信息网络综合专题研究课大作业

选题：应用层-机器学习部分

姓名：张艺

学号：17221267

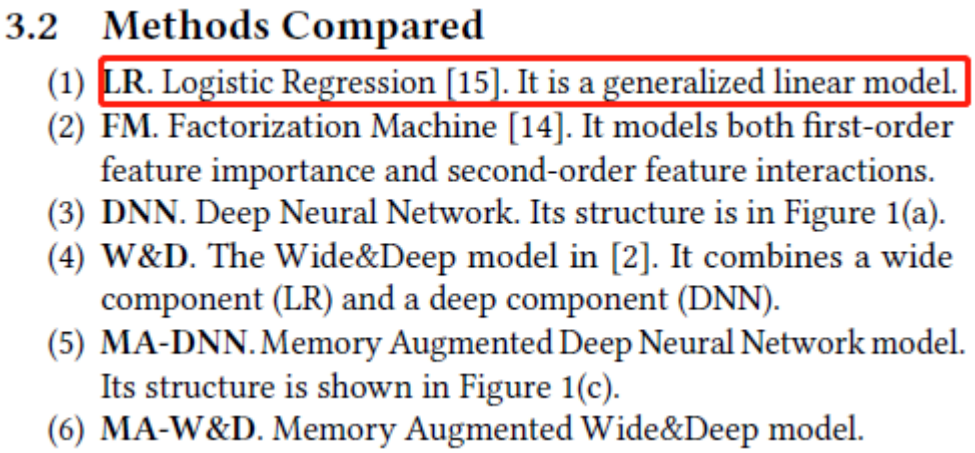
班级：通信 1708

日期：2020 年 5 月 21 日

指导老师：陈一帅

一、设计思路

之前读了网络层的一篇文献《Click-Through Rate Prediction with the User Memory Network》，在文中作者采用 MA-DNN 模型对网络广告的点击率进行了预测，过程中提及并且用到了 Logistic 回归算法，当时我并不太了解这个算法，只知道到它是一个广义的线性模型，对其具体的模型以及在文中的推导看不太懂，因此在阅读笔记中留下了存疑的标记。如图 1。



3.2 Methods Compared

(1) **LR**. Logistic Regression [15]. It is a generalized linear model.
(2) **FM**. Factorization Machine [14]. It models both first-order feature importance and second-order feature interactions.
(3) **DNN**. Deep Neural Network. Its structure is in Figure 1(a).
(4) **W&D**. The Wide&Deep model in [2]. It combines a wide component (LR) and a deep component (DNN).
(5) **MA-DNN**. Memory Augmented Deep Neural Network model. Its structure is shown in Figure 1(c).
(6) **MA-W&D**. Memory Augmented Wide&Deep model.

Figure 1 **论文中用到了 LR 模型**

因此在此次应用层机器学习的大作业中，我在 GitHub 上下载了有关 Logistic 回归算法的程序进行学习。

代码地址：

https://github.com/apachecn/AiLearning/blob/master/src/py2.x/ml/5.Logistic/logistic.py

此代码的提供者是一个叫 ApacheCN 的 AI learning 爱好者，专门制作机器学习有关的学习资料以及代码，对于我们一些机器学习

入门学习者来说，是一个宝库！

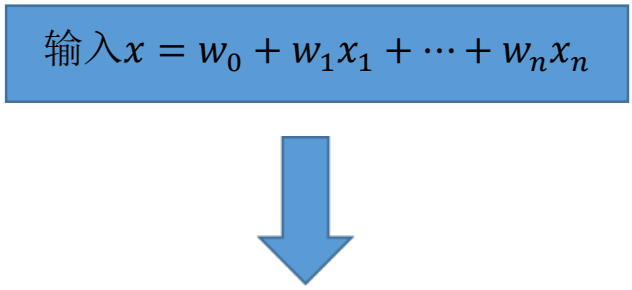
Figure 2 **代码提供者**

接下来我将对算法的具体内容以及代码的运行过程及结果进行介绍和演示。

二、内容及实现方式

首先我对 Logistic 回归的算法进行了初步的了解，其主要思想是：其主要思想是: 根据现有数据对分类边界线(Decision Boundary)建立回归公式，以此进行分类。

归纳其算法流程如下：



用梯度上升等算法求解 logistic 回归模型参数 w(训练模型)

代入 Sigmoid 函数 $S(x)=\frac{1}{1+e^{-z}}$，

进行回归计算、分类（使用算法）

下面，对代码进行讲解：

此代码的语言是 python，内容不太复杂，只引用了一个 numpy 第三方库。

```
import numpy as np
```

（1）

首先，定义了一个数据集函数，用来检索原始数据的特征及标签。

```
def load_data_set():
    """
    加载数据集
    :return:返回两个数组，普通数组
        data_arr -- 原始数据的特征
        label_arr -- 原始数据的标签，也就是每条样本对应的类别
    """
    data_arr = []
    label_arr = []
    f = open('TestSet.txt', 'r')
    for line in f.readlines():
        line_arr = line.strip().split()
        # 为了方便计算，我们将 X0 的值设为 1.0 ，也就是在每一行的开头添加一个 1.0 作为 X0
        data_arr.append([1.0, np.float(line_arr[0]), np.float(line_arr[1])])
        label_arr.append(int(line_arr[2]))
    return data_arr, label_arr
```

此函数将原始数据的特征集放在定义的数组 data_arr 中，标签放置在 label_arr 中，前提原始数据需要以图 3 的格式结构化存储在 txt 文本中。

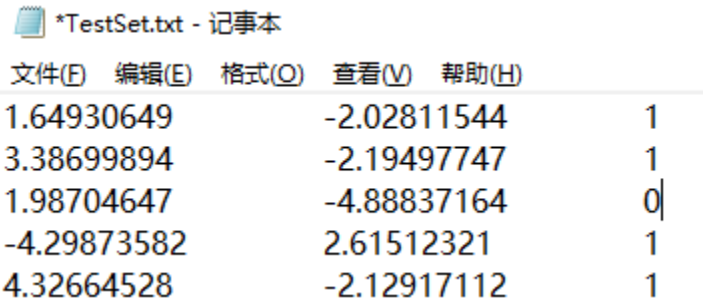


Figure 3 **原始数据需要结构化存储**

（2）之后定义了一个 Sigmoid 函数$S(x) = \frac{1}{1+e^{-x}}$，用其进行回归计算。

```
def sigmoid(x):
    # 这里其实非常有必要解释一下，会出现的错误 RuntimeWarning: overflow
encountered in exp
    # 这个错误在学习阶段虽然可以忽略，但是我们至少应该知道为什么
    # 这里是因为我们输入的有的 x 实在是太小了，比如 -6000 之类的，那么计算一个数字
np.exp(6000)这个结果太大了，没法表示，所以就溢出了
    # 如果是计算 np.exp（-6000），这样虽然也会溢出，但是这是下溢，就是表示成零
    # 去网上搜了很多方法，比如 使用 bigfloat 这个库（我竟然没有安装成功，就不尝试了，
反正应该是有用的
    return 1.0 / (1 + np.exp(-x))
```

（3）第三个函数是数据可视化的函数，用来画图。

```
def plot_best_fit(weights):
    """
    可视化
    :param weights:
    :return:
    """
    import matplotlib.pyplot as plt
    data_mat, label_mat = load_data_set()
    data_arr = np.array(data_mat)
    n = np.shape(data_mat)[0]
    x_cord1 = []
    y_cord1 = []
    x_cord2 = []
    y_cord2 = []
    for i in range(n):
```

```
        if int(label_mat[i]) == 1:
            x_cord1.append(data_arr[i, 1])
            y_cord1.append(data_arr[i, 2])
        else:
            x_cord2.append(data_arr[i, 1])
            y_cord2.append(data_arr[i, 2])
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.scatter(x_cord1, y_cord1, s=30, color='k', marker='^')
    ax.scatter(x_cord2, y_cord2, s=30, color='red', marker='s')
    x = np.arange(-3.0, 3.0, 0.1)
    y = (-weights[0] - weights[1] * x) / weights[2]
    """
    y 的由来，卧槽，是不是没看懂？
    首先理论上是这个样子的。
    dataMat.append([1.0, float(lineArr[0]), float(lineArr[1])])
    w0*x0+w1*x1+w2*x2=f(x)
    x0 最开始就设置为 1 叻， x2 就是我们画图的 y 值，而 f(x)被我们磨合误差给算到
w0,w1,w2 身上去了
    所以： w0+w1*x+w2*y=0 => y = (-w0-w1*x)/w2
    """
    ax.plot(x, y)
    plt.xlabel('x1')
    plt.ylabel('y1')
    plt.show()
```

（4）其次就是 Logistic 回归算法最核心的部分，及用梯度上升算法来计算模型参数 w。原程序定义了 3 种梯度上升算法。

第一种：经典梯度上升算法：

```
def grad_ascent(data_arr, class_labels):
    """
    梯度上升法，其实就是因为使用了极大似然估计，这个大家有必要去看推导，只看代码感觉不
太够
    :param data_arr: 传入的就是一个普通的数组，当然你传入一个二维的ndarray 也行
    :param class_labels: class_labels 是类别标签，它是一个 1*100 的行向量。
                        为了便于矩阵计算，需要将该行向量转换为列向量，做法是将原向量转置，
再将它赋值给 label_mat
    :return:
    """
    # 注意一下，我把原来 data_mat_in 改成 data_arr,因为传进来的是一个数组，用这个比
较不容易搞混
    # turn the data_arr to numpy matrix
    data_mat = np.mat(data_arr)
    # 变成矩阵之后进行转置
    label_mat = np.mat(class_labels).transpose()
    # m->数据量，样本数 n->特征数
    m, n = np.shape(data_mat)
    # 学习率，learning rate
    alpha = 0.001
```

```
    # 最大迭代次数，假装迭代这么多次就能收敛 2333
    max_cycles = 500
    # 生成一个长度和特征数相同的矩阵，此处 n 为 3 -> [[1],[1],[1]]
    # weights 代表回归系数， 此处的 ones((n,1)) 创建一个长度和特征数相同的矩阵，其中的数全部都是 1
    weights = np.ones((n, 1))
    for k in range(max_cycles):
        # 这里是点乘  m x 3 dot 3 x 1
        h = sigmoid(data_mat * weights)
        error = label_mat - h
        # 这里比较建议看一下推导，为什么这么做可以，这里已经是求导之后的
        weights = weights + alpha * data_mat.transpose() * error
    return weights
```

这个算法的目的是为了找到某个函数的最大值，在此问题中是为了找到 w 的最优解，基本的思想是是沿着该函数的梯度方向探寻。如图 4，梯度上升算法到达每个点后都会重新估计移动的方向。从 P0 开始，计算完该点的梯度，函数就根据梯度移动到下一点 P1。在 P1 点，梯度再次被重新计算，并沿着新的梯度方向移动到 P2 。如此循环迭代，直到满足停止条件。迭代过程中，梯度算子总是保证我们能选取到最佳的移动方向。

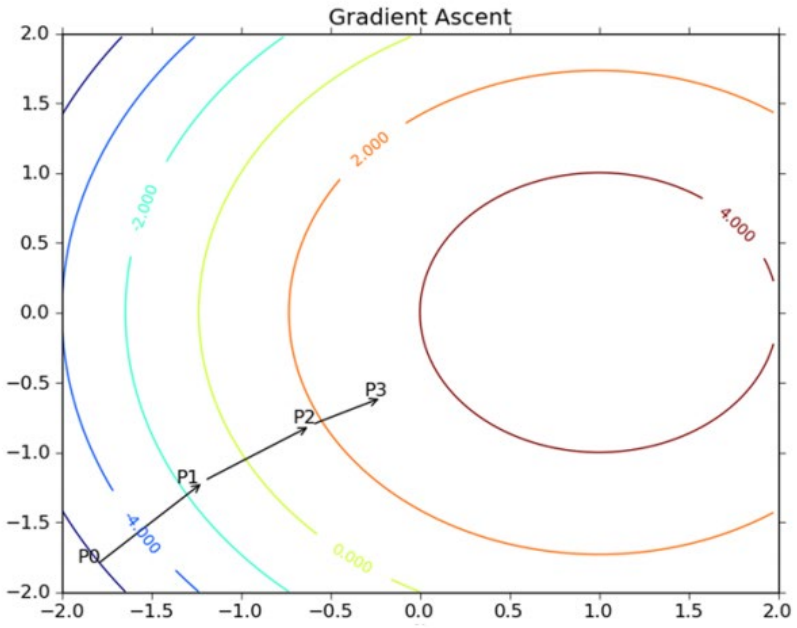


Figure 4 梯度上升法实例

具体的数学推导在此不过多论述。具体我参考了一些网页的资料，如 https://blog.csdn.net/xmu_jupiter/article/details/22220987

第二种就是梯度上升法的改进版，与经典的梯度上升法不同的是，这种算法每次只用一个样本点来更新回归系数。代码如下：

```
def stoc_grad_ascent0(data_mat, class_labels):
    """
    随机梯度上升，只使用一个样本点来更新回归系数
    :param data_mat: 输入数据的数据特征（除去最后一列）,ndarray
    :param class_labels: 输入数据的类别标签（最后一列数据）
    :return: 得到的最佳回归系数
    """
    m, n = np.shape(data_mat)
    alpha = 0.01
    weights = np.ones(n)
    for i in range(m):
        # sum(data_mat[i]*weights)为了求 f(x)的值， f(x)=a1*x1+b2*x2+..+nn*xn,
        # 此处求出的 h 是一个具体的数值，而不是一个矩阵
        h = sigmoid(sum(data_mat[i] * weights))
        error = class_labels[i] - h
        # 还是和上面一样，这个先去看推导，再写程序
        weights = weights + alpha * error * data_mat[i]
    return weights
```

第三种是改进版的随机梯度上升法，它也是每次只用一个样本点来更新回归系数，但不同的是每次使用的样本点是随机的。

```
def stoc_grad_ascent1(data_mat, class_labels, num_iter=150):
    """
    改进版的随机梯度上升，使用随机的一个样本来更新回归系数
    :param data_mat: 输入数据的数据特征（除去最后一列）,ndarray
    :param class_labels: 输入数据的类别标签（最后一列数据
    :param num_iter: 迭代次数
    :return: 得到的最佳回归系数
    """
    m, n = np.shape(data_mat)
    weights = np.ones(n)
    for j in range(num_iter):
        # 这里必须要用 list，不然后面的 del 没法使用
        data_index = list(range(m))
        for i in range(m):
            # i 和 j 的不断增大，导致 alpha 的值不断减少，但是不为 0
            alpha = 4 / (1.0 + j + i) + 0.01
            # 随机产生一个 0～len()之间的一个值
```

```
            # random.uniform(x, y) 方法将随机生成下一个实数，它在[x,y]范围内,x 是这
个范围内的最小值，y 是这个范围内的最大值。
            rand_index = int(np.random.uniform(0, len(data_index)))
            h = sigmoid(np.sum(data_mat[data_index[rand_index]] * weights))
            error = class_labels[data_index[rand_index]] - h
            weights = weights + alpha * error *
data_mat[data_index[rand_index]]
            del(data_index[rand_index])
    return weights
```

（5）最后就是串联起整个程序的测试函数。

```
def test():
    """
    这个函数只要就是对上面的几个算法的测试，这样就不用每次都在power shell 里面操作，
不然麻烦死了
    :return:
    """
    data_arr, class_labels = load_data_set()
    # 注意，这里的 grad_ascent 返回的是一个 matrix，所以要使用 getA 方法变成 ndarray
类型
    # weights = grad_ascent(data_arr, class_labels).getA()
    weights = stoc_grad_ascent0(np.array(data_arr), class_labels)
    # weights = stoc_grad_ascent1(np.array(data_arr), class_labels)
    plot_best_fit(weights)
```

接下来我们对代码进行运行，并对比三种梯度上升法的运行结果。

三、结果验证。

对于此代码的训练及测试数据，我采用了一个随机数生成的程序，批量生成了 50 个随机训练数据的二维特征值以及随机标签。

（1）首先我们来看采用经典的梯度上升法后 Logistic 回归的运行结果，如图 5。

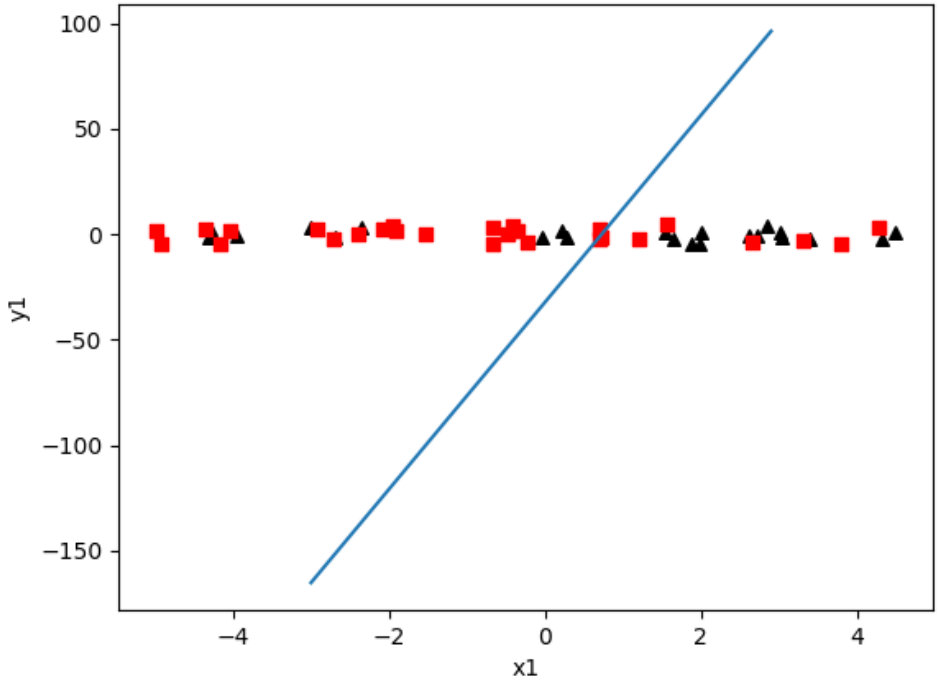


Figure 5 采用经典梯度上升法后的回归结果

可看出对随机的测试数据进行了的 Logistic 分类，而且分类的结果有点像简单的“一分为二”，并画出了其回归分类曲线。

而且多次运行之后，发现每次运行的结果都相同，这是因为每次求解的 w 值时选取的样本点都相同，计算出的 w 值也相同。

（2）然后是改进版的梯度上升算法。结果如图 6.

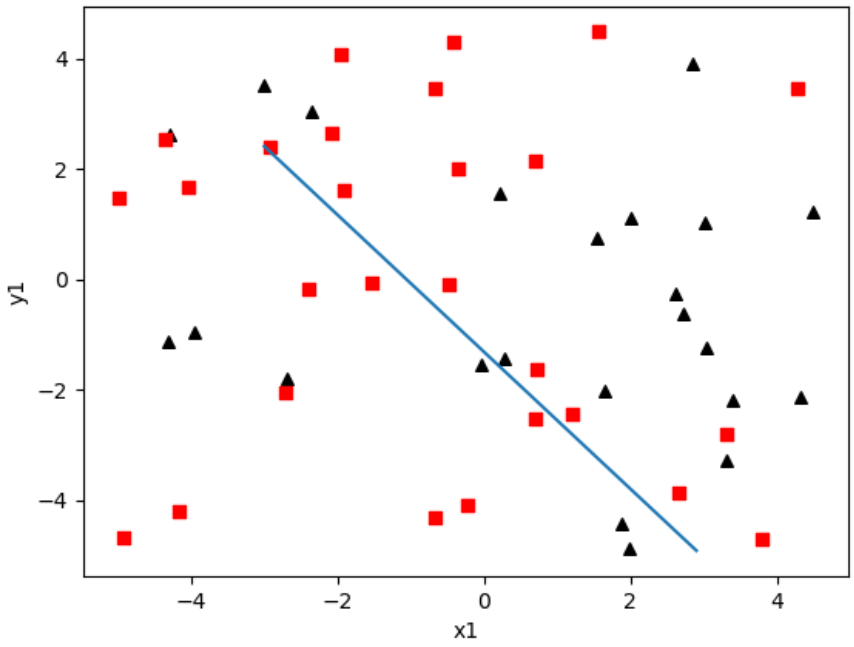
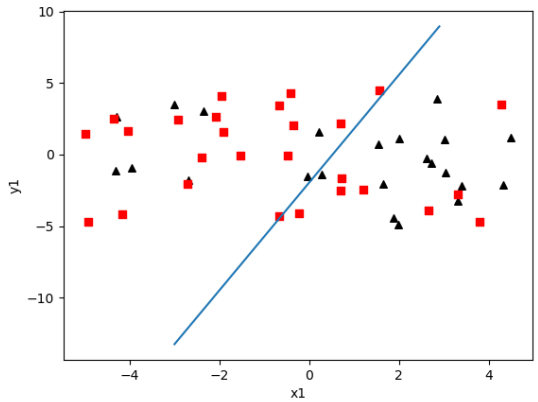
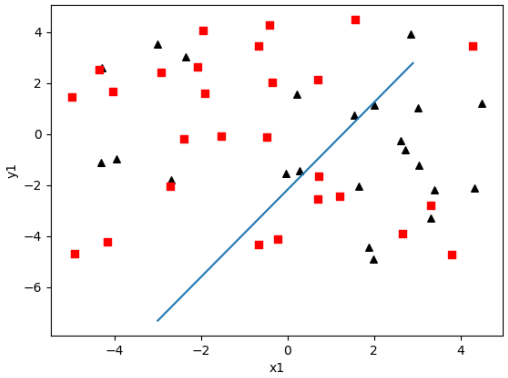


Figure 6 采用改进的梯度上升法后的回归结果

可见，分类结果同（1），且每次结果都相同。

（3）有趣的是采用改进的随机梯度上升算法时，我发现每次运行的结果都不尽相同。如图 7





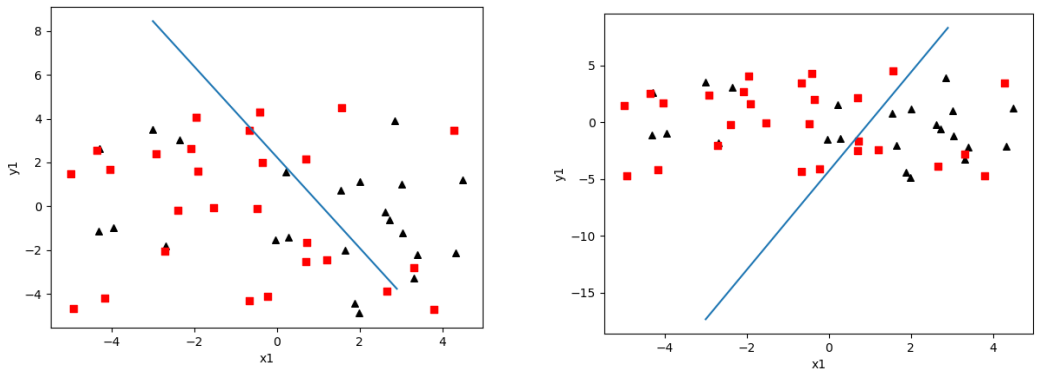


Figure 7 采用改进版的随机梯度上升法时每次运行的结果不尽相同

这是由于每次选择的样本点是随机的，导致每次计算出的回归参数 w 不同。

注：原 Github 代码后面还有一个关于 Logistic 回归应用案例“从疝气病症预测病马的死亡率”的程序，但很遗憾由于作者没有提供此程序的训练及测试数据集，，因此代码无法运行演示。

四、回顾总结

此次的大作业让我对机器学习中 Logistic 回归的知识有了近一步的了解，在对此次 Logistic 回归以及相关代码进行学习之后，我又回顾了之前我没看懂的那篇文献，发现对其中 Logistic 回归的部分有了一个大致的理解。（“大致”是因为还是有部分复杂的数学推导没有理解，但之后我还是会不断学习新的知识，对其进行逐步的学习和理解，直到完全看懂它，才能明白它的意义和价值）