单变量线性回归：

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

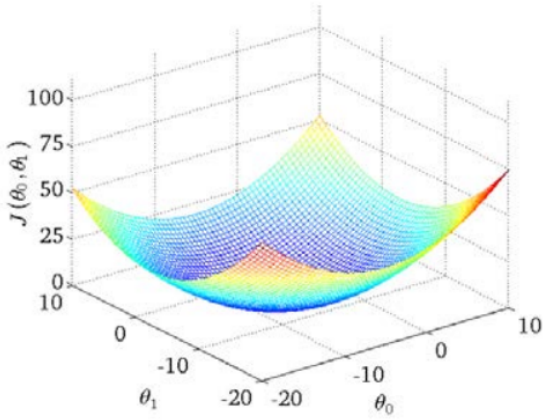
```
def hx(x, w, b):
#     return: [num,1]
    return np.dot(x,w) + b
```

损失函数：

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

（以2m为分母只是为了求导更方便）

```
def loss(x, y, y_hat,w, b, num):
    return np.sum((y_hat-y)**2)/num
```



绘制上诉三维图像，可以发现存在一个最小点，以及很多等高线。

通俗来讲，损失函数就代表了预测值与真实值的位置的偏差，最后除以m相当于标准化。若能找到最小点，则代表预测的误差最小。

但是，想要直接获取到最小值是很困难的，所以要采取一定方法求得最小值。

Gradient Descent(梯度下降)是一个求函数最小值的算法。

原理：

一开始随机初始参数，然后以非常小的步调更新参数，就会接近最低点。

Gradient descent algorithm

repeat until convergence {

$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ (for $j = 0$ and $j = 1$)

}

Correct: Simultaneous update

$\text{temp0} := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$

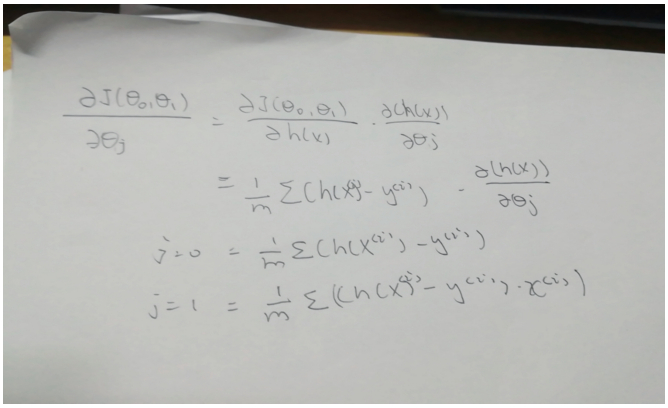
$\text{temp1} := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$

$\theta_0 := \text{temp0}$

$\theta_1 := \text{temp1}$

其中 α 对应的是学习率，代表学习的快慢.

损失函数求导过程：



```
def linear_loss(X, y, w, b):
    num_train = X.shape[0]
    num_feature = X.shape[1]
    y_hat = np.dot(X, w) + b
    loss = np.sum((y_hat-y)**2)/num_train
    dw = np.dot(X.T, (y_hat-y))/num_train
    db = np.sum((y_hat-y))/num_train
    return y_hat, loss, dw, db
```

接下来只需反复更新就能求得最小值了。

如果显示的采用一个数一个数的计算，会使得复杂度非常大，若是吧其看作矩阵运算，相对来讲，速度会得到显著的提升。

上述只展示了一维的数据的线性回归，但是现实中数据往往是多维的，就需要把一维扩展到多维，我们通常以行代表数据量，列代表特征，这样就形成了一个(量，特征)的矩阵，对于不同 θ ，只需用不同列的数据进行更新操作就行了。

经过反复迭代更新后，即可得到结果。

但是我们在使用算法的过程中，往往不会直接对着一个数据集反复训练，因为这样使得模型的拟合效果不好。因为只要不断的训练下去，模型对这个数据集的拟合效果越来越好，但是却对外部的数据集拟合效果很差，因为我们采取的数据集只是真实世界中的一部分，如果我

们用于训练的数据集和真实的数据集的分布不相同，就会使得拟合的效果非常差，即偏差太大，这样就是过拟合。但是如果在训练刚开始就结束，那么这个模型对数据集的拟合效果不是很好，即方差太大，这就是欠拟合。我们需要做的就是找到一个适合点，使得过拟合和欠拟合达到一个平衡点。

从数据的角度看，我们可以通过将数据集划分为训练集，测试集来进行交叉验证，通过不断的观察每次训练的训练集与测试集的差距，判断模型的效果，这一部分主要依靠经验，当然，最简单的方法就是增大数据集。
从模型的角度上来看，我们可以采用正则化策略，即给损失函数加上一个惩罚项，因为我们在训练时， θ 的更新主要依赖于输入的x、y，虽然 θ 参与到了y的预测当中去，但是经过各式各样的操作后，所包含参数的信息就非常少了，如果加上一个惩罚项后，参数的更新会多出一些变化，如果 θ 过大，则对应的更新程度就越大，反之则相反，这样可以将限制参数更新的速度。

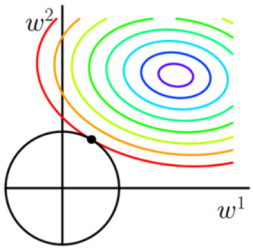
常见的正则化如下：

L2正则化（岭回归）：

损失函数：

$$J = J_0 + \lambda \sum_{w} w^2$$

相较于普通的损失函数，采用了二次惩罚项。这样损失函数就成了求两项之和的最小值，换言之，只有两项同时最小，才能求得最小值，也就是，后一项就是对前一项的约束。对于累加的平方和来讲，跟以前学到的圆的公式很像，这样的话，问题就明显的变为了求圆与等值线的交点。对应的2维平面图如下：



这样一来，一定程度上防止了过拟合。

L1正则化（Lasso回归）：

损失函数：

$$J = J_0 + \lambda(|w_1| + |w_2|)$$

与L2正则化不一样的是，该项采取了绝对值项进行的约束，这个明显就是一个菱形，与等值线的交点容易在坐标轴上，使得部分参数变为0，从而转化为稀疏矩阵，这样更容易提取到主要特征。

ElasticNet回归：

上述两种的结合体。

损失函数：

$$min(\frac{1}{2m}[\sum_{i=1}^{m}(y_i{}' - y_i)^2 + \lambda\sum_{j=1}^{n}\theta_j^2] + \lambda\sum_{j=1}^{n}|\theta|)$$

综合了两种的优点。