实验 1　线性回归与逻辑回归实验

胡祝华 2021.06.10

1 课时和实验性质：

3 学时，验证性实验

2 实验目的：

1. 掌握线性回归（linear regression）的基本原理和实现方法，掌握基本术语和概念：序关系（order）、均方差（square error）最小化、欧式距离（Euclidean distance）、最小二乘法（least square method）、参数估计（parameter estimation）、多元线性回归（multivariate linear regression）、广义线性回归（generalized linear model）、对数线性回归（log-linear regression）；
2. 掌握逻辑回归（logistic regression）的基本原理和实现方法，掌握基本术语和概念：分类、Sigmoid 函数、对数几率（log odds / logit）、极大似然法（maximum likelihood method）；
3. 熟悉 LDA 线性判别分析和多分类转二分类的方法。

3 实验要求

1．按照实验步骤独立完成实验。
2．整理并上交实验报告和实验源程序。
　　..\网络应用开发实验报告模板.doc
3．考核：以学生的实验报告情况和做实验时的表现为考核依据。

4 实验环境

软件环境

- Windows 7 以上版本的操作系统或 Ubuntu 系统；
- 安装机器学习开发环境：1. Python 3.6 以上版本；2. Anaconda3 集成环境；3. Pycharm IDE。

硬件环境

- PC 机

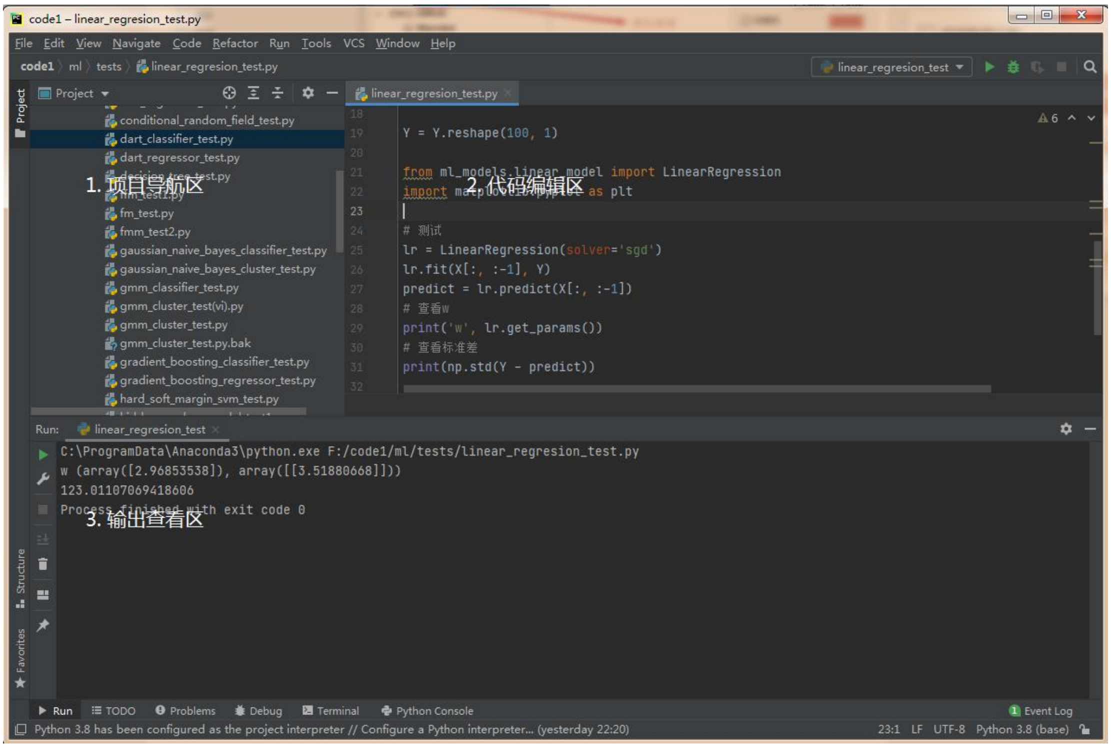
5 实验内容和步骤

1. Pycharm 仿真终端的使用介绍

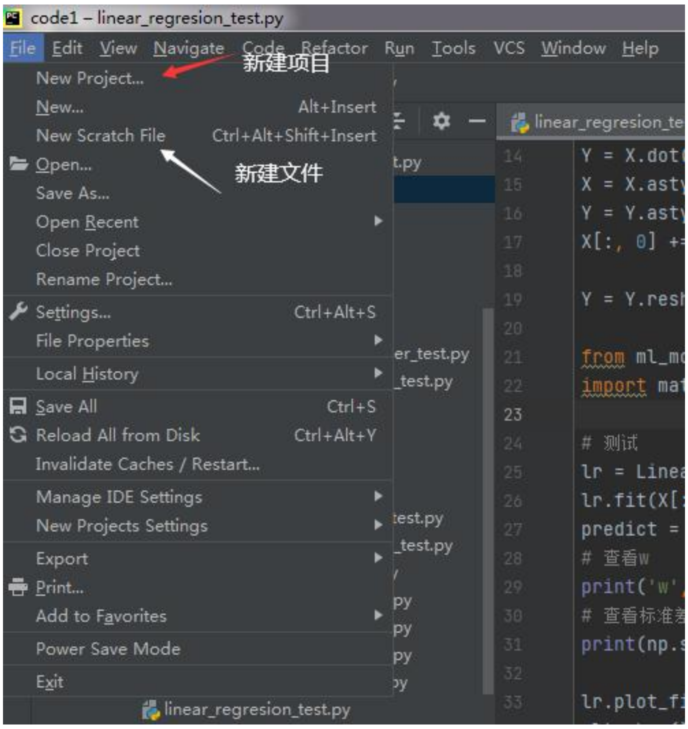
(1)认识 IDE 各部分组成

几个重要的菜单：File -> settings；View ->Tool windows；Run

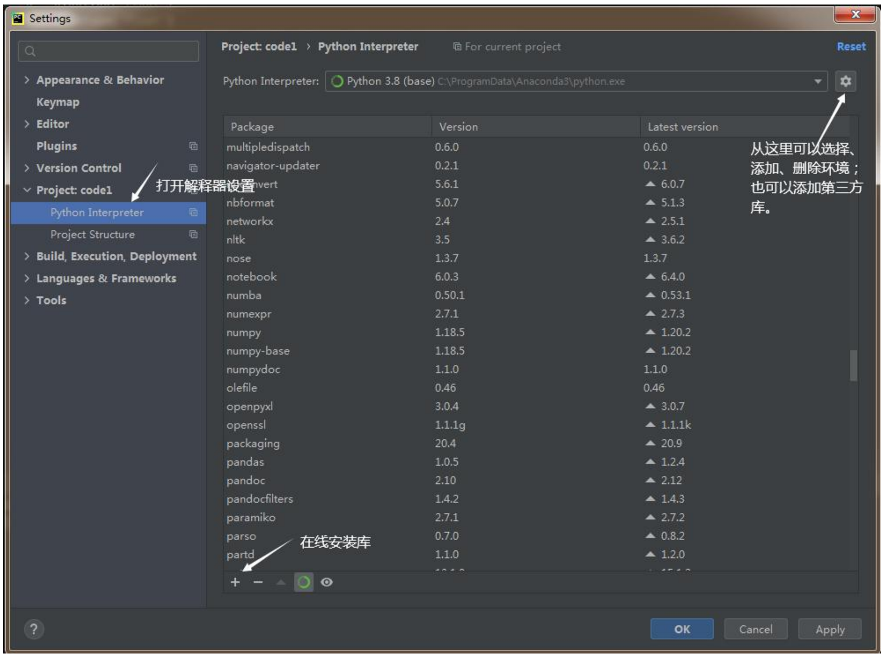
工作区的所有视图都可以通过 View->Tool windows 下的子菜单调出来。



(2)新建项目，新建文件



(3)设置好解释器环境，比如我们直接利用已经安装好的 Anaconda 3 的环境。

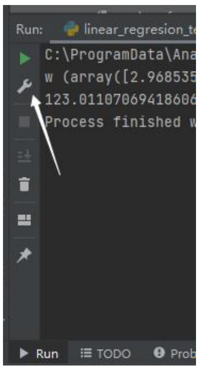


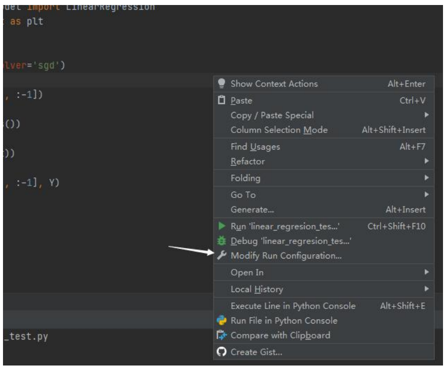
也可以在线或离线安装需要的第三方库。

https://blog.csdn.net/wmrem/article/details/80004819

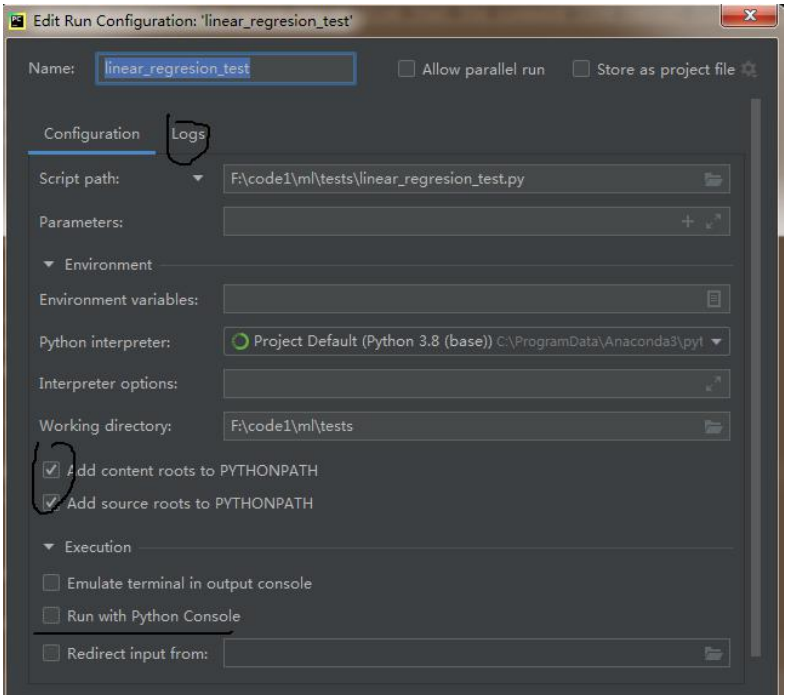
(4)运行设置,执行代码

运行设置：一般情况下通过两种方式打开“运行设置”窗口

1）

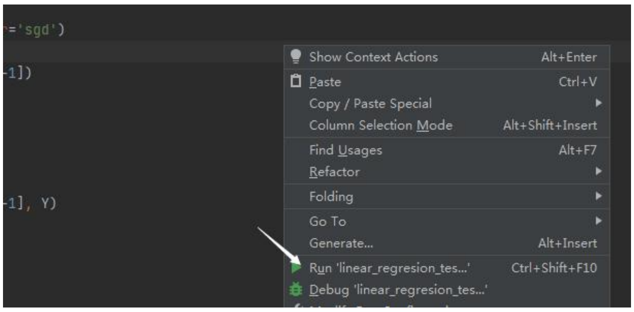
2）

运行设置窗口如下，可以在窗口中进行配置：



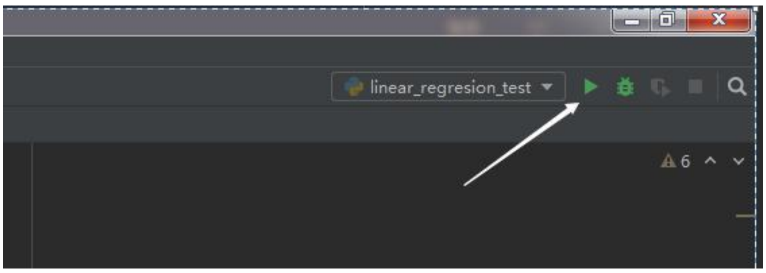
有 5 种运行方式：

1）在打开的代码上点击右键运行

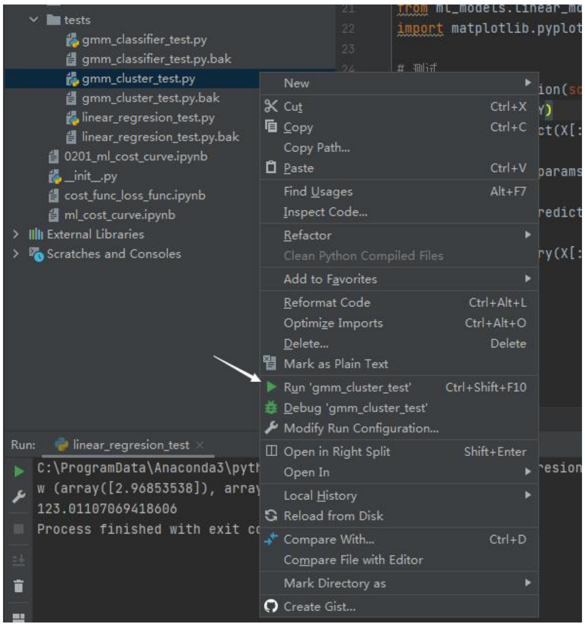


2）在菜单中运行

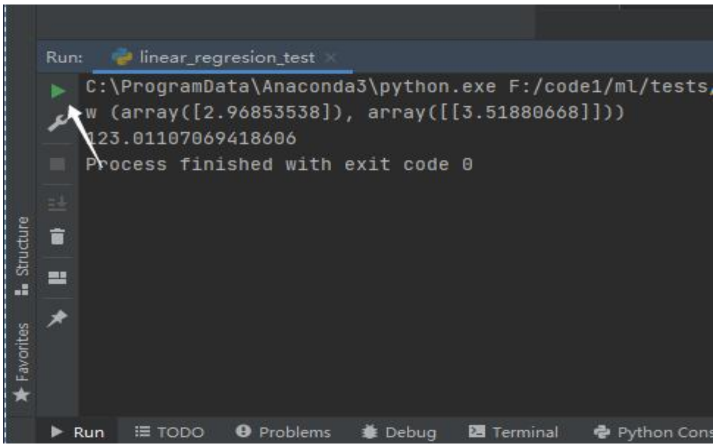
3）在工具栏中运行



4）在文件上点击右键运行



5）在 run 视图中点击运行按钮运行



(5) 调节背景

File -> settings -> editor -> color scheme

(6) 调试技巧

https://blog.csdn.net/u013088062/article/details/50214459

https://blog.csdn.net/qq_33472146/article/details/90606359

https://zhuanlan.zhihu.com/p/62610785

2. 线性回归实验

<u>原理</u>：线性回归算是回归任务中比较简单的一种模型，它的模型结构可以表示如下：

$f(x)=w^T x^*$， $x^*=\left[x^T,1\right]^T$， $x\in R^n$， 所以 $w\in R^{n+1}$， w 即是模型需要学习的参数。

步骤 1：导入 numpy 库，并将当前项目根目录加入解释器的搜索路径中。

```
import numpy as np #导入矩阵运算库，并取 np 别名

###将根目录添加到 sys.path，解决在命令行下执行时找不到模块的问题。
```

```
import sys
import os
curPath = os.path.abspath(os.path.dirname(__file__))
rootPath = os.path.split(curPath)[0]
sys.path.append(rootPath) # sys.path 是解释器的搜索路径
```

步骤 2：伪造 100 个样本数据集

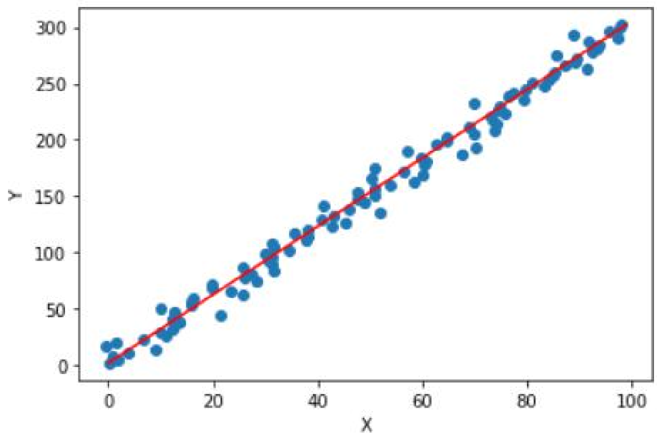
```
X = np.linspace(0, 100, 100)  #在 0-100 之间生成 100 个数。
X = np.c_[X, np.ones(100)]  #考虑到偏置 b, np.c_[a,b,c...]可以拼接多个数组，要求待拼接的多个数组的行数必须相同，  100*2
w = np.asarray([3, 2])  #参数 w=3, b=2  y=3x+b   2*1
Y = X.dot(w)  #100*2 的矩阵乘以 2*1 的矩阵 = 100*1
X = X.astype('float')  #强制类型转换
Y = Y.astype('float')
X[:, 0] += np.random.normal(size=(X[:, 0].shape)) * 3  # 添加 0,1 高斯噪声，即添加扰动，就是让这些点不要完全在一条直线上。

Y = Y.reshape(100, 1)  #以 100 行 1 列的数组形式显示  reshape（a,b）, a=-1 表示自动计算，reshape()是改变维数
```

步骤 3：把生成的样本数据集用散点图画出来，同时画出真实的直线

```
    import matplotlib.pyplot as plt  # 导入画图的函数
    plt.scatter(X[:,0],Y)  # 画出散点图
    plt.plot(np.arange(0,100).reshape((100,1)),Y,'r') # 画直线，因为 Y 的值没有加扰
    plt.xlabel('X')  #标记 X
    plt.ylabel('Y')  #标记 Y
    plt.show()
```

效果如下：



步骤 4：　参数的求解方法

（1）原理：

损失函数：利用等式 $y = 3x + 2$ 我造了一些伪数据，并给 x 添加了一些噪声数据，线性

回归的目标即在只有 x , y 的情况下，求解出最优解： $w=[3,2]^T$ ；可以通过 MSE（均方误差）来衡量 $f(x)$ 与 y 的相近程度：

$$L(w)=\sum_{i=1}^{m}(y_i-f(x_i))^2=\sum_{i=1}^{m}(y_i-w^Tx_i^*)^2=(Y-X^*w)^T(Y-X^*w) \quad (1)$$

这里 m 表示样本量，本例中 $m=100$ ， x_i,y_i 表示第 i 个样本， $X^*\in R^{m\times(n+1)},Y\in R^{m\times1}$ ，损失函数 $L(w)$ 本质上是关于 w 的函数，通过求解最小的 $L(w)$ 即可得到 w 的最优解：

$w^*=arg\min_w L(w)$

方法一：直接求闭式解 (一维的情况下参考 ppt)

而对 $\min L(w)$ 的求解很明显是一个凸问题（海瑟矩阵 $X^{*T}X^*$ 正定），我们可以直接通过求解 $\frac{dL}{dw}=0$ 得到 w^* ，梯度推导如下：

$$\frac{dL}{dw}=-2\sum_{i=1}^{m}(y_i-w^Tx_i^*)x_i^*=-2X^{*T}(Y-X^*w) \quad (2)$$

令 $\frac{dL}{dw}=0$ ，可得： $w^*=(X^{*T}X^*)^{-1}X^{*T}Y$ ，实际情景中数据不一定能满足 $X^{*T}X$ 是满秩（比如 $m<n$ 的情况下， w 的解有无数种），所以没法直接求逆，我们可以考虑用如下的方式求解：

$$X^{*+}=\lim_{\alpha\to0}(X^{*T}X^*+\alpha I)^{-1}X^{*T} \quad (3)$$

上面的公式即是 Moore-Penrose 伪逆的定义，但实际求解更多是通过 SVD 的方式：

$$X^{*+}=VD^+U^T \quad (4)$$

其中， U , D , V 是矩阵 X^* 做奇异值分解（SVD）后得到的矩阵，对角矩阵 D 的伪逆 D^+ 由其非零元素取倒数之后再转置得到，通过伪逆求解到的结果有如下优点：

（1）当 w 有解时， $w^*=X^{*+}Y$ 是所有解中欧几里得距离 $\|w\|_2$ 最小的一个；

（2）当 w 无解时，通过伪逆得到的 w^* 是使得 X^*w^* 与 Y 的欧几里得距离 $\|X^*w^*-Y\|_2$ 最小

方法二：梯度下降求解

但对于数据量很大的情况，求闭式解的方式会让内存很吃力，我们可以通过随机梯度下降法（SGD）对 w 进行更新，首先随机初始化 w ，然后使用如下的迭代公式对 w 进行迭代更新：

$$w:=w-\eta\frac{dL}{dw} \quad (5)$$

（2）模型训练（利用 SGD 求解）

前面推导出了 w 的更新公式，接下来编码训练过程，迭代结束后参数就计算出来了：

```
#参数随机初始化
w=np.random.random(size=(2,1))
#更新参数
epoches=100  #迭代次数
eta=0.0000001  #步长，可以适当调整步长
losses=[]  #记录 loss 变化
for _ in range(epoches):    # SGD 迭代
    dw=-2*X.T.dot(Y-X.dot(w))  #求 dL/dw
```

```
    w=w-eta*dw      #更新 w
losses.append((Y-X.dot(w)).T.dot(Y-X.dot(w)).reshape(-1))   #记录 loss 值
```

步骤 5：根据训练得到的参数 w 进行可视化展示

```
#可视化
plt.scatter(X[:,0],Y)       #散点图
plt.plot(X[:,0],X.dot(w),'r')  #可视化直线 y = ax+b
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

步骤 6：查看 loss 的变化。通过图直观的查看 loss 值是否随着迭代次数的增加趋于平稳？注意：loss 是否趋于平稳是衡量是一个模型是否已经收敛的重要指标。

```
#loss 变化
plt.plot(losses)
plt.xlabel('iterations')
plt.ylabel('loss')
plt.show()
```

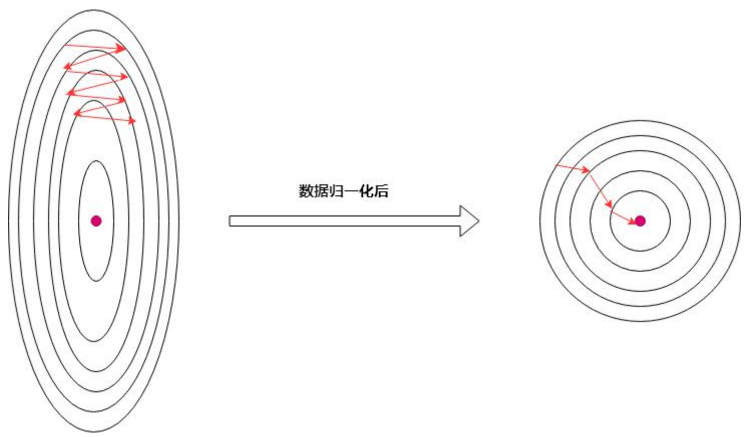
实验补充：另一种尝试，即利用求伪逆直接求闭式解。编程如下：

```
w=np.linalg.pinv(X).dot(Y)   # np.linalg 线性代数的函数模块
#可视化
plt.scatter(X[:,0],Y)
plt.plot(X[:,0],X.dot(w),'r')
plt.plot(np.arange(0,100).reshape((100,1)),Y, c='b', linestyle='--') #真实直线
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```

线性回归实验的问题讨论：

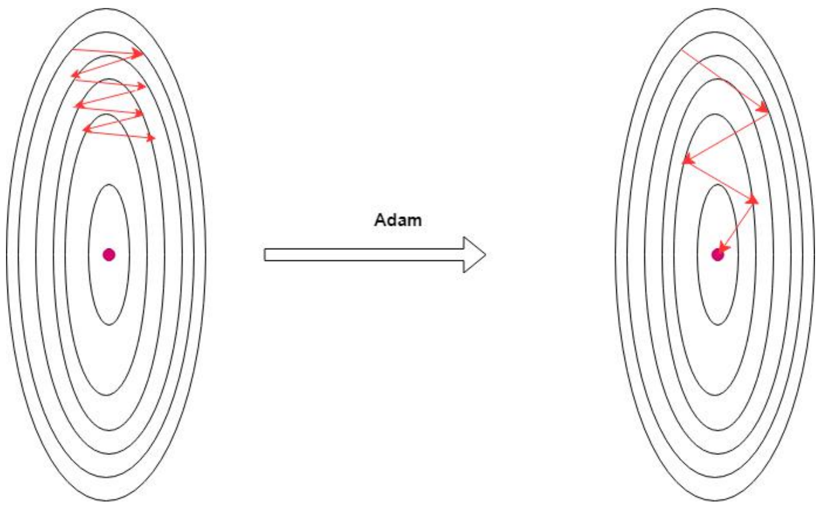
在上面的梯度下降的实验中存在一个问题，w_1 基本能收敛到 3 附近，而 w_2 却基本在 0 附近，很难收敛到 2，说明 w_1 比 w_2 更容易收敛（$w=[w_1,w_2]^T$），这很容易理解，模型可以写作：$f(x)=x*w_1+1\cdot w_2$，如果 x 量纲比 1 大很多，为了使 $f(x)$ 变化，只需更新少量的 w_1 就能达到目的，而 w_2 的更新动力略显不足；可以考虑用如下方式：

（1）对输入 X 进行归一化，使得 x 无量纲，w_1,w_2 的更新动力一样（后面封装代码时添加上），如下图；



归一化对梯度下降的影响

（2）梯度更新时，w_1, w_2 使用了一样的学习率，可以让 w_1, w_2 使用不一样的学习率进行更新，比如对 w_2 使用更大的学习率进行更新（可以利用学习率自适应一类的梯度下降法，比如 adam），如下图：



线性回归实验的提高：

（1）线性回归模块的封装，以便模块可被重复使用

简单封装线性回归模型，并放到 my_models.linear_model 模块便于后续使用，并在该两个目录下添加__init__.py 文件。

<u>该文件的作用：</u>标识该目录是一个 python 的模块包（module package），如果你是使用 python 的相关 IDE 来进行开发，那么如果目录中存在该文件，该目录就会被识别为 module package。实际上，如果目录中包含了 __init__.py 时，当用 import 导入该目录时，会执行 __init__.py 里面的代码。因此，我们可以在__init__.py 指定默认需要导入的模块，有时候我们在做导入时会偷懒，将包中的所有内容导入，比如 from mypackage import *。因此，该文件就是一个正常的 python 代码文件，可以将初始化代码放入该文件中。

注意：如果使用了中文注释，在文件的最上方加上如下的代码块：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

将跟线性回归相关的代码封装到一个类里面，linear_regression.py，并放到 my_models\linear_model 目录下。代码如下：

```
#!/usr/bin/python
```

```python
# -*- coding: UTF-8 -*-

import numpy as np
import matplotlib.pyplot as plt

class LinearRegression(object):
    def __init__(self, fit_intercept=True, solver='sgd',
if_standard=True, epochs=10, eta=1e-2, batch_size=1):
        """
        :param fit_intercept: 是否训练bias
        :param solver:
        :param if_standard:
        """
        self.w = None
        self.fit_intercept = fit_intercept
        self.solver = solver
        self.if_standard = if_standard
        if if_standard:
            self.feature_mean = None
            self.feature_std = None
        self.epochs = epochs
        self.eta = eta
        self.batch_size = batch_size

    def init_params(self, n_features):
        """
        初始化参数
        :return:
        """
        self.w = np.random.random(size=(n_features, 1))

    def _fit_closed_form_solution(self, x, y):
        """
        直接求闭式解
        :param x:
        :param y:
        :return:
        """
        self.w = np.linalg.pinv(x).dot(y)

    def _fit_sgd(self, x, y):
        """
        随机梯度下降求解
        :param x:
```

```
        :param y:
        :param epochs:
        :param eta:
        :param batch_size:
        :return:
        """
        x_y = np.c_[x, y]
        # 按batch_size 更新w,b
        for _ in range(self.epochs):
            np.random.shuffle(x_y)  # 将序列的所有元素随机排序。
            for index in range(x_y.shape[0] // self.batch_size): # 向下取
整
                batch_x_y = x_y[self.batch_size * index:self.batch_size *
(index + 1)]
                batch_x = batch_x_y[:, :-1]
                batch_y = batch_x_y[:, -1:]

                dw = -2 * batch_x.T.dot(batch_y - batch_x.dot(self.w)) /
self.batch_size
                self.w = self.w - self.eta * dw

    def fit(self, x, y):
        # 是否归一化 feature
        if self.if_standard:
            self.feature_mean = np.mean(x, axis=0)
            self.feature_std = np.std(x, axis=0) + 1e-8
            x = (x - self.feature_mean) / self.feature_std
        # 是否训练bias,np.ones_like 返回一个用1 填充的跟输入形状和类型 一致的数
组。
        if self.fit_intercept:
            x = np.c_[x, np.ones_like(y)]
        # 初始化参数
        self.init_params(x.shape[1])
        # 训练模型
        if self.solver == 'closed_form':
            self._fit_closed_form_solution(x, y)
        elif self.solver == 'sgd':
            self._fit_sgd(x, y)

    def get_params(self):
        """
        输出原始的系数
        :return: w,b
        """
```

```python
        if self.fit_intercept:
            w = self.w[:-1] # [:-1]表示除了最后一个的其他部分，[::-1]表示列表逆序，[2::-1]表示取从下标为 2 的元素翻转读取
            b = self.w[-1] # [-1]取最后一个元素
        else:
            w = self.w
            b = 0
        if self.if_standard:
            w = w / self.feature_std.reshape(-1, 1)
            b = b - w.T.dot(self.feature_mean.reshape(-1, 1))
        return w.reshape(-1), b

    def predict(self, x):
        """
        :param x:ndarray 格式数据: m x n
        :return: m x 1
        """
        if self.if_standard:
            x = (x - self.feature_mean) / self.feature_std
        if self.fit_intercept:
            x = np.c_[x, np.ones(shape=x.shape[0])]
        return x.dot(self.w)

    def plot_fit_boundary(self, x, y):
        """
        绘制拟合结果
        :param x:
        :param y:
        :return:
        """
        plt.scatter(x[:, 0], y)
        plt.plot(x[:, 0], self.predict(x), 'r')
```

（2）基于封装模块的测试，命名为 linear_regression_test.py，并放到 tests 目录下。有三种测试方式。

1）直接调用封装对象中的 SGD 方法进行训练和预测，参考代码如下：

```python
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import numpy as np
import matplotlib.pyplot as plt

###将根目录添加到 sys.path，解决在命令行下执行时找不到模块的问题。
```

```python
import sys
import os
curPath = os.path.abspath(os.path.dirname(__file__))
rootPath = os.path.split(curPath)[0]
sys.path.append(rootPath)

# 造伪样本
X = np.linspace(0, 100, 100)
X = np.c_[X, np.ones(100)]  #考虑到偏置 b
w = np.asarray([3, 2])  #参数
Y = X.dot(w)
X = X.astype('float')
Y = Y.astype('float')
X[:, 0] += np.random.normal(size=(X[:, 0].shape)) * 3  # 添加 0,1 高斯噪
声

Y = Y.reshape(100, 1)

from my_models.linear_model import *

#测试
lr=LinearRegression(solver='sgd')
lr.fit(X[:,:-1],Y)
predict=lr.predict(X[:,:-1])
#查看 w
print('w',lr.get_params())
#查看标准差，如果标准差小的话则认为真实值与预测值相符合。
print(np.std(Y-predict))

#可视化结果
lr.plot_fit_boundary(X[:, :-1], Y)  # 预测的拟合直线
plt.plot(np.arange(0,100).reshape((100,1)),Y, c='b', linestyle='--')
#真实直线
plt.show()  # 可视化显示
```

2）使用闭式解求解参数的方法进行测试

```python
… …
lr=LinearRegression(solver='closed_form')
… …
```

3）用 sklearn 中的 linear_model 模块中的 LinearRegression 类测试。

```python
… …
#与 sklearn 对比
from sklearn.linear_model import LinearRegression
```

```
lr=LinearRegression()
lr.fit(X[:,:-1],Y)
predict=lr.predict(X[:,:-1])
#查看 w,b
print('w:',lr.coef_,'b:',lr.intercept_)
#查看标准差
print(np.std(Y-predict))

#可视化结果
plt.scatter(X[:, 0], Y)
plt.plot(X[:, 0], predict, 'r')
plt.plot(np.arange(0,100).reshape((100,1)),Y, c='b', linestyle='--')
plt.show()
```

3. 逻辑回归实验

步骤 1：导入实验所需的相关资源和模块

```
import numpy as np
import os
os.chdir('../')  # 用于改变当前工作目录到指定的路径。
from my_models import utils
import matplotlib.pyplot as plt
```

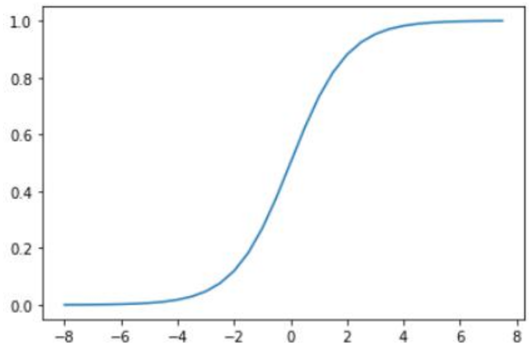
步骤 2：实验原理

逻辑回归(Logistic Regression)简单来看就是在线性回归模型外面再套了一个 Sigmoid 函数：

$$\delta(t)=\frac{1}{1+e^{-t}}$$

它的函数形状如下：

```
t=np.arange(-8,8,0.5)
d_t=1/(1+np.exp(-t))
plt.plot(t,d_t)
plt.show()
```



而将 t 替换为线性回归模型 $w^T x^*$（这里 $x^*=[x^T,1]^T$）即可得到逻辑回归模型：

$$f(x)=\delta(w^T x^*)=\frac{1}{1+e^{-(w^T x^*)}}$$

我们可以发现：Sigmoid 函数决定了模型的输出在 $(0,1)$ 区间，所以逻辑回归模型可以用作区间在 $(0,1)$ 的回归任务，也可以用作 $\{0,1\}$ 的二分类任务；同样，由于模型的输出在 $(0,1)$ 区间，所以逻辑回归模型的输出也可以看作这样的“概率”模型：

$$P(y=1|x)=f(x)$$
$$P(y=0|x)=1-f(x)$$

所以，逻辑回归的学习目标可以通过极大似然估计求解：$\prod_{j=1}^{n} f(x_j)^{y_j}(1-f(x_j))^{(1-y_j)}$，即使得观测到的当前所有样本的所属类别概率尽可能大；通过对该函数取负对数，即可得到交叉熵损失函数：

$$L(w)=-\sum_{j=1}^{n} y_j log(f(x_j))+(1-y_j)log(1-f(x_j))$$

这里 n 表示样本量，$x_j \in R^m$，m 表示特征量，$y_j \in \{0,1\}$，接下来的与之前推导一样，通过梯度下降求解 w 的更新公式即可：

$$\frac{\partial L}{\partial w}=-\sum_{i=1}^{n}(y_i-f(x_i))x_i^*$$

所以 w 的更新公式：

$$w:=w-\eta\frac{\partial L}{\partial w}$$

步骤 3：与线性回归类似，模型训练的代码及封装如下所示。代码命名为 logic_regression.py，并放到 my_models\linear_model 目录下。

```
class LogisticRegression(object):
    def __init__(self, fit_intercept=True, solver='sgd', if_standard=True, epochs=10, eta=None, batch_size=16):


        self.w = None
        self.fit_intercept = fit_intercept
        self.solver = solver
        self.if_standard = if_standard
        if if_standard:
            self.feature_mean = None
            self.feature_std = None
        self.epochs = epochs
        self.eta = eta
        self.batch_size = batch_size
        # 注册 sign 函数
        self.sign_func = np.vectorize(utils.sign)
```

```
        # 记录 losses
        self.losses = []

    def init_params(self, n_features):
        """
        初始化参数
        :return:
        """
        self.w = np.random.random(size=(n_features, 1))

    def _fit_sgd(self, x, y):
        """
        随机梯度下降求解
        :param x:
        :param y:
        :return:
        """
        x_y = np.c_[x, y]
        count = 0
        for _ in range(self.epochs):
            np.random.shuffle(x_y)
            for index in range(x_y.shape[0] // self.batch_size):
                count += 1
                batch_x_y = x_y[self.batch_size * index:self.batch_size * (index + 1)]
                batch_x = batch_x_y[:, :-1]
                batch_y = batch_x_y[:, -1:]

                dw = -1 * (batch_y - utils.sigmoid(batch_x.dot(self.w))).T.dot(batch_x) / self.batch_size
                dw = dw.T

                self.w = self.w - self.eta * dw

            # 计算 losses
            cost = -1 * np.sum(
            np.multiply(y, np.log(utils.sigmoid(x.dot(self.w)))) + np.multiply(1 - y, np.log(1 - utils.sigmoid(x.dot(self.w)))))
            self.losses.append(cost)

    def fit(self, x, y):
        """
        :param x: ndarray 格式数据: m x n
        :param y: ndarray 格式数据: m x 1
        :return:
```

```
        """
        y = y.reshape(x.shape[0], 1)
        # 是否归一化 feature
        if self.if_standard:
            self.feature_mean = np.mean(x, axis=0)
            self.feature_std = np.std(x, axis=0) + 1e-8
            x = (x - self.feature_mean) / self.feature_std
        # 是否训练 bias
        if self.fit_intercept:
            x = np.c_[x, np.ones_like(y)]
        # 初始化参数
        self.init_params(x.shape[1])
        # 更新 eta
        if self.eta is None:
            self.eta = self.batch_size / np.sqrt(x.shape[0])

        if self.solver == 'sgd':
            self._fit_sgd(x, y)

    def get_params(self):
        """
        输出原始的系数
        :return: w,b
        """
        if self.fit_intercept:
            w = self.w[:-1]
            b = self.w[-1]
        else:
            w = self.w
            b = 0
        if self.if_standard:
            w = w / self.feature_std.reshape(-1, 1)
            b = b - w.T.dot(self.feature_mean.reshape(-1, 1))
        return w.reshape(-1), b

    def predict_proba(self, x):
        """
        预测为 y=1 的概率
        :param x:ndarray 格式数据: m x n
        :return: m x 1
        """
        if self.if_standard:
            x = (x - self.feature_mean) / self.feature_std
        if self.fit_intercept:
```

```
            x = np.c_[x, np.ones(x.shape[0])]
        return utils.sigmoid(x.dot(self.w))

    def predict(self, x):
        """
        预测类别，默认大于 0.5 的为 1，小于 0.5 的为 0
        :param x:
        :return:
        """
        proba = self.predict_proba(x)
        return (proba > 0.5).astype(int)

    def plot_decision_boundary(self, x, y):
        """
        绘制前两个维度的决策边界
        :param x:
        :param y:
        :return:
        """
        y = y.reshape(-1)
        weights, bias = self.get_params()
        w1 = weights[0]
        w2 = weights[1]
        bias = bias[0][0]
        x1 = np.arange(np.min(x), np.max(x), 0.1)
        x2 = -w1 / w2 * x1 - bias / w2
        plt.scatter(x[:, 0], x[:, 1], c=y, s=50)
        plt.plot(x1, x2, 'r')
        plt.show()

    def plot_losses(self):
        plt.plot(range(0, len(self.losses)), self.losses)
        plt.show()
```

步骤 4：编写测试程序，命名为 logic_regression_test.py，并放到 tests 目录下。构建数据集，训练模型并进行测试。

```
#构造伪分类数据并可视化
from sklearn.datasets import make_classification
data,target=make_classification(n_samples=100,
n_features=2,n_classes=2,n_informative=1,n_redundant=0,n_repeated=0,n_clusters_per_class=1)
print(data.shape)
print(target.shape)
plt.scatter(data[:, 0], data[:, 1], c=target, s=50)
plt.show()
```

```
#训练模型
lr = LogisticRegression()
lr.fit(data, target)

#查看 loss 值的变化，交叉熵损失
lr.plot_losses()
```

绘制决策边界：令 $w_1x_1 + w_2x_2 + b = 0$，可得 $x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}$

```
#绘制决策边界
lr.plot_decision_boundary(data, target)
```

```
#计算 F1
from sklearn.metrics import f1_score
f1_score(target, lr.predict(data))
```

步骤 5：与 sklearn 中的逻辑回归模型的对比。

```
from sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
lr.fit(data, target)

w1=lr.coef_[0][0]
w2=lr.coef_[0][1]
bias=lr.intercept_[0]
print(w1)
print(w2)
print(bias)

#画决策边界
x1=np.arange(np.min(data),np.max(data),0.1)
x2=-w1/w2*x1-bias/w2
plt.scatter(data[:, 0], data[:, 1], c=target,s=50)
plt.plot(x1,x2,'r')
plt.show()

#计算 F1
f1_score(target,lr.predict(data))
```

6 思考与提高

1. 逻辑回归的损失函数为何不用 mse？

上面我们基本完成了二分类 LogisticRegression 代码的封装工作，并将其放到 liner_model 模块方便后续使用。我们讨论一下模型中损失函数选择的问题；

在前面线性回归模型中我们使用了 **mse** 作为损失函数，并取得了不错的效果，而逻辑回归中使用的确是**交叉熵损失函数**；这是因为如果使用 mse 作为损失函数，梯度下降将会比较困难，在 $f(x^i)$ 与 y^i 相差较大或者较小时梯度值都会很小，下面推导一下：

我们令：

$$L(w)=\frac{1}{2}\sum_{i=1}^{n}(y^i-f(x^i))^2$$

则有：

$$\frac{\partial L}{\partial w}=\sum_{i=1}^{n}(f(x^i)-y^i)f(x^i)(1-f(x^i))x^i$$

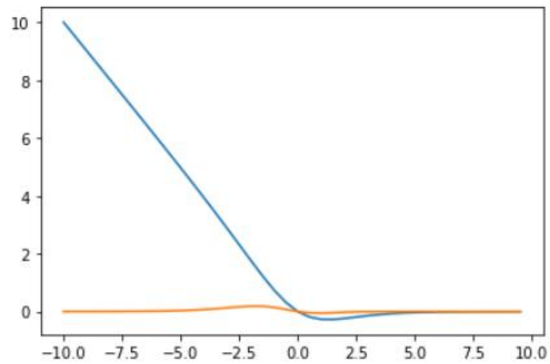
我们简单看两个极端的情况：

（1） $y^i=0,f(x^i)=1$ 时， $\frac{\partial L}{\partial w}=0$；

（2） $y^i=1,f(x^i)=0$ 时， $\frac{\partial L}{\partial w}=0$

接下来，我们绘图对比一下两者梯度变化的情况，假设在 $y=1,x\in(-10,10),w=1,b=0$ 的情况下：

```
y=1
x0=np.arange(-10,10,0.5)
#交叉熵
x1=np.multiply(utils.sigmoid(x0)-y,x0)
#mse
x2=np.multiply(utils.sigmoid(x0)-y,utils.sigmoid(x0))
x2=np.multiply(x2,1-utils.sigmoid(x0))
x2=np.multiply(x2,x0)
plt.plot(x0,x1)
plt.plot(x0,x2)
plt.show()
```



可见在错分的那一部分（x<0）,mse 的梯度值基本停留在 0 附近，而交叉熵会让越“错”情况具有越大的梯度值。