什么是回归问题

身高预测

本小节我们开始正式学习机器学习中的第一个算法，线性回归算法。下面我们来看一个小例子。

身高/m	1.1	1.16	1.35	1.51	1.55	1.6	1.63	1.67	1.75
体重/kg	20	24.2	36.3	46.5	49.1	52.3	54.2	56.7	?

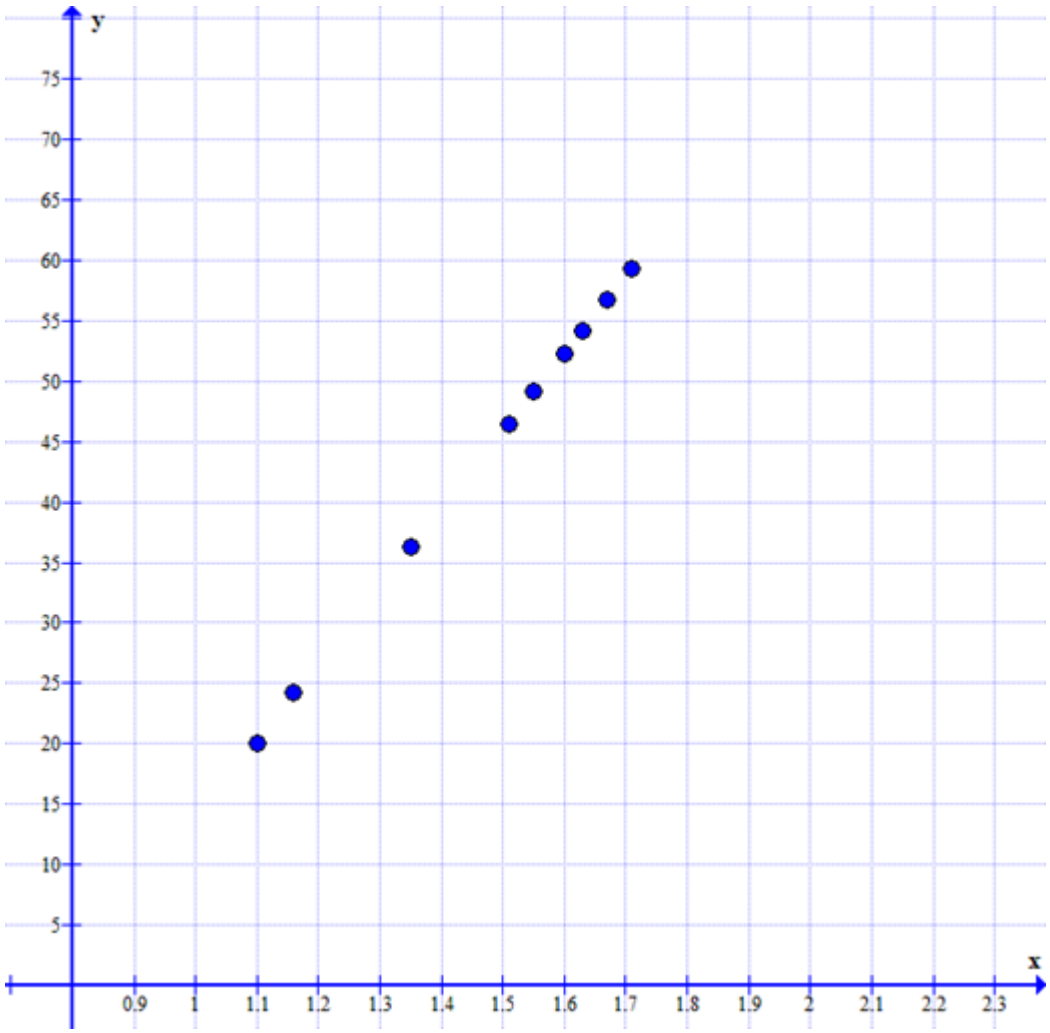
假设你是某地区统计局的工作人员，为了了解青少年的健康状况，现在要求你根据该地区统计的青少年的身高体重情况，预测当青少年的身高达到1.75米时的体重。

面对这个问题，你打算如何解决呢？大家不妨思考一下。

一元线性回归

基于假设求解

面对这个问题，我们不妨做这样的一个尝试，我们分别以身高和体重作为横纵坐标绘制直角坐标系，然后将这些数据点绘制在这个直角坐标系中：



根据数据在这个坐标系中的分布情况，我们发现似乎这些点都落在了同一条直线上，我们尝试绘制这样一条直线，它能够穿过这些所有的点：

我们发现绘制的过程非常顺利，在实际的绘制过程中，正好有这样一条直线非常巧合的穿过了所有的点，那么我们是否可以作这样的假设，该地区统计的身高与体重的关系全都符合这条直线所表示的这样一个关系？

那么这样的话，只需要我们把这条直线的方程求出来，再讲对应的身高代入函数，就能够将我们需要求的体重数据求出来。这里随便选取两组身高体重信息，利用二元一次方程组求解方法求解：

$$\begin{cases} 1.1a + b = 20 \\ 1.67a + b = 56.7 \end{cases}$$

最终的求解得到这样的一个函数：

$$y = 63.93x - 50$$

然后将需要求解的身高信息带入到这里的x中，求得的y就表示是对应的体重信息，最终我们解决了这个问题：

身高/m	1.1	1.16	1.35	1.51	1.55	1.6	1.63	1.67	1.75
体重/kg	20	24.2	36.3	46.5	49.1	52.3	54.2	56.7	61.8

上述这个解决问题的过程，我们就可以将其称为一元线性回归。在统计学中，线性回归是一种用于表示自变量和因变量之间关系的线性模型。所谓一元线性回归就是指在这个函数中只有一个自变量。

假设

在前面的例子中，我们简单的介绍了一元线性回归解决问题的方式。一元线性回归假设自变量和因变量之间存在一种线性模型，一般在数学中，我们把这个假设表示为：

$$y = \beta_0 + \beta_1$$

在这个模型中的两个 β 是未知的常数系数，我们将其称为回归系数。不过在机器学习中，很多时候我们会把这个式子写作：

$$h_\theta(x) = \theta_0 + \theta_1 x$$

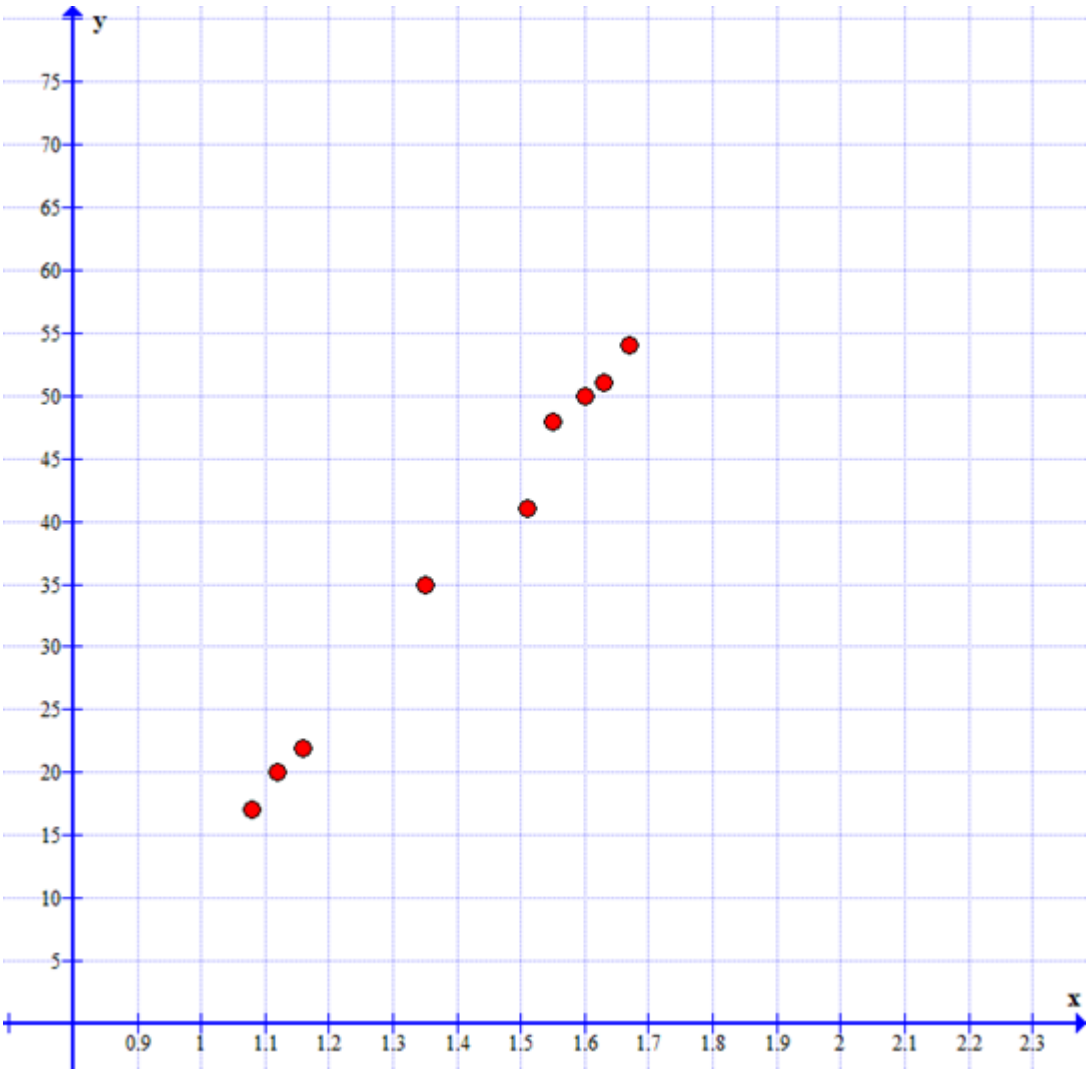
使用一元线性回归方法解决问题就是对数据进行如上的建模，然后求解这两个常数系数的过程。

存在偏差的情况

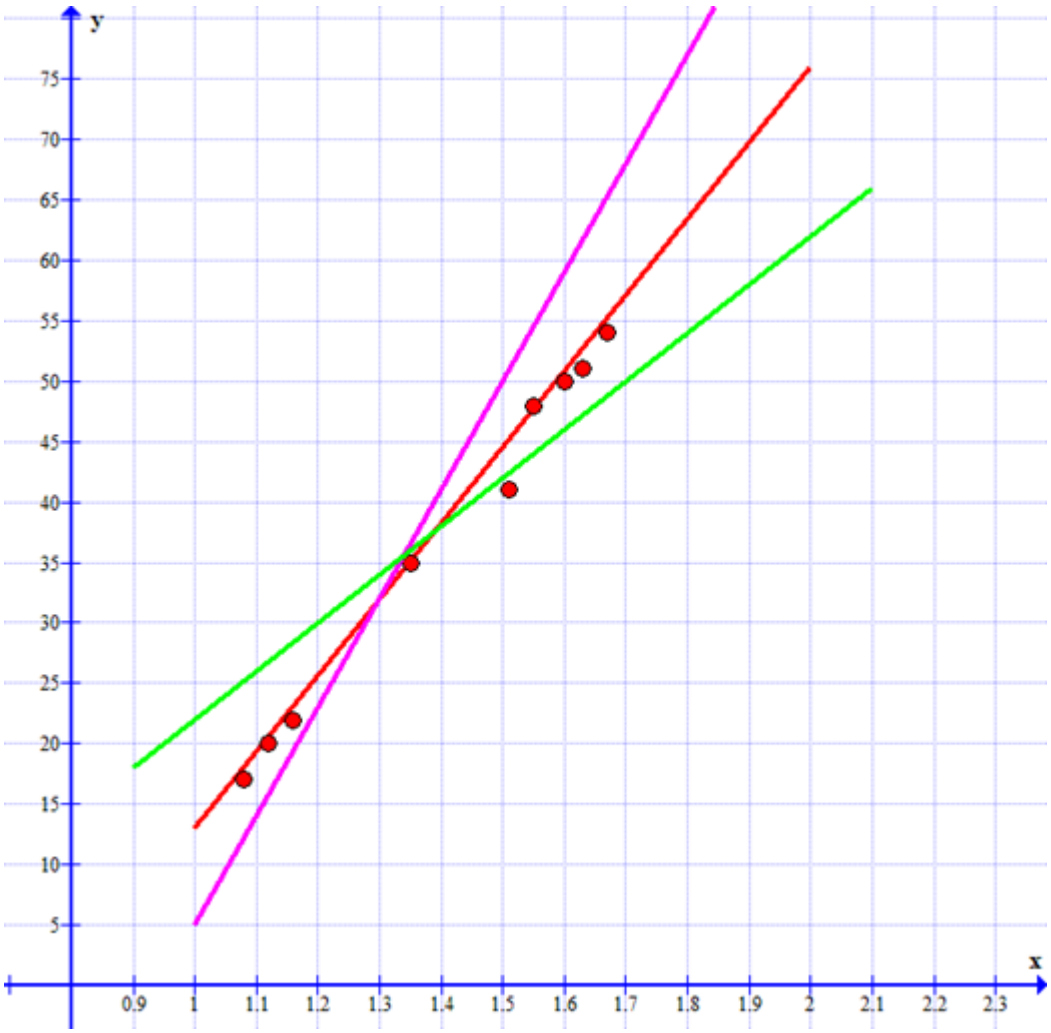
在前面的例子中，为了简化问题，将所有的数据经过处理后，使其完全符合一个线性关系。但在面对现实生活中的实际问题时，观测到的数据中是会含有偏差的，比如实际的身高和体重数据：

身高/m	1.1	1.16	1.35	1.51	1.55	1.6	1.63	1.67	1.75
体重/kg	20	22	35	41	48	50	51	54	?

不知道大家能否直接从这个表格中看出数据存在偏差的情况，不过我们可以用更直观的方式来观察：



我们仍然使用同样的方法将数据表示在这样一个直角坐标系中，通过观察，我们可以很明显的发现这些数据并不能完全符合一条直线的规律：

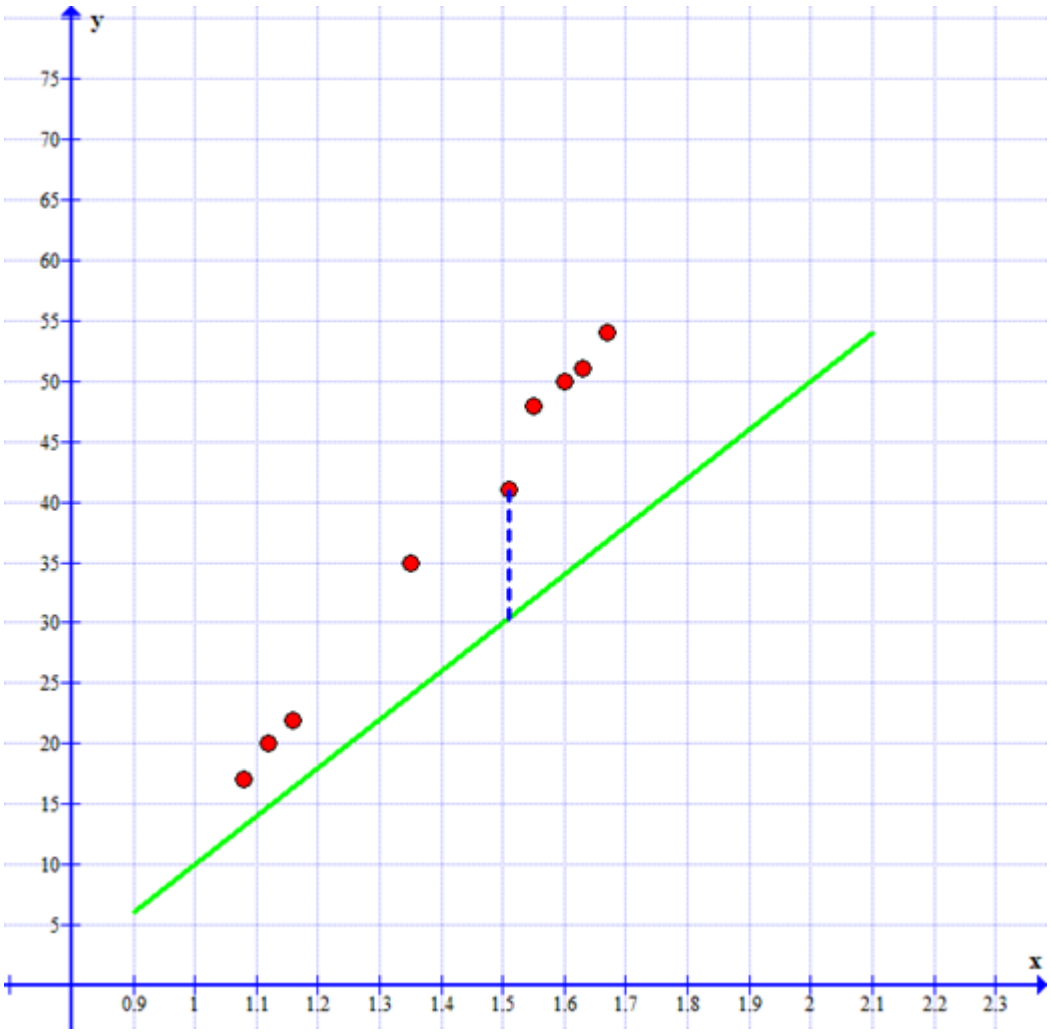


无论我们试图用什么样的直线去表示这个规律，都是会存在一定的误差。那么现在的问题就变成了，既然一定要有误差，那么我们是否可以找出误差最小的情况下的模型，用这个近似的模型来表示变量之间的关系呢？

显然是可以的，不过首先要解决的问题是我们如何来表示这个误差，然后使其最小。

最小二乘问题

这里我们提出一种方法来衡量模型和数据之间的误差，首先我们随便画一条直线，为了表示方便，我这里选择一条离我们的散点有一定距离的直线：



然后我从这些散点中任意选择一个点，作垂直于横坐标的线段，与直线产生一个交点，这个时候这个点与直线之间就形成了这样的一个线段，这个线段的长度就可以表示这个点和模型之间的误差，如果我们将所有点和模型的距离都表示出来，是不是就可以表示一个模型和数据的所有误差？

我们给出以下的一个正式定义，首先我们有这样的一些数据：

序号	特征	标签
1	x^1	y^1
2	x^2	y^2
……	……	……
i	x^i	y^i
……	……	……
N	x^N	y^N

对于每一条数据，我们可以表示为：

$$(x^{(i)}, y^{(i)}),\ i \in (1, N)$$

这样对于单条数据的误差，我们就可以表示为：

$$L_i = |h_\theta(x^{(i)} - y^{(i)})|$$

在这个式子中的函数h，表示的就是我们的模型，将一组数据中的x代入这个式子求得的就是根据模型得到的y的估计值，这个估计值与实际的y之间的差的绝对值，就是我们的误差。

那么全部的误差就可以表示为：

$$J(\theta_0, \theta_1) = |h_\theta(x^{(1)}) - y^{(1)}| + |h_\theta(x^{(2)}) - y^{(2)}| + \cdots + |h_\theta(x^{(i)}) - y^{(i)}|$$

为了便于计算，我们将绝对值替换成二次函数，然后化简，最终得到我们的损失函数为：

$$J(\theta_0, \theta_1) = (h_\theta(x^{(1)}) - y^{(1)})^2 + (h_\theta(x^{(2)}) - y^{(2)})^2 + \cdots + (h_\theta(x^{(i)}) - y^{(i)})^2$$

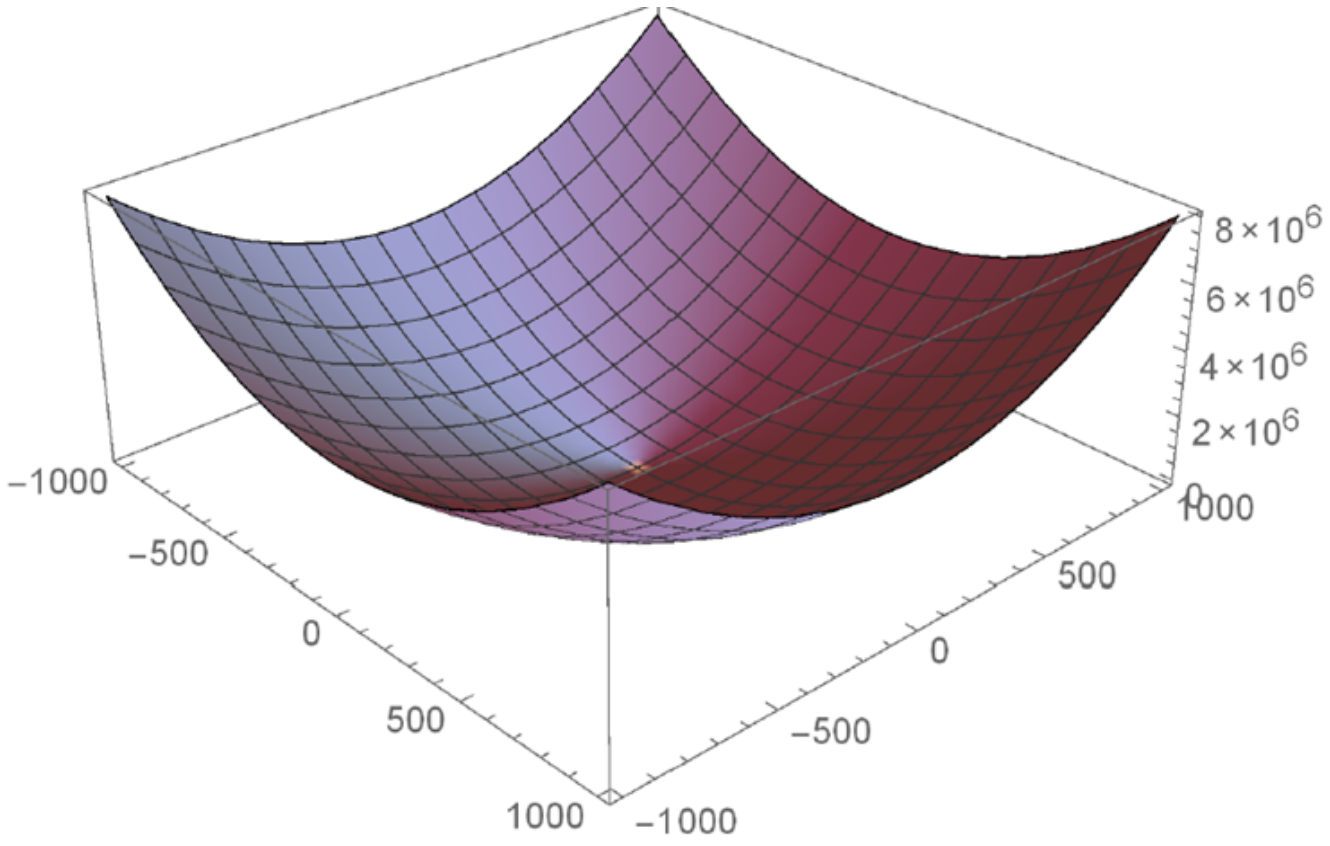
使用求和公式来表示：

$$J(\theta_0, \theta_1) = \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

求得这个函数的最小值的问题被称为最小二乘问题。那么接下来的问题就是如何求得误差的最小值。

梯度下降算法

梯度下降算法是解决最小二乘问题的一种方法，接下来我们主要介绍如何使用梯度下降算法求得系数。前面说到我们的损失函数是平方损失函数，这类函数的特点是有一个全局最小值，我们利用梯度下降算法求得这个全局最小值的过程类似于一个从山坡下到山谷中的过程，我们用下面的一个图像来表示平方损失函数：



这是一个在三维空间中的平方损失函数的图像，在这个曲面中有一个最低点，梯度下降算法求得这个最低点的过程类似于下山的过程：

1. 首先在这个三维空间的曲面上随机取一点，将这个点作为起点
2. 以这个点为中心，寻找往山谷中下降的方向
3. 按确定的方向往前移动

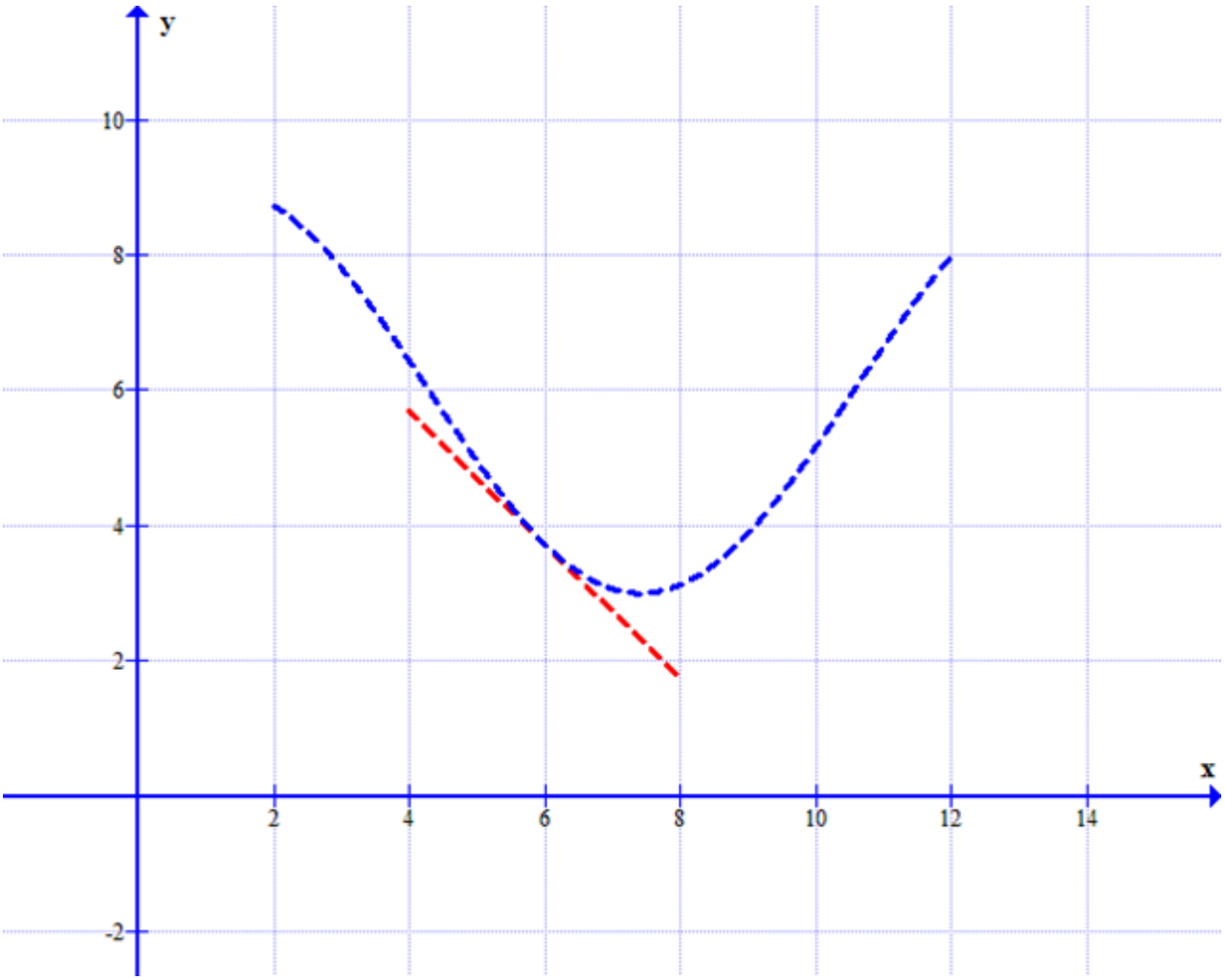
通过不断的迭代2）3）两个步骤，最终逐渐接近最低点。这就是梯度下降算法的基本原理，不过要实现这个算法，我们还面临要解决三个问题：

1. 如何选择下降的方向
2. 如何确定步长
3. 何时停止运动

选择下降的方向

我们首先来解决第一个问题，如何选择下降的方向。我们首先来看一个二维平面的例子，然后将其推广到三维和更高维度。

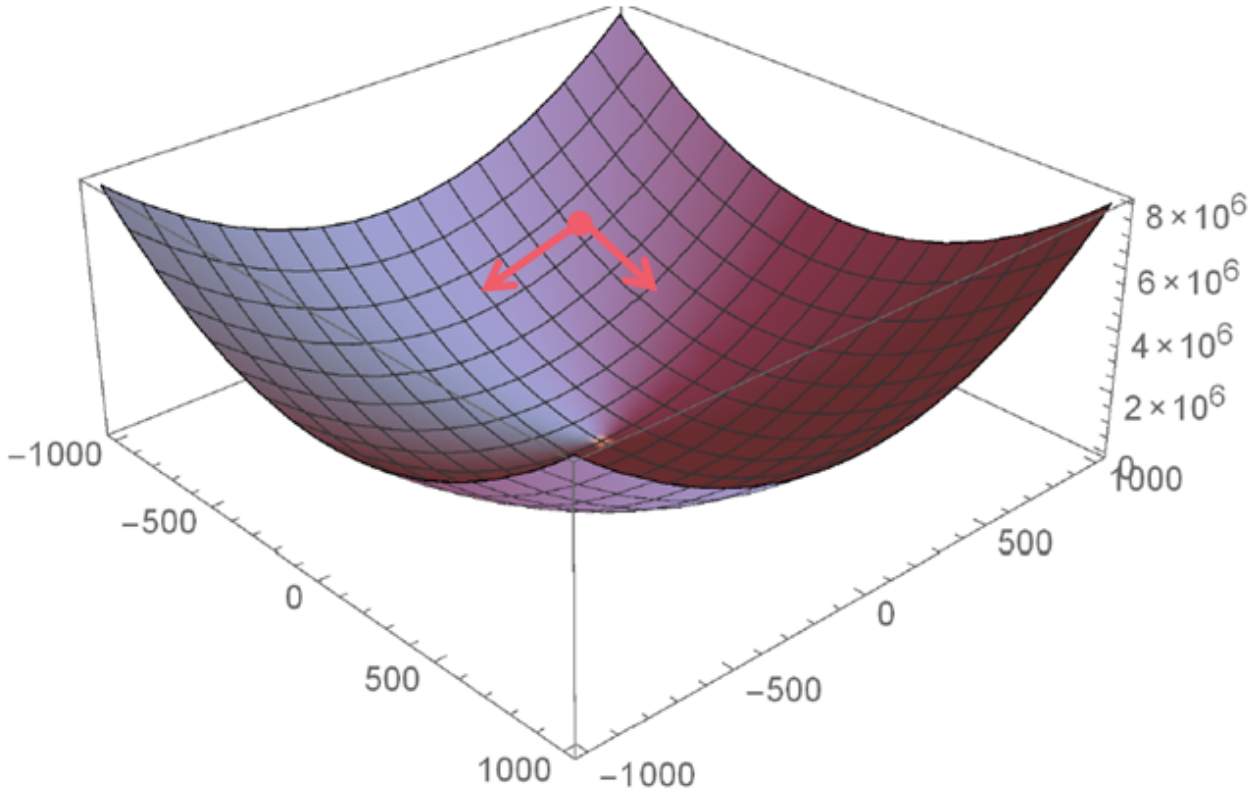
我们知道在二维平面上的一条曲线，可以使用导数表示曲线上某个点的倾斜度。即作曲线上某个点的切线，只需要找到导数绝对值最大的情况下，就是倾斜度最大的点。



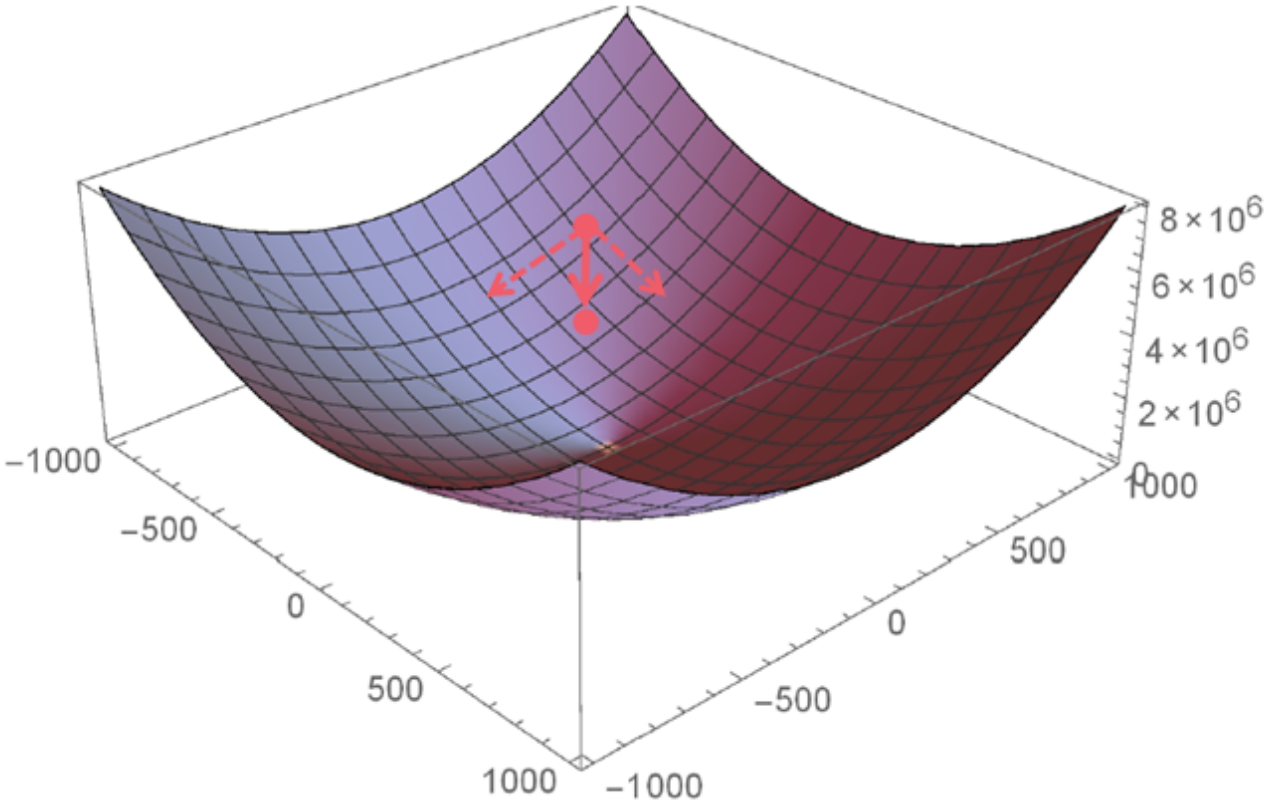
我们将这个概念推广一下，同样的道理，在一个三维空间的曲面上，如果能够计算出平面上某个点在某个方向上的导数，是否就可以找到去面上某个点倾斜度最大的方向了？

此处引入偏导和梯度的概念。对于一个曲面没有办法作其切线，那么我们可以将这个曲面变成曲线。想象一下，在曲面上任意取一个点，过这个点沿着平面上的两个方向中的任意一个方向平行的方向，一刀切下去，这个切面是不是就形成了一个曲线？

参照这个方法分别沿着平面的两个方向切出来两个曲线，然后对这两个曲线求导，所得到的就是这个曲面在这两个平面方向上在该点的偏导。



然后分别沿着两个偏导方向往前移动某个固定的步长：

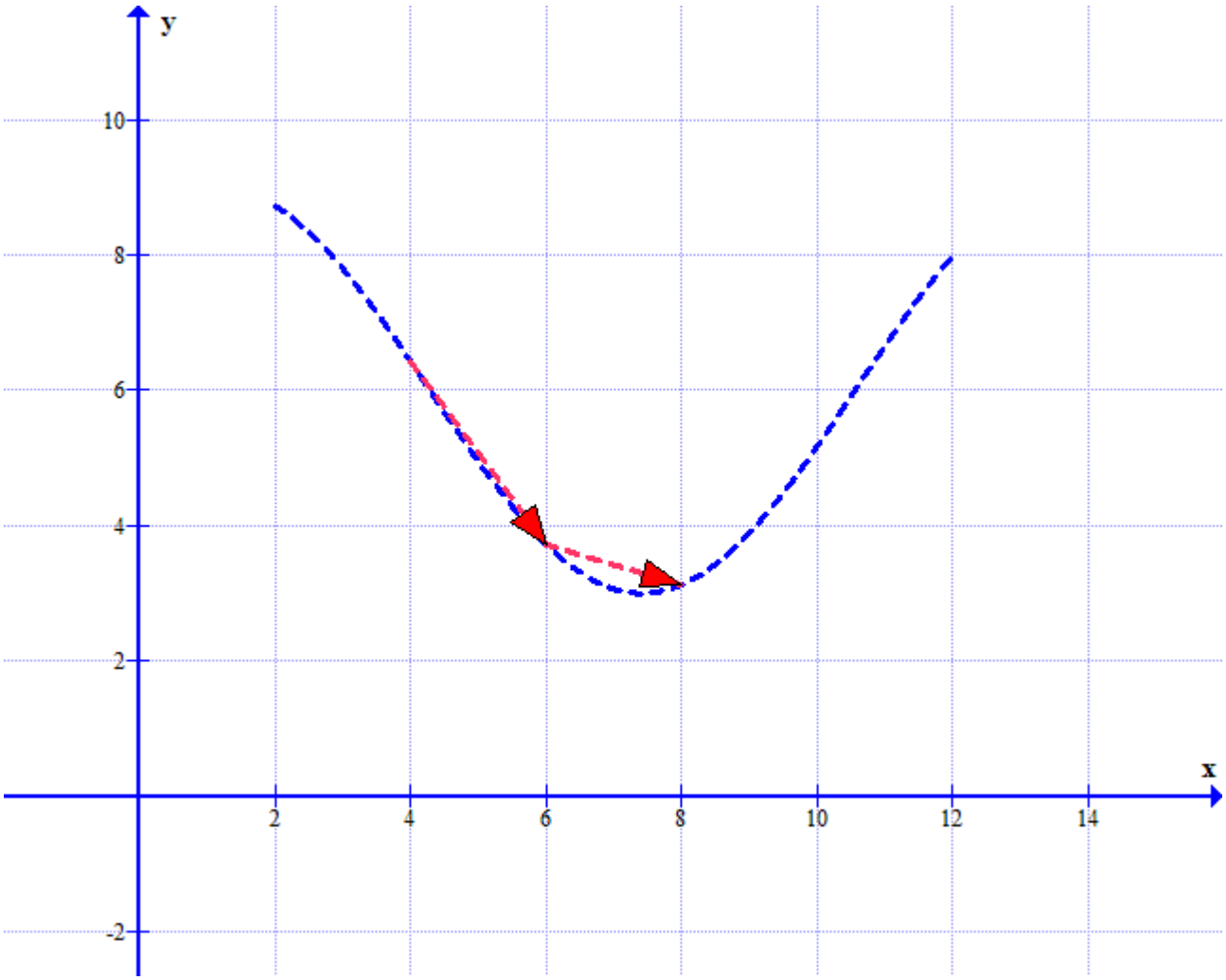


沿平面上每个轴分别移动一定距离后，落到曲面上的点，就相当于是往梯度方向下降一步之后新的落点。

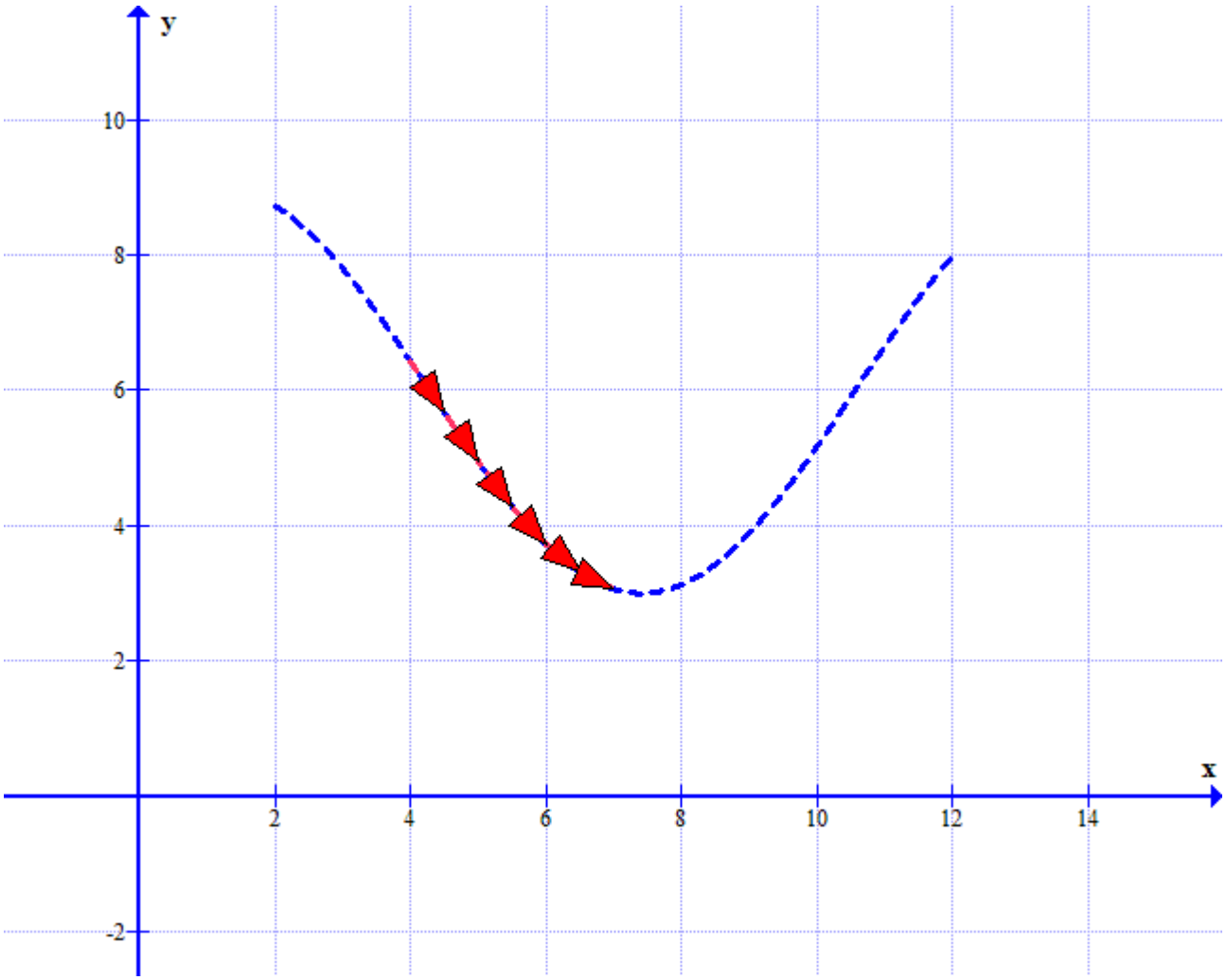
解决了下降方向的问题之后，我们来看看关于下降的步长问题。

下降的步长

我们仍然以二维平面上曲线的例子来看看，这个步长的大小对于整个梯度下降过程有什么用的影响，我们假设从如下的一个曲线上走到最低点：



如果每一步很长，那么在往下走的过程中就有可能越过最低点；



如果每一步很短，那么在往下走的过程中就会增加很多步运算，会导致运算周期过长；

并且大家需要注意一个问题，就是在我们从曲线较高位置往较低位置运动时，曲线的斜率并不是固定的，而是在不断变化中的，大家考虑一下如果步长固定，曲线的斜率对于每一次运动的距离有什么样的影响？

由于步长是固定的，而运动的距离是步长乘以斜率，因此，如果斜率越大，那么运动的距离就会越大，往最低点逼近的速度就越快；

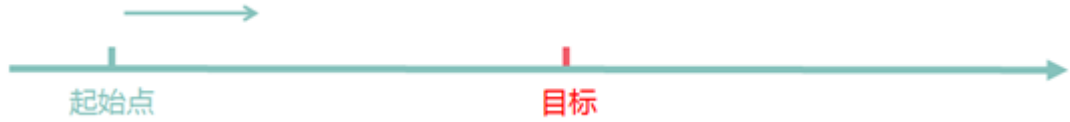
当逐渐逼近最低点时，斜率会逐渐趋近于零，那么对应的向最低点逼近的速度就会变得很慢；

因此，在实际实现算法的时候，最好要根据斜率的大小，适当的调整步长，具体的值没有明确的标准，需要根据实际的数据尝试计算之后才能得出。

何时停止运算

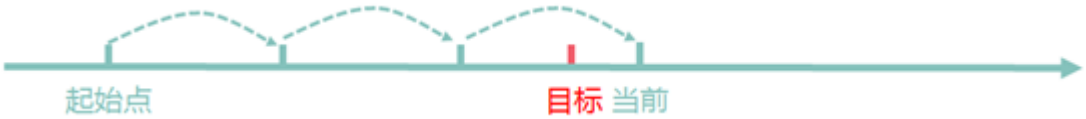
在解决了梯度下降方向和步长的两个问题之后，还有一个问题需要解决。可能初一看这个问题，大家会有点奇怪，难道不是下降到最低点时停止运算么？

显然不是。我们可以把在一个曲面上向最低点逼近的过程看做是在一个数轴上向一个点逼近：



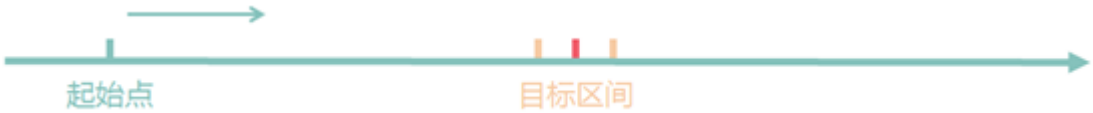
从这个起始点往目标按一定的距离逼近，跨出第一步之后，到达如下位置：

如此往复的往前走，最终你会发现居然错过了：



不过没事，我们可以调整一下出事的步长，然后再试一次。但结果显然是可以预料到的，你会再一次失败，无论你尝试多少次，你都几乎不可能正好某次移动后落到目标点上。

因此，要使这个移动能够停止下来，我们需要增加这个目标的范围，将一个点扩充为一个区间：



这样我们只需要随便调整一个步长，很容易就能够落到我们目标区间范围内：

如此一来，只需要这个目标区间的两端距离我们的实际目标中的误差是在我们可以接受的范围内，那么我们就可以近似的用这个解来作为我们的最终结果。

因此在实际梯度下降运算过程中，当误差值小于我们设定的某个阈值时，即停止运算，返回当前计算得到的系数。

梯度下降算法表示

回顾前面所介绍的梯度下降算法的过程，实际上就是一个计算系数的过程。在曲面上随机选一点，实际上就是随机选择一组初始的系数。

当沿某个方向下降一个步长时，实际上就是挨个计算下一组系数中每个系数的值的过程。我们使用下面的一个式子表示这个梯度下降的计算过程：

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

我们来看一下算法具体的逻辑。首先在初始状态下随机选择一组 (θ_a, θ_b)，然后开始梯度下降，首先沿着 θ_a 所在的轴往前跨一步：

$$\theta_0 = \theta_a - \alpha \frac{\partial}{\partial \theta_a} J(\theta_a, \theta_b)$$

然后就得到一个 θ_0 ，这个 θ_0 就是沿着 θ_a 轴的方向往前移动一步之后新的 θ_a ，同样的道理沿着 θ_b 所在方向也移动一个位置：

$$\theta_1 = \theta_b - \alpha \frac{\partial}{\partial \theta_b} J(\theta_a, \theta_b)$$

然后就得到了一组 (θ_0, θ_1) ，这组 (θ_0, θ_1) 就是移动之后在山坡上新的位置，并且由于他们都是从相同的位置开始移动的，因此式子中的函数 J 中所使用的都是相同的起点。

然后计算这组新的 θ 的偏差是否满足我们设置的阈值，如果满足那么就停止计算，返回这组 θ 即可，否则继续进行梯度下降。在重新开始梯度下降时，起始点的位置就发生变化了，因此新的一轮计算因该是将这组新的 θ 代入函数 J 中重新计算出一组 θ 。

到目前为止我们已经了解了梯度下降算法的主要逻辑，不过要用代码实现这个式子还有一个问题，就是这个式子中的偏导如何计算，我们接下来解决这个问题。

利用化简解决偏导

在前面介绍算法逻辑时，我们发现式子中存在一个求偏导的符号，在具体实现代码时可能会造成一些麻烦，下面我们看看如何解决这个问题。我们首先定义一下训练数据：

序号	特征	标签
1	x^1	y^1
2	x^2	y^2
……	……	……
i	x^i	y^i
……	……	……
N	x^N	y^N

这个训练集中共有一个特征，对应一个标签，上标表示第几组样本。特征中的下标签表示一个样本中的第几个特征。除此之外关于整个算法我们有三个式子：

$$\begin{cases} \theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) & amp;(1) \\ J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 & amp;(2) \\ h_\theta(x) = \theta_0 + \theta_1 x_1 & amp;(3) \end{cases}$$

式子（2）中的代价函数为了便于后续化简，我们损失函数取平均数的二分之一，这个对最终结果没有影响。我们要想办法利用这几个式子进行化简，看看能否有办法把求偏导的内容化简掉。不过在开始之前我们先要对数据集和式子进行一点小小的调整：

序号/特征	x_0	x_1	y
1	1	${x_1}^1$	y^1
2	1	${x_1}^2$	y^2
……	1	……	……
i	1	${x_1}^i$	y^i
……	1	……	……
m	1	${x_1}^m$	y^m

我们给训练集中的每条数据都增加一个特征 x_0 ，并将这个特征的值都设置为 1，这样我们就能够得到这样两个等价的式子：

$$\begin{cases} h_\theta(x) = \theta_0 + \theta_1 x_1 \\ h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 \end{cases}$$

这样对于第二个式子就可以使用求和公式表示，替换后的式子就变成了：

$$\begin{cases} \theta_j = \theta_j - \alpha \frac{\partial}{\partial\, \theta_j} J(\theta_0, \theta_1) & amp;(1) \\ J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 & amp;(2) \\ h_\theta(x) = \sum_{j=0}^{n} \theta_j x_j & amp;(3) \end{cases}$$

其中 $n = 2$，接下来就可以开始化简了，下面具体的化简过程只需了解，重点关注最终化简得到的结果。

首先将式子（2）代入式子（1）得到：

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial\, \theta_j} \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

根据求导运算法则，导数的和等于和的导数，因此可以对每一项求导：

$$\theta_j = \theta_j - \alpha \cdot 2 \cdot \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot \frac{\partial}{\partial\, \theta_j} (h_\theta(x^{(i)}) - y^{(i)})$$

然后将式子（3）代入这个式子中，得到：

$$\theta_j = \theta_j - \alpha \cdot \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot \frac{\partial}{\partial\, \theta_j} (\sum_{j=0}^{n} \theta_j {x_j}^{(i)} - y^{(i)})$$

接下来的重点就是对求偏导这部分化简，这里需要回顾一下偏导的概念，所谓偏导就是求沿某个轴所在平面曲线的导数，根据求偏导的定义，实际上就是使除了这个 θ_j 以外的所有维度都为零的情况下函数的导数。 ：

$$\frac{\partial}{\partial \theta_j}(\sum_{j=0}^{n}\theta_j x_j^{(i)} - y^{(i)})amp;= \frac{\partial}{\partial \theta_j}(\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \cdots + \theta_j x_j^{(i)} + \cdots + \theta_n x_n^{(i)} - y^{(i)})$$

$$amp;= \frac{\partial}{\partial \theta_j}(\theta_j x_j^{(i)} - y^{(i)})$$

$$amp;= x_j^{(i)}$$

将这个式子代回原式子中，得到：

$$\theta_j = \theta_j - \alpha \cdot \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

这样的一个式子用代码实现起来应该就没有什么障碍了。下面我们来看看如何实现这个算法。

一元线性回归的代码实现

代码框架

在一开始实现算法的时候，大家面对空白的代码编辑器可能会感觉无从下手，我们从零开始帮助大家一步步实现这个算法。

整个代码一共也就几十行，大家先要从心理上藐视它，不要害怕，然后我们依照前面的逻辑将整个框架搭建起来。首先我们要实现一个梯度下降算法，这个算法的核心逻辑是随机取一组初始系数Theta，然后与训练集的数据计算损失，如果损失符合我们的要求，就返回这组系数，否则就计算一组新的系数，再次判断损失是否符合，我们用注释的形式表达这个逻辑：

```
# @return float[] newtheta
def gradientDescent():
    # 1.计算当前系数的损失

    # 2.如果损失小于规定的阈值，则返回这组系数

    # 3.否则计算一组新的系数

    # 4.重复上述三个过程

    return newtheta
```

首先看这个逻辑中的步骤一，计算当前系数的损失，那么根据我们前面对损失函数的定义，实现计算损失的函数：

```
# 计算损失
# @param float [] theta
# @param float [][] x
# @param float[] y
# @return loss
def calculateLoss(theta, x, y):
    # coding
    # loss = ...
    return loss
```

在实现这个函数的过程中我们涉及到几个前置变量的定义。首先我们需要有一些训练集，这个训练集中包含一个特征和一个标签：

```
x = [2.1, 2.5, 3.9, 5.1, 2.7]
y = [4.2, 5.1, 7.75, 10.18, 5.35]
```

不过在之前化简时，我们假设给所有数据增加了一个值为1的特征，因此我们需要的训练集应该是：

```
x = [[1, 2.1], [1, 2.5], [1, 3.9], [1, 5.1], [1, 2.7]]
y = [4.2, 5.1, 7.75, 10.18, 5.35]
```

注意，这里的这些点是随意写的。有了训练集之后，我们可以将其作为参数传入 `calculateLoss()` 函数。与此同时，`gradientDescent()` 函数也就需要传入这个参数。同样的道理还需要声明一组初始化的系数，比如：

```
theta = [5, 1]
```

注意，这里声明的这几个变量都应该是全局变量，作为参数传入这些函数使用。

在计算损失时，我们需要计算每个样本的估计值，也就是函数 $h(\theta)$ ，这里我打算单独用一个函数实现这个过程，计算假设函数的值需要有参与运算的特征以及函数的系数：

```
# 计算假设
# @param float [] theta
# @param float [] x
# @return y
def calculateHypothesis(theta, x):
    # coding
    # y = ...
    return y
```

然后第二部分是比较阈值与损失，那么又需要定义一个全局变量：

```
threshould = 0.01
```

然后到第三步，需要计算一组新的系数的函数：

```
# 计算系数
# @param float[] theta
# @param float[][] x
# @param float[] y
# @param float alpha
# @return float[] theta
def calculateTheta(theta, x, y, alpha):
    # coding
    # theta = ...
    return theta
```

这里又有一个全局变量，步长：

```
alpha = 0.001
```

为了简单起见，在这个算法的实现中我们使用一个固定的步长参与计算。到第四步的话，使用一个循环将上面的逻辑套在里面即可，那么完整的代码框架应该就是：

```
x = [[1, 2.1], [1, 2.5], [1, 3.9], [1, 5.1], [1, 2.7]]
y = [4.2, 5.1, 7.75, 10.18, 5.35]

theta = [5, 1]
threshould = 0.01
alpha = 0.001

# @param float[] theta
# @param float[][] x
# @param float[] y
# @param float alpha
# @param float threshould
# @return float[] theta
def gradientDescent(theta, x, y, alpha, threshould):

    while True:
        # 1.计算当前系数的损失
        dev = calculateLoss(theta, x, y)

        # 2.如果损失小于规定的阈值，则返回这组系数
        if dev < threshould:
            return theta

        # 3.否则计算一组新的系数
```

```
        theta = calculateTheta(theta, x, y, alpha)

    return theta

# 计算损失
# @param float[] theta
# @param float[][] x
# @param float[] y
# @return loss
def calculateLoss(theta, x, y):
    # coding
    # loss = ...
    return loss

# 计算假设
# @param float[] theta
# @param float[] x
# @return y
def calculateHypothesis(theta, x):
    # coding
    # y = ...
    return y

# 计算系数
# @param float[] theta
# @param float[][] x
# @param float[] y
# @param float alpha
# @return float[] theta
def calculateTheta(theta, x, y, alpha):
    # coding
    # theta = ...
    return theta
```

多元线性回归

截止目前为止，我们已经学习了一元线性回归算法的基本原理以及基于梯度下降算法的实现过程。但一元线性回归只是一种非常简单的形式，其中的特征数量只有一个，而在实际的问题场景中特征数量甚至有数百万上千万个，我们来看一下多元线性回归于一元线性回归的区别。

一元线性回归的假设函数

$$h_\theta(x) = \theta_0 + \theta_1 x_1$$

多元线性回归的假设函数

$$h_\theta(x_1, x_2, \cdots, x_n) = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

除此以外，在算法实现上两者基本没有什么区别。因此在实现之前代码的基础上，我们只需要修改计算假设函数的函数。

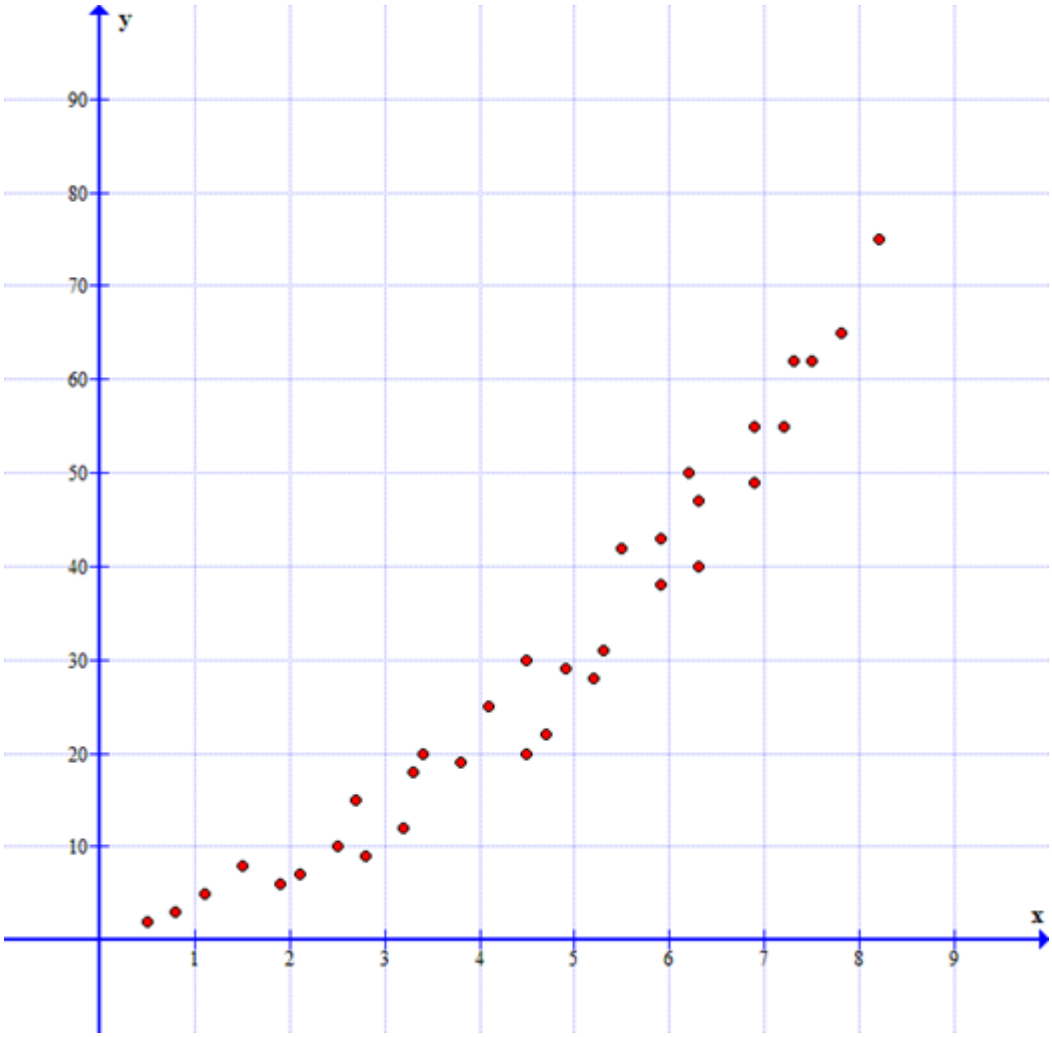
多项式回归

我们先来看一个例子：

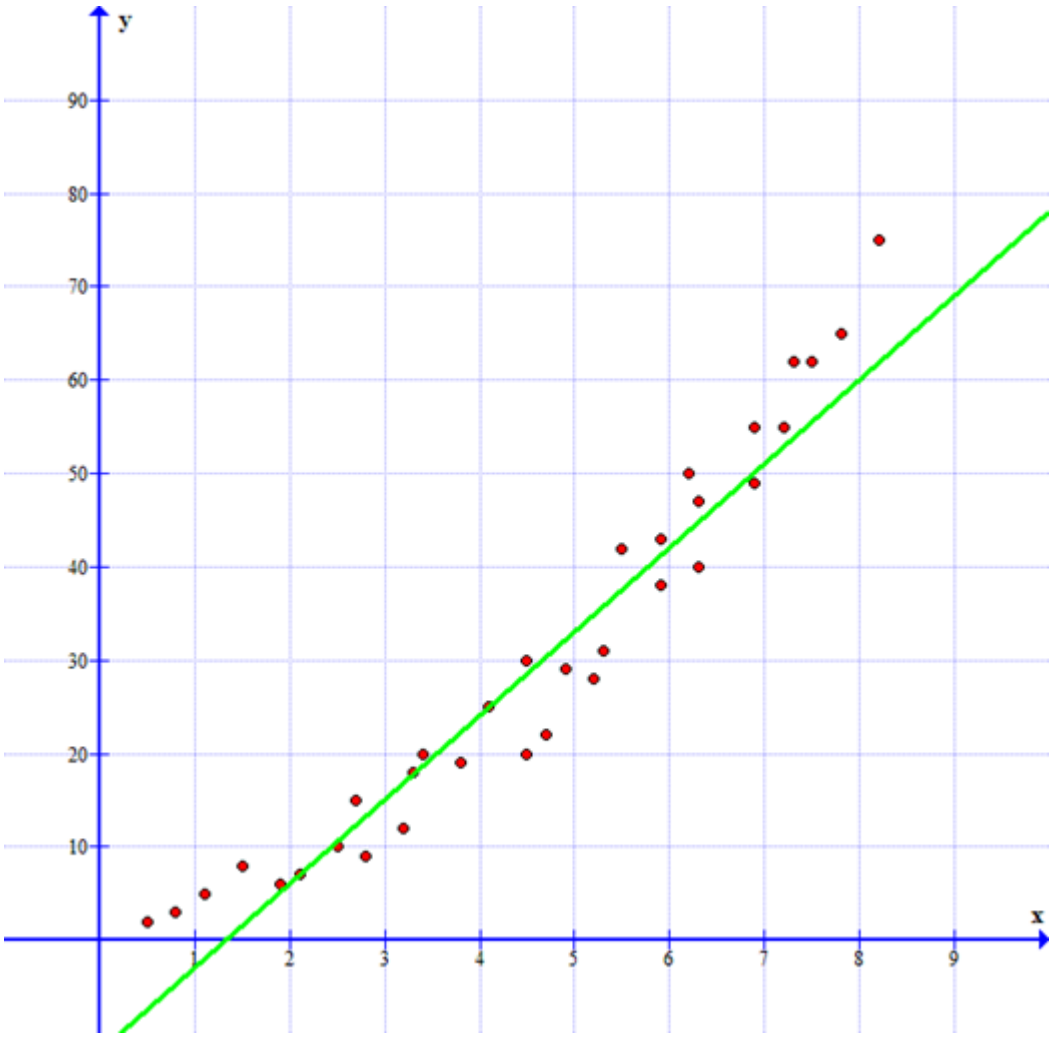
$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x^2$$

大家注意观察这个式子，其中有一个项的次数是二次，我们把含有一个或多个特征，并且存在二次和二次以上项的回归称为多项式回归。

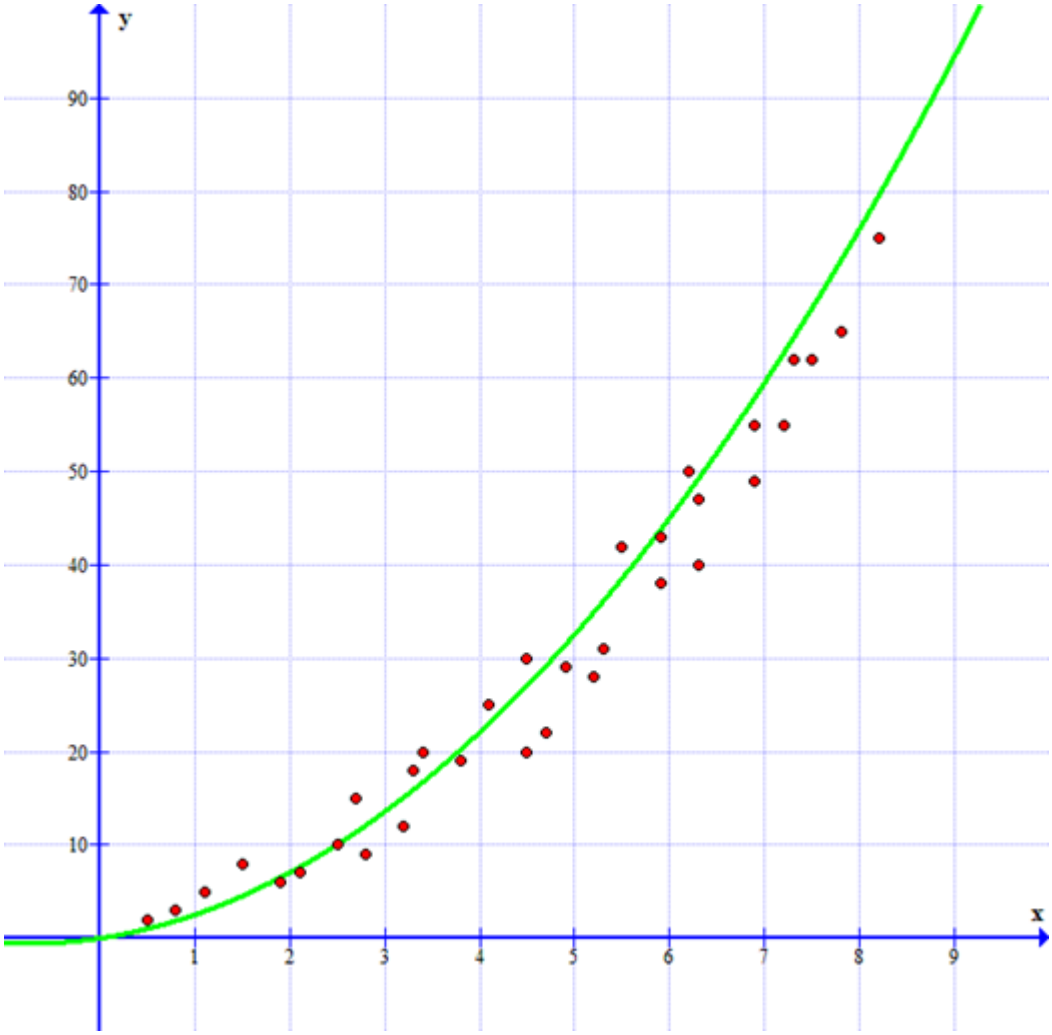
我们来看一个案例，这是一组匀加速运动过程实验数据，横坐标表示的是时间，纵坐标表示的是距离。我们的目标是预测在这个匀加速运动持续的过程中，未来某个时间点，运动的距离。



因为是匀加速运动，这里涉及到的变量只有时间，因此我们对时间和距离进行建模，我们采用之前的方法得到一个结果，大家觉得这个模型的预测结果如何？



如果大家还记得我们初中学过的匀加速运动过程时间和距离的关系的话，会很明确的给出答复，这个预测结果肯定有较大偏差，为什么？根据我们已知的背景，匀加速运动中距离和时间的关系是非线性的。



因此，对于这种情况，我们就要调整模型，改成这种非线性关系的模型。多项式回归可以用来处理非线性关系。

对于具体多项式回归的内容属于一个比较大的主题，这里不展开，有兴趣的同学可以参加相关著作。

矩阵表示

在很多符号表示和代码实现中都经常使用向量的表示方法来表示线性回归中的运算。对于多元线性回归的模型我们一般用表达式：

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

不过为了便于表示和计算，我们会给其增加一个值为1的特征，将式子改写为：

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n$$

如此一来，这个表达式其实正好相当于两个矩阵相乘：

$$\theta = [\theta_0 \quad \theta_1 \quad \theta_2 \quad \cdots \quad \theta_n] \times X^T = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix}$$

这样模型的表达式就变成了两个矩阵相乘：

$$h_\theta(x) = X^T\theta$$

基于 SkLearn 的实践

下面我们结合Pandas、Seaborn以及skLearn来对一个联合发电厂的发电效能数据进行处理。

数据来源：Combined Cycle Power Plant Data Set

使用Pandas准备数据

首先导入Pandas的包，然后读取数据：

```
import pandas as pd
data = pd.read_csv("CombinedCyclePowerPlant.csv")
```

数据集是一个CSV文件，位于代码同一目录下。使用代码测试一下，看看是否正确加载数据：

```
print(data.head())
```

如果正确加载数据，那么就会在控制台中打印输出如下内容：

	AT	V	AP	RH	PE
0	8.34	40.77	1010.84	90.01	480.48
1	23.64	58.49	1011.40	74.20	445.75
2	29.74	56.90	1007.15	41.91	438.76
3	19.07	49.69	1007.22	76.79	453.09
4	11.80	40.66	1017.13	97.20	464.43

这是数据集中的前五条数据。我们来查看下数据集中包含的数据条数：

```
print(data.shape)
```

会打印出“(9568, 5)”，说明这个数据集中有9568条数据。

使用Seaborn分析数据

因为我们打算使用线性回归来处理这个数据集，因此我们首先要了解下数据集中的每个特征与结果的一个关系，一般来说给定的原始数据集中特征并不是都和结果呈很强的线性关系的，我们需要通过逐个分析的方式来挑选合适的特征。

首先导入必要的包：

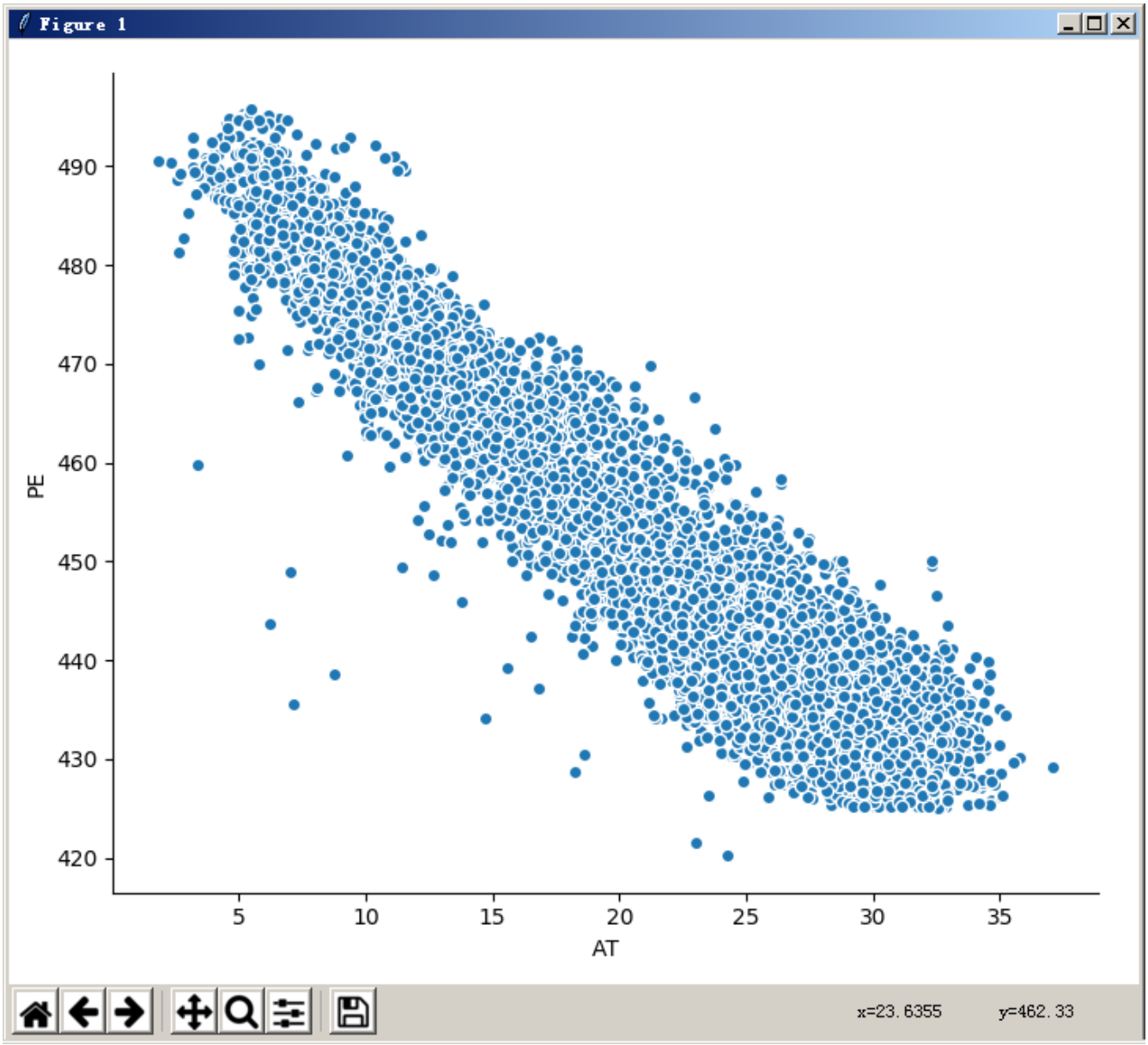
```
import seaborn as sb
import matplotlib.pyplot as plt
```

其中seaborn就是我们用来使用图形化的方式进行数据分析的库，而这个库是基于matplotlib进行作图的，因此也要导入matplotlib。

然后使用如下代码首先绘制“AT”这个特征与结果的关系：

```
sb.pairplot(data, x_vars=["AT"], y_vars='PE', size=6, aspect=1.2)
plt.show()
```

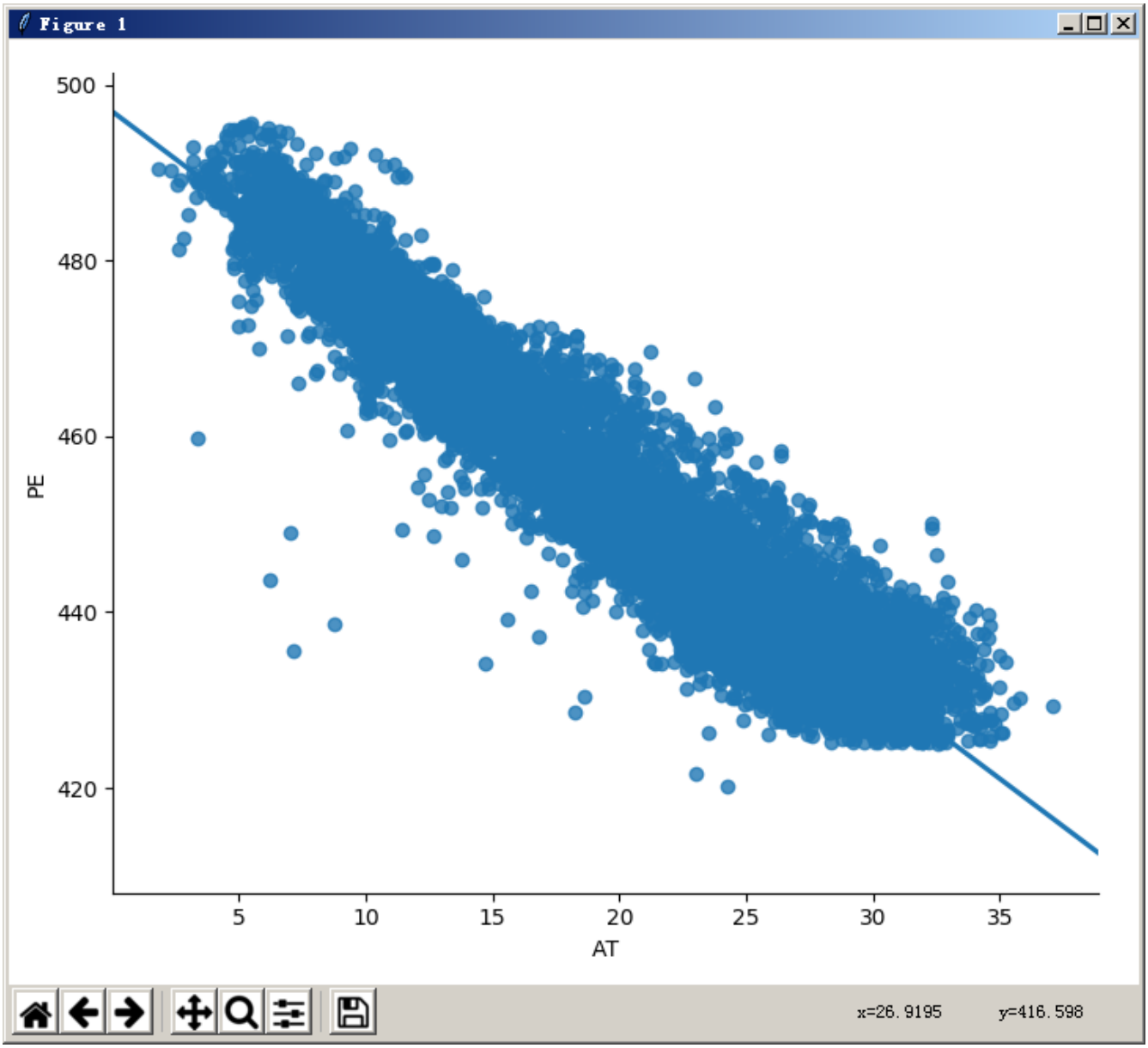
在上述的代码中，调用pairplot()函数来绘制关系图，其中这个函数接收的x_vars是需要绘制的特征名称，而y_vars则表明结果的列名，后面的size和aspect则是设置生成的图形尺寸。运行后会生成如下的图形：



这个图形是一个二维坐标系，其x轴就是我们选定的特征，而y轴就是结果。从这个图中我们可以看出来数据的分布呈现一个很强的线性关系。seaborn中还还可以为这个分布添加一条拟合直线，通过增加一个属性：

```
sb.pairplot(data, x_vars=["AT"], y_vars='PE', size=6, aspect=1.2, kind='reg')
```

绘制的图形中就增加了一条最佳拟合的直线：



采用相同的方式我们可以对其余的几个特征值一一分析，查看与结果的关系。

使用sklearn训练模型

拆分特征和标签

接下来我们利用sklearn中的线性回归算法来构建模型。首先将特征和标签拆分成开来：

```
x = data[["AT","V","AP","RH"]]
y = data["PE"]
```

拆分训练集和测试集

利用列名，结合Pandas中的DataFrame可以很方便的拆分数据。然后利用sklearn中的训练集和测试集拆分函数将数据集进行拆分：

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y, random_state=1)
```

可以分别查看一下两个数据集中的数据情况：

```
print(x_train.shape)
print(x_test.shape)
```

训练模型

准备好数据之后，就可以使用sklearn中的线性回归算法来构造模型了：

```
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(x_train, y_train)
```

上述代码中调用sklearn中的线性回归算法，来训练模型，我们可以使用下面的两个属性查看训练的结果：

```
print(lr.intercept_)
# 447.062970987

print(lr.coef_)
# [-1.97376045 -0.23229086  0.0693515  -0.15806957]
```

这两个属性分别表示 θ_0 和 $\theta_{1\sim n}$。

测试模型

对于分类模型来说，测试模型的方法比较简单只管，只需要将测试集中的特征，利用模型运算之后的结果与测试集中的标签对比，查看匹配程度即可。而与回归问题来说，由于其结果是连续的，无法使用直接对比的方式来判断，因此这里引入两个概念**MSE(Mean Square Error)均方误差**和**RMSE(Root Mean Square error)均方根误差**。

其中均方误差的表达式为：

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

而均方根误差则是对均方误差开平方：$RMSE = \sqrt{MSE}$。根据这个思路，我们可以评价我们的模型了，首先利用模型对测试集进行计算：

```
y_pred = lr.predict(x_test)
```

然后使用sklearn中带的均方误差函数进行评价：

```
import numpy as np
from sklearn import metrics

print(np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
```

然后我们可以通过增加和删除用于训练模型的特征来对比每个模型的评价情况，选取最优的那个组合作为预测模型。