实验二：线性回归及梯度下降算法的实现

杨扬　计算机科学与技术　1611132

摘　要

本次实验是对线性回归的学习及实现，所使用的数据集是著名的波士顿房价预测数据集。本文档针对于此问题，分析数据集本身的特性，介绍了线性回归及各种梯度下降算法的原理，结合于实验数据集本身设计了算法，并基于源程序对实验的设计进行了解析，最后对实验的结果进行了分析。简单来说，本文档在对实验所有要求均做出实现的同时，也成功地进行了多方面的拓展，以极高的质量完成了此次实验。

关键词：房价预测，线性回归，梯度下降

目录

1 问题描述

线性回归（Linear regression）是一个统计学中的概念。简单来说，线性回归是利用称为线性回归方程的最小二乘函数对一个或多个自变量和因变量之间关系进行建模的一种回归分析。这种函数是一个或多个称为回归系数的模型参数的线性组合。只有一个自变量的情况称为简单回归，大于一个自变量情况的叫做多元回归。

线性回归是机器学习领域应用最广泛的算法之一，也是我们后续学习更复杂算法的基石，而在本次实验中，我们将使用包括梯度下降、随机梯度下降等多种最优化算法对线性回归进行实现，并分析其原理及实现的思路。

本次实验的数据集是著名的 Boston housing 数据集，来源于上世纪波士顿的房价市场，旨在解决房价的预测问题。具体来说，数据集为一个 *housing.data* 的文本文件，数据的存储形式如图 1 所示，文件中包含 506 行、14 列的浮点数字。其中各列数据的含义如图 2 所示，其中前十三列为影响房价变化的特征，最后一列为房价的数额。此次实验需要我们基于前十三列的特征构造线性回归分类器，对房价作出预测，并与最后一列的实际房价进行比较，并计算出 RMSE。

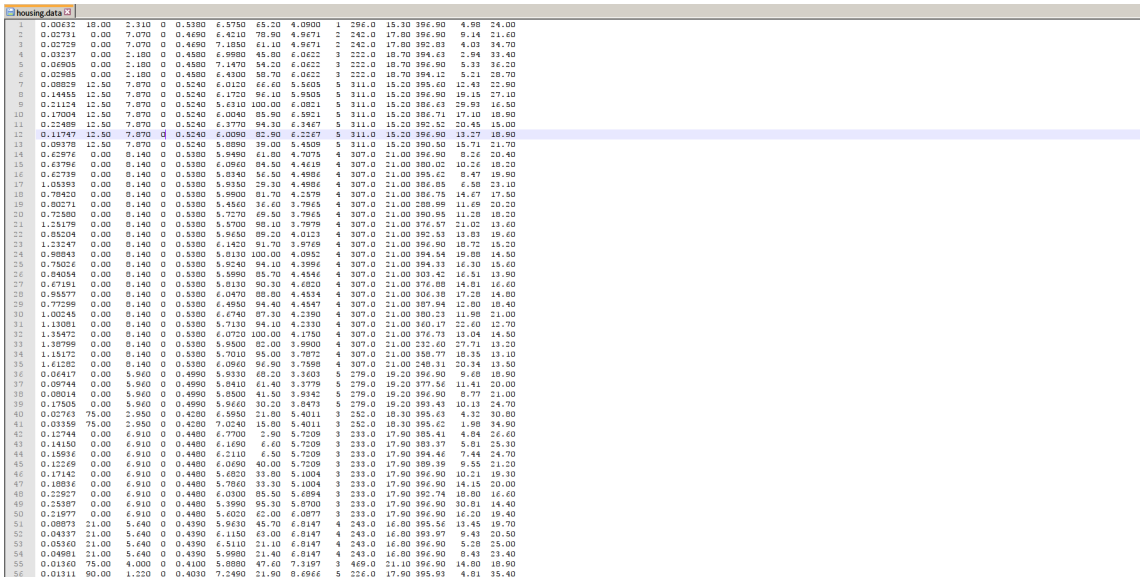

图 1: 数据存储形式

1. CRIM - per capita crime rate by town
2. ZN - proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS - proportion of non-retail business acres per town.
4. CHAS - Charles River dummy variable (1 if tract bounds river; 0 otherwise)
5. NOX - nitric oxides concentration (parts per 10 million)
6. RM - average number of rooms per dwelling
7. AGE - proportion of owner-occupied units built prior to 1940
8. DIS - weighted distances to five Boston employment centres
9. RAD - index of accessibility to radial highways
10. TAX - full-value property-tax rate per $10,000
11. PTRATIO - pupil-teacher ratio by town
12. B - 1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
13. LSTAT - % lower status of the population
14. MEDV - Median value of owner-occupied homes in $1000's

图 2: 数据中各列的含义

2 梯度下降算法

不难看出，此次实验的问题是一个多变量回归问题。针对于此问题，我们将建立线性回归分类器，并使用梯度下降算法来解决。

梯度下降算法的原理非常简单，其思想可以类比为一次下山的过程。我们可以想象这样一个场景：一个人需要从山上往山下走，即需要找到山的最低点，也就是山谷。如果他想要找到一条最短的下山路径，他便可以采用梯度下降算法，具体来说，他以他当前的所处的位置为基准，向四周观察，寻找这个位置最陡峭的地方，然后朝着山的高度下降的地方走，如此迭代下去，最后便能成功的抵达山谷。

梯度下降与下山的过程别无二样，但此时的人变成了线性回归问题中我们需要进行优化的分类器。以本次实验为例，我们将前十三列的所代表的数据输入定义为 x，将权值定义为 θ，将最后一列的所代表的房价定义为 y，则可定义线性回归方程，即目标函数为：

$$h_{(\theta)} = \theta^T x$$

为使用梯度下降算法来使 $h_{(\theta)}$ 的值尽可能的拟合 y，则我们需要定义一个代价函数，本实验中我所采用的代价函数为：

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

此代价函数公式中，m 的含义为数据的行数，本实验中为 506。基于以上公

式，我们则可以求出梯度为：

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

在明确了代价函数和梯度函数之后，我们便可以开始进行梯度下降算法的执行了，对于每一次新的训练数据的输入，我们采用如下的公式对 θ 的值进行更新：

$$\theta^i = \theta^{i-1} - lr * \nabla J(\theta) \quad \text{started at } \theta^1$$

其中 lr 的值为一个常数，在梯度下降中被称作学习率 (Learning rate) 或步长，可根据训练的经验进行调整，在将所有的训练数据输入后，θ 的值便会更新为使代价最低的最优值。

2.1 L1 正则化与 L2 正则化

众所周知，机器学习领域面临的一大难题便是过拟合问题，很多时候，由于我们所定义的目标函数过于复杂，从而导致模型过于复杂，刻意地去拟合了训练集中的数据点，但却在测试集中表现较差。

线性回归问题同样也面临着过拟合的困扰，而 L1 正则化和 L2 正则化即是两种解决过拟合问题的方法，简单来说，其思想是通过在代价函数中增加与权值 θ 有关的正则化项，让 θ 在梯度下降过程中受到一定程度的惩罚，最后得到的 θ 值也不过于大，从而使得最后得到的目标函数不过于复杂，保证了模型的普适性。

具体来说，L1 正则化后的代价函数为：

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \alpha\|\theta\|$$

其中 $\|\theta\|$ 代表 θ 的绝对值。

L2 正则项与 L1 有所区别，其正则化后的代价函数为：

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \alpha\|\theta\|^2$$

以上两个公式中的 α 都是一个常数，可根据经验进行设定。由于理论课上已经对两种正则化方法为什么能降低模型复杂性的原因进行了详细的数学推导，这里我们也不再赘述。

2.2 随机梯度下降

从前文的叙述可以看出，一次迭代中，梯度下降算法需要将所有的训练数据进行输入，才能进行一次 θ 值的更新，不得不说这样做的时间代价是极高的，因此我们也需要寻找更高效的算法。

随机梯度下降算法 (Stochastic Gradient Descent, SGD) 即是这样一种算法，此算法优化的不是在全部训练数据上的代价函数，而是在每轮迭代中，随机优化某一条训练数据上的代价函数，这样每一轮 θ 的更新速度便大大加快。在机器学习领域，随机梯度下降算法也被称为” 在线学习”。

至此，本次实验中所有算法的理论基础便介绍完毕。

3 实验及源程序分析

3.1 实验环境

本次实验在 Ubuntu 18.04 下进行，实验所使用的编程语言为 Python 3.6，IDE 为 Pycharm 2018.3.2，主要使用了 numpy、matplotlib 等 Python 包。

3.2 源程序分析

由于此次实验的内容较少，实验数据规模也不算太大，所以我们在一个 py 文件里即可完成本次实验，具体来说，程序的步骤如下所示：

首先我们在 Python 中导入所需要用到的 Python 包：

```
import numpy as np
import matplotlib.pyplot as plt
```

然后我们需要导入相应的实验数据，*housing.data* 文件中数据的分割方式为空格和换行符，如第一次实验一般，我们采用 numpy 中的 loadtxt 方法就可以了：

```python
def read_data(path):
    data = np.loadtxt(path)
    label = data[:, -1]
    data = data[:, :-1]
    return data, label
```

简单看一下导入的数据，我们便可以发现各列数据间的大小分布极为不均，一些数值较大的数据列可能会对模型的参数造成影响，因此我们需要进行简单的数据预处理，将数据的数值统一缩放到一个合理的区间，这里为了让离群点 (outliers) 不对数值区间产生影响，我们采用数据标准化 (Data Normalization) 的方法来进行处理，其计算公式为 $x^* = \frac{x-\mu}{\sigma}$，其中 μ 为各列数据的列均值，σ 为列标准差，程序设计如下：

```python
def data_normalization(data):
    data_normed = data
    for i in range(data.shape[1]):
        cur_col = data[:, i]
        mean = np.mean(cur_col)
        std = np.std(cur_col)
        for j in range(cur_col.shape[0]):
            data_normed[j, i] = (data[j, i] - mean) / std
    return data_normed
```

本次实验要求我们用留一法 (Leave One Out) 进行训练集和测试集的分割，程序设计如下：

```python
def loo_split(data, label, idx):
    train_data = []
    test_data = []
    train_label = []
    test_label = []
    for i in range(0, idx):
        train_data.append(data[i])
        train_label.append(label[i])
    test_data.append(data[idx])
    test_label.append(label[idx])
    for i in range(idx+1, data.shape[0]):
        train_data.append(data[i])
        train_label.append(label[i])
    return np.array(train_data), np.array(train_label), \
           np.array(test_data), np.array(test_label)
```

至此，数据的预处理及测试集划分的部分便已完成，接下来我们进入到梯度

下降算法的核心部分，首先参照前文的公式，设计梯度函数，此梯度函数中已经包括了是否使用随机梯度下降的参数：

```
def gradient(weight, x, y, random=False):
    if random is False:
        error = np.dot(x, weight) - y
        return abs(error.mean()), (1/x.shape[0]) * np.dot(np.transpose(x), error)
    else:
        rand = np.random.randint(0, x.shape[0] - 1)
        x_rand = x[rand, :]
        y_rand = y[rand]
        error = np.dot(x_rand, weight) - y_rand
        return abs(error.mean()), (1/x.shape[0]) * np.dot(x_rand, error)
```

在完成梯度函数的计算后，我们便可以开始梯度下降的流程了，这里我们采用用户指定迭代次数的方式来进行迭代训练，同时值得注意的是，我们在训练数据矩阵中加了值全为 1 的一列，大小变为 506×14，同时 θ 的维度也设为 14 维，这样做的目的是使新增的一列与权值 θ 中的 θ_0 相乘，代表线性方程中的常数项：

```
def gradient_descent(data, label, lr, epochs, reg=None, alpha=None, random=False):
    weight = np.zeros((14))
    x = np.hstack((np.ones((data.shape[0], 1)), data))
    y = label
    errors = []
    if reg is None:
        for epoch in range(epochs):
            error, grad = gradient(weight, x, y, random)
            weight = weight - lr * grad
            errors.append(error)
    elif reg is 'L1':
        for epoch in range(epochs):
            error, grad = gradient_L1(weight, x, y, alpha, random)
            weight = weight - lr * grad
            errors.append(error)
    else:
        for epoch in range(epochs):
            error, grad = gradient_L2(weight, x, y, alpha, random)
            weight = weight - lr * grad
            errors.append(error)
    return errors, weight
```

然后我们定义一个 *linear_regression* 的函数，作为用户调用我们程序的接口，同时在此函数中进行留一法的划分，将每条数据都预测一次，最后计算出所有预测值的均方根误差 (Root Mean Square Error, RMSE) 并返回：

```python
def linear_regression(data, label, lr, epochs, reg=None, alpha=None, random=False):
    data = data_normalization(data)
    preds = []
    for i in range(data.shape[0]):
        train_data, train_label, test_data, test_label = loo_split(data, label, i)
        errors, weight = gradient_descent(train_data, train_label, lr, epochs, reg, alpha, random)
        test_x = np.hstack((np.array([1]).reshape((1, 1)), test_data))
        pred_y = np.dot(test_x, weight)
        preds.append(pred_y)
        print(i, 'loo')
    preds = np.array(preds).reshape(label.shape)
    rmse = np.sqrt(((preds - label) ** 2).mean())
    return preds, rmse, errors
```

3.3 L1 正则化与 L2 正则化源程序分析

前文的程序段中便可以看出，我们已经在 *gradient_descent* 函数中添加了正则化方式的选项，这里我们参照第二节中的公式，正则化项的倒数可计算得知为：

$$sign(x) \quad \text{if } x > 0 \text{ then } sign(x) = 1 \text{ else } sign(x) = -1$$

则定义 L1 正则化后的梯度函数为：

```python
def gradient_L1(weight, x, y, alpha, random=False):
    if random is False:
        error = np.dot(x, weight) - y
        return abs(error.mean()), (1/x.shape[0]) * (np.dot(np.transpose(x), error)
                                                    + alpha * np.sign(weight))
    else:
        rand = np.random.randint(0, x.shape[0] - 1)
        x_rand = x[rand, :]
        y_rand = y[rand]
        error = np.dot(x_rand, weight) - y_rand
        return abs(error.mean()), (1/x.shape[0]) * (np.dot(x_rand, error) + alpha * np.sign(weight))
```

L2 正则化后的梯度函数为：

```python
def gradient_L2(weight, x, y, alpha, random=False):
    if random is False:
        error = np.dot(x, weight) - y
        return abs(error.mean()), (1/x.shape[0]) * (np.dot(np.transpose(x), error) + alpha * weight)
    else:
        rand = np.random.randint(0, x.shape[0] - 1)
        x_rand = x[rand, :]
        y_rand = y[rand]
        error = np.dot(x_rand, weight) - y_rand
        return abs(error.mean()), (1/x.shape[0]) * (np.dot(x_rand, error) + alpha * weight)
```

3.4 随机梯度下降源程序分析

前文提到，随机梯度下降算法的原理是随机选取一条数据来对 θ 进行更新，这里我们采用 numpy 中的随机数生成函数来生成即可，以不加正则化项的梯度函数为例，加入随机选项后的梯度函数为：

```
def gradient(weight, x, y, random=False):
    if random is False:
        error = np.dot(x, weight) - y
        return abs(error.mean()), (1/x.shape[0]) * np.dot(np.transpose(x), error)
    else:
        rand = np.random.randint(0, x.shape[0] - 1)
        x_rand = x[rand, :]
        y_rand = y[rand]
        error = np.dot(x_rand, weight) - y_rand
        return abs(error.mean()), (1/x.shape[0]) * np.dot(x_rand, error)
```

至此，我们完成了所有算法的程序设计部分。

4 实验结果分析

基于经验，实验中我设置的学习率 lr 为 0.1，迭代次数 $epochs$ 为 1000，正则化项系数 α 为 0.1，得到梯度下降、L1 正则化梯度下降、L2 正则化梯度下降及其对应的随机梯度下降形式的 RMSE 值及其时间代价为：

	梯度下降	L1 正则化	L2 正则化
RMSE	4.87090	4.87079	4.87085
Time cost (s)	15.35	15.35	15.13
	随机梯度下降	L1 正则化 + 随机	L2 正则化 + 随机
RMSE	19.7512	19.7910	19.8000
Time cost (s)	8.97	10.26	9.88

表 1: 6 种形式的梯度下降算法的 RMSE 值与时间代价

从上表可以看出，在进行完全梯度下降时，L1 正则化和 L2 正则化均在一定程度上改善了模型的泛化能力，RMSE 值有所降低，这点与我们的预期一致，但其改善较小，我认为这是由于我们所采用的是留一法进行数据集划分的缘故，与只有 1 条数据的测试集相比，验证集的数据量高达 505 条。因此，在这种训练数

据较多的情况下，不管有没有经过正则化，模型的泛化能力都已经到了相当不错的程度，改善较少也就可以理解了。

此外，从运行时间上看，随机梯度下降优于完全梯度下降，但在精度方面，随机梯度下降的 RMSE 值远高于完全梯度下降，为找到这种情况的出现原因，我在编写程序时记录了两种算法每次迭代的 loss 值，并绘制 100 次迭代次数以内 loss 值随迭代次数的变化曲线图如下：

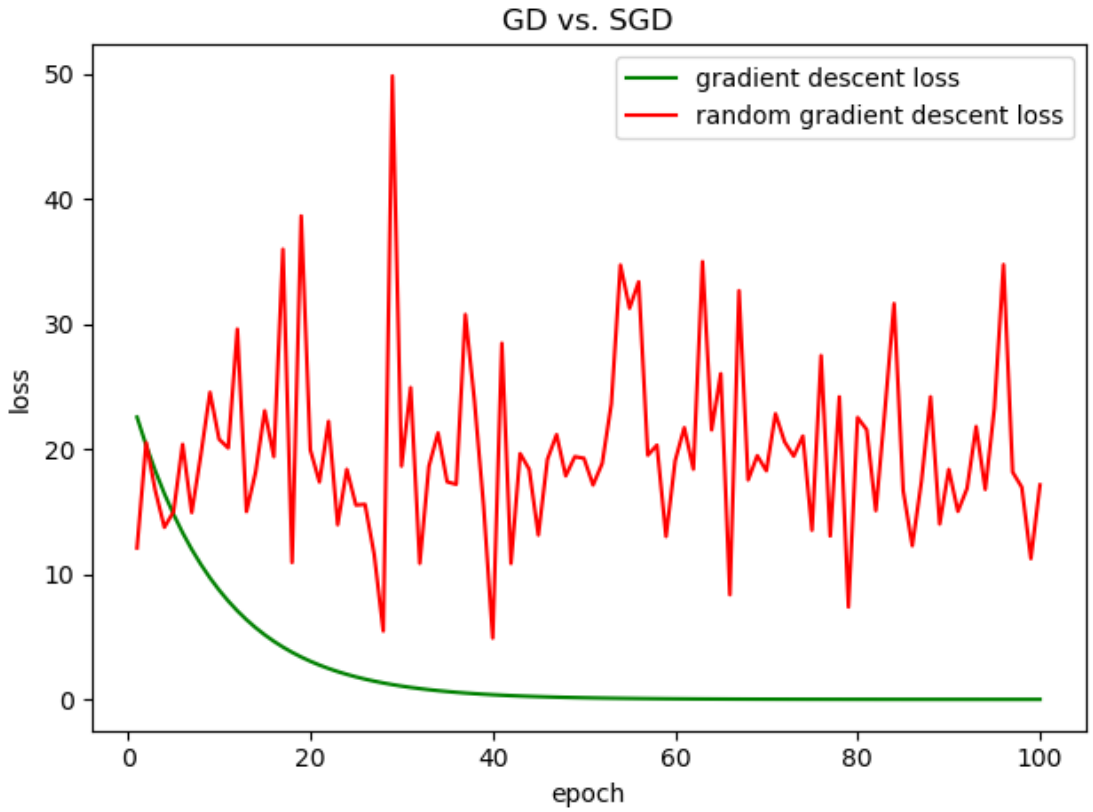

图 3: 梯度下降与随机梯度下降的 loss 值变化曲线图

由图 3 可以看出，相比于随机梯度下降的 loss 值变化极为不稳定，每次迭代都发生了剧烈的抖动，而其背后原因即在于随机梯度下降算法的本身的性质：随机梯度下降算法每次更新权值 θ 只选择一条训练数据，虽然这样做带来了更快的计算速度，但在某一条数据上代价函数更小并不代表在全部数据上的代价函数更小，甚至在全局上变得更遭，因此使用随机梯度下降优化得到的目标函数极有可能无法达到全局最优。

5 总结

线性回归是一个经典的问题，梯度下降更是一个极其优美的算法。通过此次实验，我更深层次地领略了机器学习的无穷魅力，并在实现各种梯度下降算法及程序运行结果分析的过程中感受到了理论学习与工程实践相结合的重要性。

参考文献

[1] Thomas M. Mitchell. 1997. Machine Learning (1 ed.). McGraw-Hill, Inc., New York, NY, USA.