。
62     """
63     # 定义x_list与y_list，用来保存梯度下降过程中，自变量x与函数值y的更新轨迹。
64     x_list = []
65     y_list = []
66     x = x_init
67
68     for i in range(max_iter):
69         x_list.append(x)
70         y = f(x)
71         y_list.append(y)
72         # 如果在两次迭代中，函数y比上一次的减少值小于tol，则停止迭代。
73         if len(y_list) >= 2 and y_list[-2] - y < tol:
74             break

```

```

75     x -= eta * g(x)
76     if verbose:
77         for index, (a, b) in enumerate(zip(x_list, y_list), start=1):
78             print(f"第{index}次迭代, x={a}, y={b}")
79     return x_list, y_list
80
81
82 x_list, y_list = grad_descent(fun, grad_f, x_init=10)

```

```

1  第1次迭代, x=10, y=81
2  第2次迭代, x=8.2, y=51.839999999999996
3  第3次迭代, x=6.76, y=33.1776
4  第4次迭代, x=5.608, y=21.233663999999997
5  第5次迭代, x=4.6864, y=13.58954496
6  第6次迭代, x=3.9491199999999997, y=8.697308774399998
7  第7次迭代, x=3.3592959999999996, y=5.5662776156159985
8  第8次迭代, x=2.8874367999999997, y=3.562417673994238
9  第9次迭代, x=2.5099494399999998, y=2.279947311356313
10 第10次迭代, x=2.2079595519999997, y=1.4591662792680404
11 第11次迭代, x=1.9663676415999998, y=0.9338664187315455
12 第12次迭代, x=1.7730941132799998, y=0.5976745079881893
13 第13次迭代, x=1.6184752906239999, y=0.38251168511244105
14 第14次迭代, x=1.4947802324992, y=0.24480747847196227
15 第15次迭代, x=1.3958241859993599, y=0.15667678622205594
16 第16次迭代, x=1.316659348799488, y=0.10027314318211578
17 第17次迭代, x=1.2533274790395903, y=0.06417481163655414
18 第18次迭代, x=1.2026619832316723, y=0.041071879447394544
19 第19次迭代, x=1.1621295865853378, y=0.026286002846332535
20 第20次迭代, x=1.1297036692682703, y=0.01682304182165284
21 第21次迭代, x=1.1037629354146161, y=0.01076674676585787
22 第22次迭代, x=1.0830103483316929, y=0.0068907179301489485
23 第23次迭代, x=1.0664082786653544, y=0.00441005947529538
24 第24次迭代, x=1.0531266229322835, y=0.0028224380641890257
25 第25次迭代, x=1.0425012983458268, y=0.0018063603610809498
26 第26次迭代, x=1.0340010386766614, y=0.001156070631091799
27 第27次迭代, x=1.0272008309413292, y=0.0007398852038986714
28 第28次迭代, x=1.0217606647530633, y=0.0004735265304951497
29 第29次迭代, x=1.0174085318024506, y=0.00030305697951682475
30 第30次迭代, x=1.0139268254419604, y=0.00019395646689090995
31 第31次迭代, x=1.0111414603535684, y=0.00012413213881012908

```

结果可视化

```

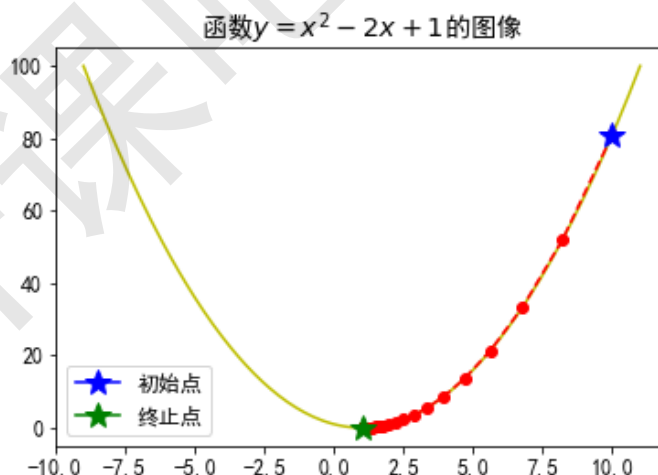
1  def plot_result(f, x, x_li, y_li):
2      """绘制可视化结果。
3
4      Parameters
5      -----
6      f : function
7          原函数。
8      x : array
9          绘制图像的定義域范围。
10     x_li : list
11         自变量x的更新轨迹。
12     y_li : list
13         自变量y的更新轨迹。
14     """
15
16     y = f(x)
17     plt.plot(x, y, c="y")
18     plt.title("函数 $y=x^2-2x+1$ 的图像")

```

```

19 plt.plot(x_list, y_list, marker="o", ls="--", c="r")
20 plt.plot(x_list[0], y_list[0], marker="*", ms=15, c="b", label="初始点")
21 plt.plot(x_list[-1], y_list[-1], marker="*", ms=15, c="g", label="终止点")
22 plt.legend()
23 plt.show()
24
25
26 domain = np.linspace(-9, 11, 200)
27 plot_result(fun, domain, x_list, y_list)

```



学习率对梯度下降的影响

在梯度下降过程中，学习率控制每次更新的幅度。学习率的选择应该适度，既不是越大越好，也不是越小越好。

较大的学习率

如果学习率较大，则：

- 优点：每次更新的幅度加快，减少迭代次数，能够更快的到达极值点。
- 缺点：如果学习率过大，容易跳过极值点，出现震荡，严重时会导致发散效果，函数值不降反升。

较大的学习率，可以更快的达到极值点。

```

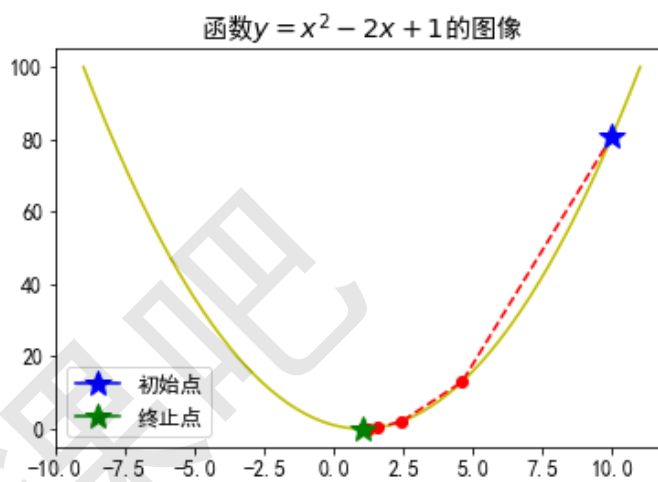
1 x_list, y_list = grad_descent(fun, grad_f, x_init=10, eta=0.3)
2 plot_result(fun, domain, x_list, y_list)

```

```

1 第1次迭代, x=10, y=81
2 第2次迭代, x=4.6000000000000005, y=12.960000000000003
3 第3次迭代, x=2.4400000000000004, y=2.0736000000000008
4 第4次迭代, x=1.576, y=0.33177600000000007
5 第5次迭代, x=1.2304, y=0.053084160000000005
6 第6次迭代, x=1.09216, y=0.008493465599999972
7 第7次迭代, x=1.036864, y=0.0013589544959999866
8 第8次迭代, x=1.0147456, y=0.0002174327193600334
9 第9次迭代, x=1.00589824, y=3.478923509758758e-05
10 第10次迭代, x=1.002359296, y=5.566277615720594e-06

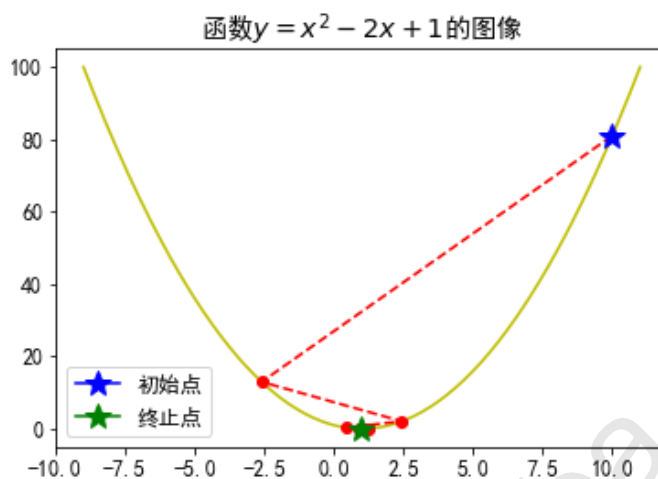
```



如果学习率过大，会出现震荡的现象，但依然能够到达极值点。

```
1 x_list, y_list = grad_descent(fun, grad_f, x_init=10, eta=0.7)
2 plot_result(fun, domain, x_list, y_list)
```

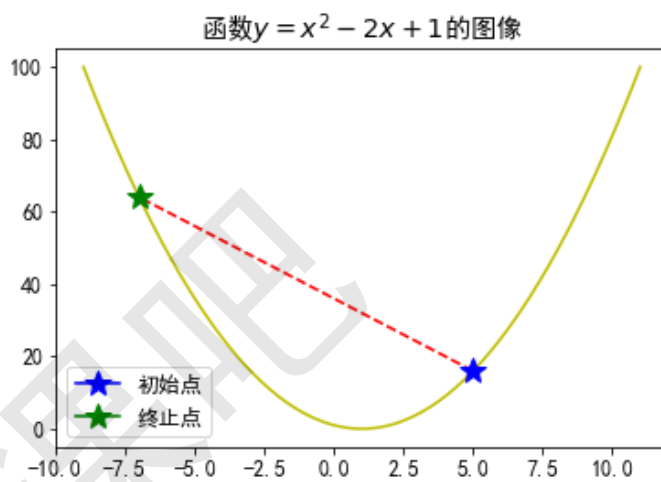
```
1 第1次迭代, x=10, y=81
2 第2次迭代, x=-2.5999999999999996, y=12.959999999999997
3 第3次迭代, x=2.4399999999999995, y=2.0735999999999999
4 第4次迭代, x=0.42400000000000004, y=0.33177599999999996
5 第5次迭代, x=1.2304, y=0.053084160000000005
6 第6次迭代, x=0.90784, y=0.0084934655999999972
7 第7次迭代, x=1.036864, y=0.00135895449599999866
8 第8次迭代, x=0.9852544, y=0.0002174327193600334
9 第9次迭代, x=1.00589824, y=3.478923509758758e-05
10 第10次迭代, x=0.9976407039999999, y=5.566277615609572e-06
```



如果学习率过大，会令函数值不降反升。

```
1 x_list, y_list = grad_descent(fun, grad_f, x_init=5, eta=1.5)
2 plot_result(fun, domain, x_list, y_list)
```

```
1 第1次迭代, x=5, y=16
2 第2次迭代, x=-7.0, y=64.0
```

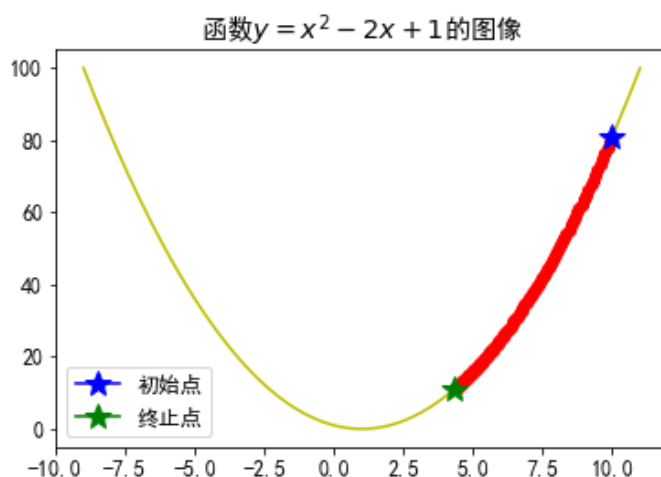



较小的学习率

如果学习率较小，则：

- 优点：不容易出现震荡现象（跳过极值点），函数值能够稳定下降。
- 缺点：更新幅度过慢，需要更多次的迭代，降低程序的性能。可能在达到最大迭代次数时依然没有靠近极值点。

```
1 x_list, y_list = grad_descent(fun, grad_f, x_init=10, eta=0.005, verbose=False)
2 plot_result(fun, domain, x_list, y_list)
```



课堂练习



使用梯度法求解函数极值中，梯度为我们提供什么信息？

- A 每次更新的方向。
B 每次更新值的大小。
C A与B。



二维梯度下降

使用梯度下降求解方程 $y = 0.2(x_1 + x_2)^2 - 0.3x_1x_2 + 0.4$ 的最小值。

编写程序

```

1 from mpl_toolkits.mplot3d import Axes3D
2 %matplotlib qt
3
4
5 def fun(x1, x2):
6     """定义二元函数，根据自变量x1与x2返回函数值y。
7
8     Parameters
9     -----
10    x1, x2 : array或者标量
11            自变量。
12
13    Returns
14    -----
15    y : array或者标量
16        根据自变量x1与x2，返回其所对应的y值。
17    """
18    return 0.2 * (x1 + x2) ** 2 - 0.3 * x1 * x2 + 0.4
19
20
21 def grad_f_x1(x1, x2):
22     """定义原函数对x1的偏导函数。根据自变量x1返回偏导函数值y。
23
24     Parameters
25     -----
26    x1, x2 : array或者标量
27            自变量。
28
29    Returns
30    -----
31    y : array或者标量
32        根据自变量x1与x2，返回其所对应的y值。
33    """
34    return 0.4 * (x1 + x2) - 0.3 * x2
35
36
37 def grad_f_x2(x1, x2):
38     """定义原函数对x2的偏导函数。根据自变量x1返回偏导函数值y。
39
40     Parameters
41     -----
42    x1, x2 : array或者标量
43            自变量。
44
45    Returns
46    -----
47    y : array或者标量
48        根据自变量x1与x2，返回其所对应的y值。
49    """
50    return 0.4 * (x1 + x2) - 0.3 * x1
51
52
53 def grad_descent(f, g, x_init, eta=0.1, tol=1e-4, max_iter=100, verbose=True):
54     """通过梯度下降求解二元函数的极小值。
55
56     根据给定的函数f与初始点，通过在循环中计算梯度，并更新初始点位置，
57     从而逼近函数的极小值。
58
59     Parameters
60     -----
61    f : function
62        原函数。
63    g : tuple, 形状为(2, )

```

```

64         f对x1与x2的偏导函数。
65     x_init : tuple, 形状为(2, )
66             自变量初始点的位置(值)。
67     eta : float, 默认值为0.1
68           学习率。控制梯度下降中x_init的更新幅度。
69     tol : float, 默认值为1e-4
70           容忍值。当两次函数值的差值小于tol时, 停止迭代。
71     max_iter : int, 默认值为100
72              最大迭代次数。当达到最大迭代次数后, 停止迭代。
73     verbose : bool, 默认值为True。
74              是否打印更新过程中的轨迹信息。
75
76     Returns
77     -----
78     x_list : list
79             自变量x1与x2的更新轨迹。
80     y_list : list
81             函数值y的更新轨迹。
82     """
83     # 定义x_list与y_list, 用来保存梯度下降过程中, 自变量x与函数值y的更新轨迹。
84     x_list = []
85     y_list = []
86     x1, x2 = x_init
87     g1, g2 = g
88     for i in range(max_iter):
89         x_list.append((x1, x2))
90         y = f(x1, x2)
91         y_list.append(y)
92         # 如果在两次迭代中, 函数y比上一次的减少值小于tol, 则停止迭代。
93         if len(y_list) >= 2 and y_list[-2] - y < tol:
94             break
95         x1 -= eta * g1(x1, x2)
96         x2 -= eta * g2(x1, x2)
97     if verbose:
98         for index, (a, b) in enumerate(zip(x_list, y_list), start=1):
99             print(f"第{index}次迭代, x={a}, y={b}")
100     return x_list, y_list
101
102
103 x_list, y_list = grad_descent(fun, g=(grad_f_x1, grad_f_x2), x_init=(4.8, 4.5),
                               eta=0.4)

```

```

1  第1次迭代, x=(4.8, 4.5), y=11.218000000000002
2  第2次迭代, x=(3.8519999999999994, 3.62592), y=7.393744353280001
3  第3次迭代, x=(3.0906431999999993, 2.9221470719999996), y=4.921335177768551
4  第4次迭代, x=(2.4792544051199994, 2.3554333642751994), y=3.322925602204132
5  第5次迭代, x=(1.9883563657297916, 1.899029771361976), y=2.2895698153912587
6  第6次迭代, x=(1.5942581563585458, 1.5314146816897178), y=1.6215250339872198
7  第7次迭代, x=(1.2779202640735898, 1.2352715220564194), y=1.1896530378735917
8  第8次迭代, x=(1.0240421609395587, 0.99666639208981), y=0.9104640894897645
9  第9次迭代, x=(0.8203287595056369, 0.8043866189752149), y=0.729981569022374
10 第10次迭代, x=(0.6569006932257264, 0.6494087322101514), y=0.6133097490838161
11 第11次迭代, x=(0.5258202330212041, 0.524470525735679), y=0.5378889713694954
12 第12次迭代, x=(0.42071017470838423, 0.42372683462963495), y=0.4891349153602483
13 第13次迭代, x=(0.33644747336985736, 0.34247264215409906), y=0.4576192881072987
14 第14次迭代, x=(0.26891697194451625, 0.2769203405316626), y=0.43724710050455784
15 第15次迭代, x=(0.21481344281212714, 0.2240205483341115), y=0.42407826678257415
16 第16次迭代, x=(0.17148247002882233, 0.18131796179950077), y=0.4155657733546593
17 第17次迭代, x=(0.13679255635223073, 0.1468353856574914), y=0.4100631655676179
18 第18次迭代, x=(0.10903233190957415, 0.11898043067590981), y=0.4065061498379129
19 第19次迭代, x=(0.08682794157700588, 0.096470444104684), y=0.40420676061331395
20 第20次迭代, x=(0.06907665316049758, 0.07827210692151465), y=0.40272029886495364

```

```

21 第21次迭代, x=(0.054893504377957376, 0.063552829638954), y=0.4017593155488039
22 第22次迭代, x=(0.04356843049192603, 0.05164163967704432), y=0.40113800793568855
23 第23次迭代, x=(0.03453181602613609, 0.041997704687671784), y=0.40073627640459863
24 第24次迭代, x=(0.027326817274447447, 0.034184999246666396), y=0.40047649054596324
25 第25次迭代, x=(0.0215871265406692, 0.027851914305573006), y=0.400308470912405
26 第26次迭代, x=(0.017019109721939207, 0.02271484362780376), y=0.40019978148495444
27 第27次迭代, x=(0.013387458421316783, 0.018544970310502486), y=0.40012945499525565

```

结果可视化

```

1  def plot_result(f, x, x_li, y_li):
2      """绘制可视化结果。
3
4      绘制结果包括两张图，一张图为在三维空间中，点坐标的更新轨迹。
5      另外一张图为二元函数等高线与坐标更新轨迹在底面的投影。
6
7      Parameters
8      -----
9      f : function
10         原二元函数。
11      x : tuple, 形状为(2, 定义域取样数)
12         x1与x2的定义域数组。
13      x_li : list
14         自变量x1与x2的更新轨迹。
15      y_li : list
16         自变量y的更新轨迹。
17      """
18
19      x1, x2 = x
20      X1, X2 = np.meshgrid(x1, x2)
21      X = np.array([X1.ravel(), X2.ravel()]).T
22      y = f(X[:, 0], X[:, 1])
23      Y = y.reshape(X1.shape)
24      fig = plt.figure()
25      ax = Axes3D(fig)
26      surf = ax.plot_surface(X1, X2, Y, rstride=5, cstride=5, cmap="rainbow",
27                             alpha=0.8)
28      # 将自变量的轨迹列表转换为ndarray数组，方便按列单独提取x1与x2的轨迹。
29      array = np.asarray(x_li)
30      x1_trace = array[:, 0]
31      x2_trace = array[:, 1]
32      ax.plot(x1_trace, x2_trace, y_list, c="b", ls="--", marker="o")
33      ax.set_title("函数$y = 0.2(x_1 + x_2)^2 - 0.3x_1x_2 + 0.4$")
34      # 绘制三维图时，参数必须是数组类型，不支持标量，这点与绘制二维图不同。
35      ax.plot(x1_trace[0:1], x2_trace[0:1], y_list[0:1], marker="*", ms=15, c="b",
36             label="初始点")
37      ax.plot(x1_trace[-1:], x2_trace[-1:], y_list[-1:], marker="*", ms=15, c="g",
38             label="终止点")
39      # 为曲面对象增加颜色条。
40      fig.colorbar(surf)
41      ax.legend()
42      # 创建新的画布，用来绘制等高线图。
43      fig2 = plt.figure()
44      ax2 = fig2.gca()
45      m = ax2.contourf(X1, X2, Y, 10)
46      ax2.scatter(x1_trace, x2_trace, c="r")
47      # 为等高线图增加颜色条。
48      fig2.colorbar(m)
49      ax2.set_title("轨迹更新投影图")
50      plt.show()

```

```
50 x1_domain = np.arange(-5, 5, 0.1)
51 x2_domain = np.arange(-5, 5, 0.1)
52 plot_result(fun, (x1_domain, x2_domain), x_list, y_list)
```



梯度与损失函数

求解损失函数最小值

结合之前的介绍，我们完全可以使用梯度下降的方法来针对损失函数求解极小值。

对于损失函数：

$$J(w) = \frac{1}{2} \sum_{i=1}^m (y^{(i)} - w^T x^{(i)})^2$$

我们可以使用梯度下降的方式，不断去调整权重 w ，进而去减小损失函数 $J(w)$ 的值。经过不断迭代，最终求得最优的权重 w ，使得损失函数的值最小（近似最小）。调整方式为：

$$w_j = w_j - \eta \frac{\partial J(w)}{\partial w_j}$$

我们这里单独对一个样本求梯度来演示，所有样本，只需要分别对每个式子进行求梯度，最后将每个求梯度的结果求和即可。

$$\begin{aligned} \frac{\partial J(w)}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} (y^{(i)} - w^T x^{(i)})^2 \\ &= 2 * \frac{1}{2} * (y^{(i)} - w^T x^{(i)}) \frac{\partial}{\partial w_j} (y^{(i)} - w^T x^{(i)}) \\ &= (y^{(i)} - w^T x^{(i)}) \frac{\partial}{\partial w_j} (y^{(i)} - \sum_{j=1}^n w_j x_j^{(i)}) \\ &= -(y^{(i)} - w^T x^{(i)}) x_j^{(i)} \\ &= -(y^{(i)} - \hat{y}^{(i)}) x_j^{(i)} \end{aligned}$$

梯度下降分类

根据权重更新的方式不同，可以将梯度下降分为三类：

- 随机梯度下降 (SGD-Stochastic Gradient Descent)
- 批量梯度下降 (BGD-Batch Gradient Descent)
- 小批量梯度下降 (MBGD-Mini-Batch Gradient Descent)

随机梯度下降

随机梯度下降每次使用一个样本更新权重，其中样本 (i) 可能是按顺序选择，也可能是随机选择。

$$w_j = w_j + \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

批量梯度下降

批量梯度下降使用所有样本来更新权重。

$$w_j = w_j + \frac{1}{m}\eta \sum_{i=1}^m (y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

小批量梯度下降

小批量梯度下降每次使用一个批次的样本更新数据，样本批次数量为 k ，当 $k = 1$ 时，小批量梯度下降等同于随机梯度下降，当 $k = m$ 时，小批量梯度下降等同于批量梯度下降。

$$w_j = w_j + \frac{1}{k}\eta \sum_{i=1}^k (y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

三种梯度下降的不同

- 随机梯度下降更新速度较快，但是方向上不够稳定。
- 批量梯度下降更新方向较稳定，但是更新速度较慢。
- 小批量梯度下降是随机梯度下降与批量梯度下降之间的一个折中。

sklearn实现随机梯度下降

```
1 from sklearn.linear_model import SGDRegressor
2 from sklearn.datasets import make_regression
3
4 X, y, coef = make_regression(n_samples=1000, n_features=5, bias=2.5, coef=True,
5                             noise=5, random_state=0)
6 print(f"真实权重: {coef}")
7 # eta0 指定学习率
8 sgd = SGDRegressor(eta0=0.2)
9 sgd.fit(X, y)
10 print(f"预测权重: {sgd.coef_}")
11 print(f"预测截距: {sgd.intercept_}")
12 print(f"R^2值: {sgd.score(X, y)}")
```

```
1 真实权重: [41.20593377 66.49948238 10.71453179 60.19514224 25.96147771]
2 预测权重: [40.43149409 66.27914415 11.26502536 61.25753591 24.93162419]
3 预测截距: [3.04732968]
4 R^2值: 0.9972227400981705
```



课堂练习



对于LinearRegression与SGDRegressor, 多次运行同一个程序, 拟合出来的权重会相同吗?

- A 二者都相同。
- B 二者都不同。
- C 前者相同, 后者可能不同。
- D 前者可能不同, 后者相同。



数据标准化

在很多机器学习算法中(包括梯度下降), 如果数据集的特征之间不是同一个数量级(量纲), 则数量级大的特征就可能会成为模型函数的主要影响者, 从而导致评估器无法从其他特征中学习信息, 导致训练的效果较差。

拟合对比

我们以波士顿房价数据集为例, 来观察线性回归与梯度下降的表现。

```
1 from sklearn.datasets import load_boston
2 from sklearn.model_selection import train_test_split
3 from sklearn.linear_model import LinearRegression, SGDRegressor
4
5 X, y = load_boston(return_X_y=True)
6 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
7 random_state=0)
8 lr = LinearRegression()
9 lr.fit(X_train, y_train)
10 print("线性回归拟合分值: ")
11 print(lr.score(X_train, y_train))
12 print(lr.score(X_test, y_test))
13 sgd = SGDRegressor(max_iter=100, eta0=0.01)
14 sgd.fit(X_train, y_train)
15 print("随机梯度下降拟合分值: ")
16 print(sgd.score(X_train, y_train))
17 print(sgd.score(X_test, y_test))
```

```
1 线性回归拟合分值:
2 0.7697699488741149
3 0.6354638433202129
4 随机梯度下降拟合分值:
5 -2.341718927888053e+25
6 -2.35504100217903e+25
```

```
1 D:\software\anaconda3\lib\site-
  packages\sklearn\linear_model\_stochastic_gradient.py:1187: ConvergenceWarning:
  Maximum number of iteration reached before convergence. Consider increasing max_iter
  to improve the fit.
2 ConvergenceWarning)
```

由结果得知, 梯度下降远不及线性回归的效果好, 然而, 这并不表示梯度下降真的很差, 仅仅是因为梯度下降需要对训练数据进行标准化而已。

```
1 from sklearn.datasets import load_boston
2 from sklearn.model_selection import train_test_split
3 from sklearn.linear_model import SGDRegressor
4 from sklearn.pipeline import Pipeline
5 from sklearn.preprocessing import StandardScaler, MinMaxScaler
6
7 X, y = load_boston(return_X_y=True)
8 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
9 random_state=0)
10 pipeline = Pipeline([("ss", StandardScaler()), ("sgd", SGDRegressor(eta0=0.01,
11 max_iter=100))])
12 # pipeline = Pipeline([("ss", MinMaxScaler((-1, 1))), ("sgd", SGDRegressor(eta0=0.01,
13 max_iter=100))])
14 pipeline.fit(X_train, y_train)
15 print(pipeline.score(X_train, y_train))
16 print(pipeline.score(X_test, y_test))
```

```
1 0.7690155411197578
2 0.6272972092554572
```



课堂练习



在使用梯度下降求解极值前，一定要对训练数据进行标准化，这种说法正确吗？

- A 正确
B 错误



作业

- 改写本节梯度下降的程序，不传递梯度函数，来实现同样的功能。
 - 提示：通过数值微分，自行计算梯度。
- 通过make_regression函数构造回归数据集，手写随机梯度下降算法实现回归任务，要求：
 - 编写一个梯度下降类，具有fit与predict方法。
 - 具有coef与intercept属性，能够返回权重与偏置。
- 对梯度下降进行改进，连续n次没有改进，才退出循环。