实验二：线性回归分类器

姓名：孙铭

学号：1711377

专业：计算机科学与技术

日期：2020年3月22日

摘要

本次实验要求实现基于数据集`winequality-white.csv`的线性回归分类器，助教给出的实验指导中关于本次实验的三类要求已全部实现。本人在此声明，本次实验全部代码及作业报告均由个人独立完成。具体实现的功能如下：

> *实验基本要求：*
>
> > *（1）基于数据集`winequality-white.csv`构造了线性回归分类器，同时实现了批量梯度下降（BGD）和随机梯度下降（SGD）两种算法。*
> >
> > *（2）参照周志华教授的西瓜书，将数据集划分成训练集和测试集。分别使用**留一法划分的数据集**和**个人划分的数据集**，对BGD和SGD两种梯度下降算法的均方根误差（RMSE）进行计算，同时得到收敛曲线。*
>
> *实验中级要求：*
>
> > *（1）对不同学习率下，损失函数值与循环次数的关系进行分析，并作出了对比图像，分析得到最佳学习率。*
> >
> > *（2）对BGD和SGD在不同循环次数下的分类准确率和运行时间做对比并画出图像，即得到两种梯度下降算法的收敛速度和最终结果的区别，同时进一步分析给出结论。*
>
> *实验高级要求：*
>
> > *对批量梯度下降算法引入拉格朗日常数，实现L2正则化，并将L2正则化实现的BGD和普通的BGD在不同迭代次数下分类准确率进行对比分析，并绘制图像，从而得到实验结果。*

以上是本次实验个人实现的功能，接下来将从环境配置、数据处理、模型实现、实验及结果分析几个模块展开进行论述。

环境配置

本次作业使用语言版本为`Python 3.6.5 64bit`，处理器为`Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80GHz`，项目工程文件如下。

```
| data
|    | winequality-white.csv #原始数据集
|    | winequality-white.pkl #序列化后的数据集，便于读写
|    | train.pkl             #人为划分的训练集
|    | test.pkl              #人为划分的测试集
| picture                    #实验结果图片保存文件夹
| cut_data.py                #数据集切分和转换成序列化文件
| regression.py              #线性回归模型实现
| experiment.py              #各种对比实验
```

项目实现过程中用到的包及作用如下。

```
import numpy as np                      #numpy数据结构
import pickle                           #处理序列化文件
import matplotlib.pyplot as plt         #绘制图像
import math                             #数学函数
import time                             #时间记录
import csv                              #csv文件处理
```

数据处理

本次实验所使用的数据集为白葡萄酒数据集`winequality-white.csv`，本数据集共有4898条观测值。数据集变量组成为11个输入变量和1个输出变量，各变量名称及含义如下。

```
fixed acidity               # 固定酸度
volatile acidity            # 挥发性酸度
citric acid                 # 柠檬酸
residual sugar              # 残留的糖分
chlorides                   # 氯化物
free sulfur dioxide         # 游离态二氧化硫
total sulfur dioxide        # 二氧化硫总量
density                     # 密度
pH                          # pH值
sulphates                   # 硫酸盐
alcohol                     # 酒精含量
quality                     # 品质（输出变量）
```

经过统计，白葡萄酒品质分类数量总共有7类，分别是：

$$3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0$$

由于`.csv`文件是逗号分隔符文件，默认以英文逗号作为分隔符，而数据集中的数据满足这一要求，因此使用`csv.reader()`函数直接读取数据即可，不需要额外对字符串进行分隔等处理。实现代码如下。

```
def load_data(filename):
    with open(filename, encoding='utf-8') as f:
        csv_reader = csv.reader(f)
        _data = [line for line in csv_reader]
    data = np.array(_data[1: ], dtype=float)
    return data
```

接下来是对训练集和测试集的划分，在周志华教授的西瓜书中，常见的做法是将2/3~4/5的样本数据用于训练，剩余样本用于测试。这里为了方便，我的划分方式为$\text{训练集}:\text{测试集}=7:3$。训练集和测试集以`numpy`矩阵形式保存在`.pkl`类型文件中。`.pkl`，即`pickle`，该类型文件具有好处是能够保留原有数据的序列化特征。

将数据写入`pickle`的代码如下。

```
# 将序列化对象存储在pickle文件中
def write_pkl(filename, data):
    f = open(filename, 'wb')
    pickle.dump(data, f)
    f.close()
```

划分训练集和测试集的函数`cut_data()`代码如下。

```
def cut_data(filename):
    data = load_data(filename);
    np.random.shuffle(data)
    train_count = int(len(data) / 10 * 7)
    write_pkl("data/train.pkl", data[: train_count])
    write_pkl("data/test.pkl", data[train_count: ])
```

如摘要中所提，在计算BGD和SGD的均方误差时，除了使用个人划分的数据集，我还使用了留一法处理序列化后的原始数据集`winequality-white.csv`，留一法实现代码如下。

```
data = read_pkl("data/winequality-white.pkl")
for iters in _iters:
    # 使用留一法
    for i in range(0, len(data)):
        test_X = X[i, :]
        test_Y = Y[i, :]
        train_X = np.delete(X, i, axis=0)
        train_Y = np.delete(Y, i, axis=0)
        # (and so on.....)
```

模型实现

接下来，需要对整个线性回归模型的实现进行分析。具体包括：线性回归方程及损失函数、批量梯度下降算法、随机梯度下降算法、L2-正则化批量梯度下降算法、线性回归分类器实现。

1. 线性回归方程及损失函数

线性回归常用于分析自变量和因变量之间的线性关系，由于本次实验所使用的线性回归模型为多元线性回归模型，故这里从多元线性回归相关概念及公式的角度进行分析。

首先，定义N个自变量$x_1, x_2, \ldots, x_N$，对于这N个自变量，我们定义一个常数项θ_0以及N个系数项$\theta_1, \theta_2, \ldots, \theta_N$，于是对于因变量$h_\theta(x)$，可以得到一个多元线性回归的假设函数：

$$h_\theta(x) = \sum_{i=1}^{N} \theta_i x_i + \theta_0$$

为了使得方程在形式上更加整齐，可以假设$x_0 = 1$，于是，原多元线性回归函数可以写为：

$$h_\theta(x) = \sum_{i=0}^{N} \theta_i x_i$$

在线性回归模型中，需要使用损失函数（代价函数）来度量真实值与预测值之间的关系，而最简单的损失函数便是经常使用的平方距离公式：

$$J_\theta(x) = \frac{1}{2m} \sum_{i=1}^{N} (h_\theta(x)^{(i)} - y^{(i)})^2$$

其中，$y^{(i)}$即为样本真实值。

在实际代码实现中，我使用`np.array()`存储数据，利用`numpy`的`broadcasting`机制，可以大幅增加计算速度的同时，缩减代码量。关于损失函数`cost_function()`实现的代码如下。

```
# 损失函数，X:样本特征矩阵，Y:样本的真实值，theta:参数矩阵
def cost_function(X, Y, theta):
    inner = np.power((np.dot(X, theta) - Y), 2)
    return np.sum(inner) / (2 * len(inner))
```

2. 批量梯度下降

在定义了线性回归函数和损失函数之后，我们的目标转化为，求使得损失函数值$J_\theta(x)$最小时的一组参数值$\theta_0, \theta_1, \ldots, \theta_N$。有三种算法可以实现，分别是批量梯度下降、随机梯度下降和小批量梯度下降。这里先说批量梯度下降。

批量梯度下降法是最原始的形式，它是指在每一次迭代时使用所有的样本来进行梯度更新。从数学角度理解如下。

首先，对目标函数求偏导：

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

其次，每次迭代对参数进行更新如下。

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

其中，j为样本的特征数。

批量梯度下降实现代码如下。

```
# 批量梯度下降，learning_rata:学习率 iters:迭代次数
def batch_gradient_descent(X, Y, theta, learning_rate,
iters):
    parameters = X.shape[1]
    # 暂时存储梯度下降后的参数值，以待同步更新
    temp = np.zeros((parameters, 1))
    cost = np.zeros(iters)  # 记录损失值
```

```
    for i in range(0, iters):
        error = np.dot(X, theta) - Y
        for j in range(0, parameters):
            X_j = X[:, j].reshape(-1, 1)
            temp[j] = theta[j] - learning_rate /
X.shape[0]\                         * np.sum(error * X_j)
        theta = temp
        cost[i] = cost_function(X, Y, theta)
    return theta, cost
```

批量梯度下降的一次迭代是对所有样本进行计算，由整个数据集确定的方向能够更好地代表样本总体，从而更准确地朝向极值所在的方向。且当目标函数为凸函数时，BGD一定能够得到全局最优。但批量梯度下降的缺点是当样本数目非常大时，每迭代一步都需要对所有样本计算，训练过程会很慢。

3. 随机梯度下降

与批量梯度下降类似，但不同之处在于，随机梯度下降是每次迭代使用一个样本来对参数进行更新，使得训练速度加快。

用数学的方式表示，则随机梯度下降每次迭代时，参数更新的公式为：

$$\theta_j := \theta_j - \alpha(h_\theta(x)^{(i)} - y^{(i)})x_j^{(i)}$$

实现的代码如下。

```
def stochastic_gradient_descent(X, Y, theta,
learning_rate, iters):
    parameters = X.shape[1]
    temp = np.zeros((parameters, 1))
    cost = np.zeros(iters)  # 记录损失值
    for i in range(0, iters):
        start = np.random.randint(0, X.shape[0], 1)[0]
        error = np.dot(X[start], theta)[0] - Y[start][0]
        for j in range(0, parameters):
            temp[j] = theta[j] - learning_rate \
                      * error * X[start][j]
        theta = temp
        cost[i] = cost_function(X, Y, theta)
    return theta, cost
```

随机梯度下降由于不是在全部训练数据上的损失函数，而是在每轮迭代中，随机优化某一条训练数据上的损失函数，因此每轮的参数更新速度会大大加快。但其缺点是准确度下降，即使在目标函数为强凸函数的情况下，SGD仍无法做到线性收敛。此外，对于非凸函数而言，随机梯度下降算法可能会收敛到局部最优，因为单个样本并不能代表全体样本的趋势。

4. L2-正则化批量梯度下降

正则化的作用是为了降低模型过拟合带来的影响，通过引入拉格朗日常数，缩小线性回归函数中的特征参数，从而使得模型更加平滑。

这里直接给出公式，具体原理参见吴恩达的机器学习课程，前文算法同。

L2-正则化线性回归的损失函数公式为：

$$J_\theta(x) = \frac{1}{2m}[\sum_{i=1}^{N}(h_\theta(x)^{(i)} - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2]$$

对于每次迭代，需要执行的参数更新为：

$$\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{N}(h_\theta(x)^{(i)} - y^{(i)})x_0^{(i)}$$

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

实现代码如下。

```
def L2_BGD(X, Y, theta, _lambda, learning_rate, iters):
    parameters = X.shape[1]
    temp = np.zeros((parameters, 1))
    cost = np.zeros(iters)  # 记录损失值
    regularization = 1 - learning_rate * _lambda \
                     / X.shape[0]
    for i in range(0, iters):
        error = np.dot(X, theta) - Y
        X_0 = X[:, 0].reshape(-1, 1)
        temp[0] = theta[0] - learning_rate / X.shape[0] \
                  * np.sum(error * X_0)
        for j in range(1, parameters):
            X_j = X[:, j].reshape(-1, 1)
            temp[j] = theta[j] * regularization \
                      - learning_rate / X.shape[0] \
                      * np.sum(error * X_j)
        theta = temp
        cost[i] = cost_function(X, Y, theta)
    return theta, cost
```

5. 线性回归分类器

在实现了线性回归模型之后，还需要一个分类器，对利用训练集训练出的特征参数和测试集中的特征进行计算，并对计算结果进行分类。

这里我们使用最简单的方式，即四舍五入。对于小数位不小于0.5的数向上取整，对小数位低于0.5的数向下取整。值得指出，前文提到标签分类数量总共有7类，最低为3.0，最高为9.0，因此，再进行取整之前，我事先规定对于高于9.0的计算结果，认为其标签是9.0；对于低于3.0的计算结果，认为其标签是3.0。

分类器实现代码如下。

```
# 线性分类器，X：要预测的数据，theta：拟合的参数
def classify(theta, X):
    y_hat = np.dot(X, theta).reshape(1, -1)[0] # 预测值
    for i in range(0, len(y_hat)):
        if y_hat[i] >= 9:
            y_hat[i] = 9
            continue
        if y_hat[i] <= 3:
            y_hat[i] = 3
            continue
        decimal = y_hat[i] - math.floor(y_hat[i])
        if decimal >= 0.5: # 向上取整
            y_hat[i] = math.ceil(y_hat[i])
        elif decimal < 0.5: # 向下取整
            y_hat[i] = math.floor(y_hat[i])
    return y_hat
```

实验结果分析

本部分主要论述本次实验过程中涉及到的所有对比实验、结果图像以及相应分析。将从学习率分析、两种梯度算法下RMSE与迭代次数关系、BGD与SGD收敛速度和分类准确度分析、L2正则化实现的BGD与普通BGD分类准确度分析几个角度展开论述。

1. 学习率分析

为了探究不同学习率下，损失函数值与迭代次数的关系，从而选择合适的学习率，我设计了一组对比实验。

对于两种梯度算法而言，在相同训练集下，随机梯度算法由于每次迭代需要随机选取一个值进行参数更新，因此训练得到的特征参数浮动程度比较大。而批量梯度下降算法由于每次迭代都是针对样本整体而言，结果更加稳定。因此，为了减小梯度算法带来的误差，在度量不同学习率下损失函数值与迭代次数的关系时，我选择批量梯度下降算法。

其次，对于学习率的选择，经过数次筛选，我最终选取了对比实验结果相对较好的一组学习率，即：

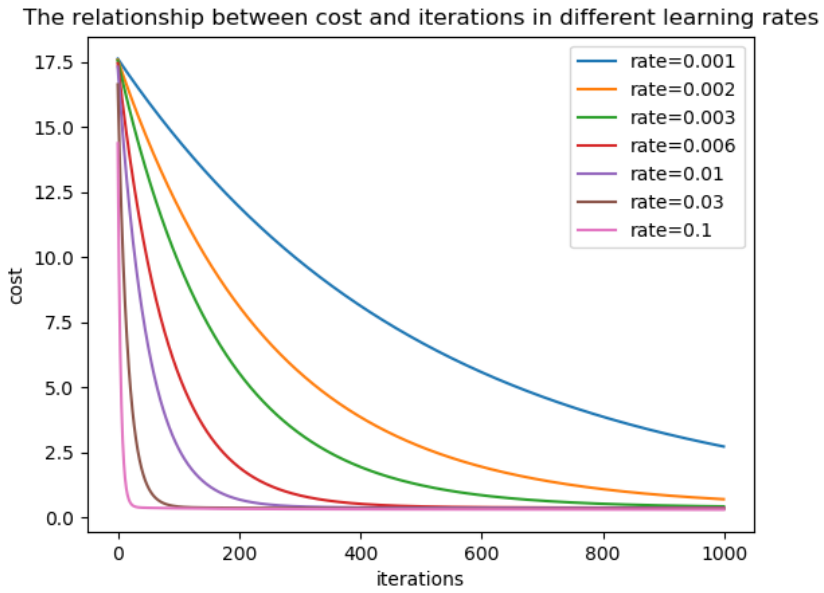
$$learning\_rate = \{0.001, 0.002, 0.003, 0.006, 0.01, 0.03, 0.1\}$$

我记录了迭代次数从1至1000时，采用BGD算法的损失函数值，以绘制实验结果图。实现代码如下。

```
# 对比学习率
def contrast_learning_rate():
    data = read_pkl("data/winequality-white.pkl")
    X, Y, theta = init_data(data)
    iters = 1000
    gradient = "BGD"
    learning_rate = [0.001, 0.002, 0.003, 0.006, \
```

```
                    0.01, 0.03, 0.1]

    plt.title("The relationship between cost and
iterations in different learning rates")
    for rate in learning_rate:
        _theta, cost = model(X, Y, theta, gradient, rate,
iters)
        label_ = "rate=" + str(rate)
        plt.plot(np.arange(iters), cost, label=label_)
    plt.legend(loc="best")
    plt.xlabel("iterations")
    plt.ylabel("cost")
    plt.show()
```

实验结果图如下。



从图像中我们可以看到，对于1000次的迭代次数而言，当学习率为0.001时，由于学习率过低，模型收敛速度过慢，即使1000次迭代，模型也没能完全收敛；而当学习率为0.03或0.1时，由于学习率过高，每次迭代下降梯度较大，模型收敛速度过快。因此，从图像中不难得出结论，一个比较合适的学习率选择区间是$[0.003, 0.006]$。

综上，为方便后续实验，后文实验选取的学习率均为`learning_rate=0.005`，迭代次数均为`iters=1000`。

2. 两种梯度算法下RMSE分析

均方根误差（RMSE，Root Mean Squard Error），是回归算法评价指标的一种。均方根误差公式如下。

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}$$

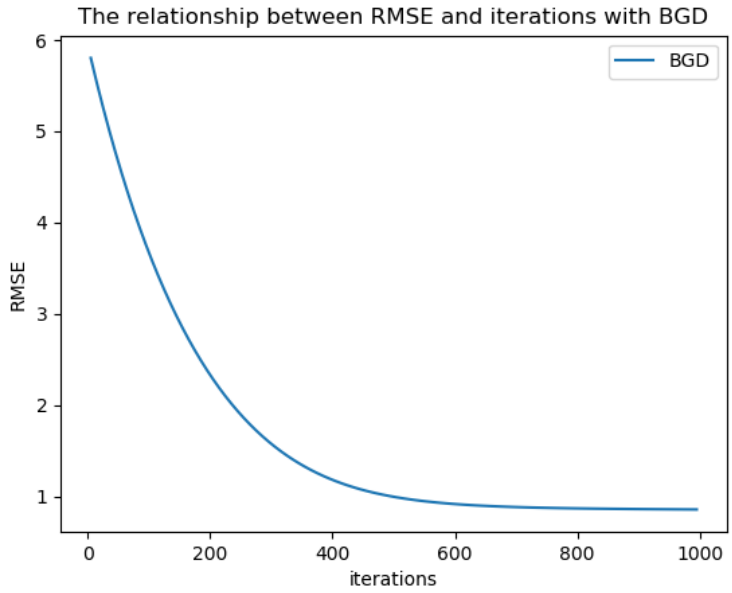
我的目的是分析BGD和SGD两种不同梯度算法下，RMSE和迭代次数的关系。起初，我选择的方法是留一法进行分析，代码如下。

```
# 使用留一法的RMSE
def RMSE_LeaveOne():
    data = read_pkl("data/winequality-white.pkl")
    X, Y, theta = init_data(data)
    iters_max = 20
    learning_rate = 0.005

    _iters = np.array([i for i in range(5, iters_max,
10)])
    _rmse = []  # 均方误差保存的列表

    for iters in _iters:
        _err = 0
        # 使用留一法
        for i in range(0, len(data)):
            if i % 500 == 0:
                print("now in iters: ", iters, " data: ",
i)
            test_X = X[i, :]
            test_Y = Y[i, :]
            train_X = np.delete(X, i, axis=0)
            train_Y = np.delete(Y, i, axis=0)
            res_theta, res_cost = model(train_X, train_Y,
theta, "BGD", learning_rate, iters)
            _err += (test_Y[0] - np.dot(test_X, res_theta)
[0]) ** 2
        _rmse.append(math.sqrt(_err / len(data)))
    return _rmse,_iters
```

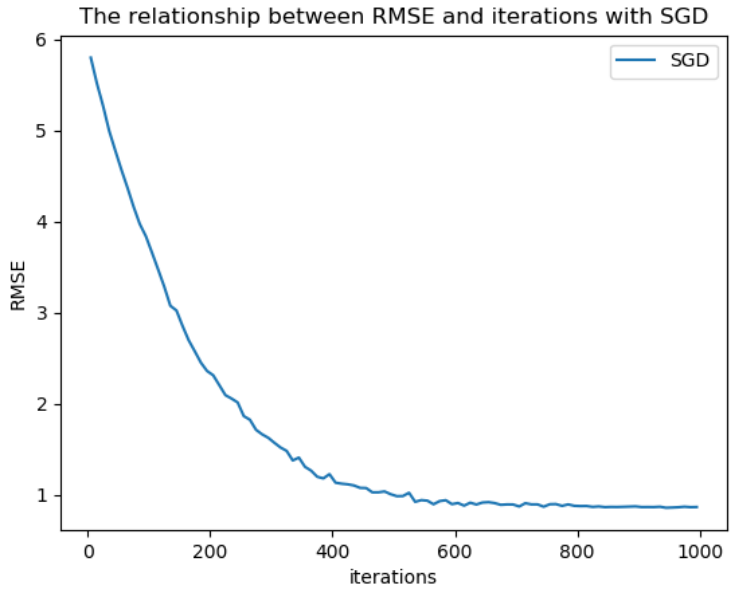
但随着迭代次数的增加，留一法执行时间过长，运行效率过低，比如当`iters_max=100`时，执行的时间就已经长达1个多小时，可想而知迭代次数增大以后，代码运行的恐怖的时间成本。

因此，为了减小时间成本，我最终选择了我自己按照7：3比例划分的数据集。这里为了减小篇幅，代码部分不再赘述，详见`experiment.py`中的`RMSE()`函数。

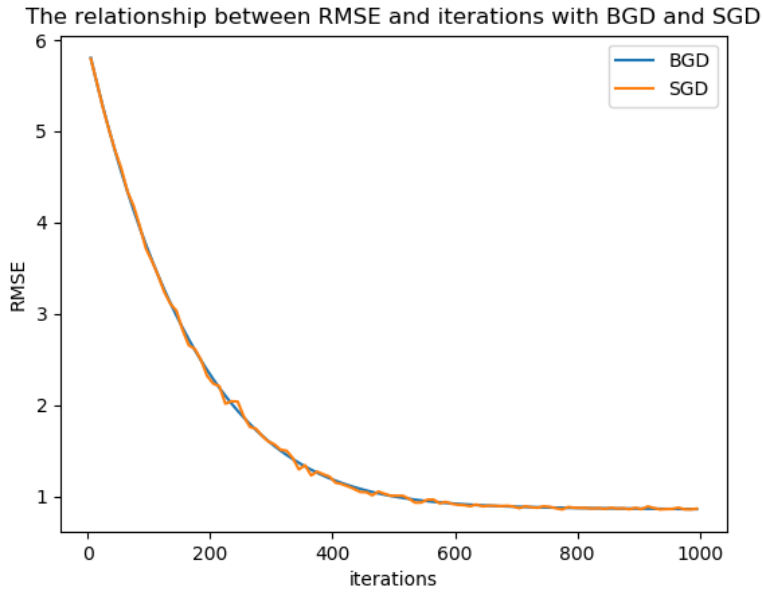
对于批量梯度下降算法，实验结果图如下。



对于随机梯度下降算法，实验结果图如下。



对比两个图象，不难发现，两种算法的收敛曲线在收敛时所需的迭代次数基本一致，但随机梯度下降算法由于其每次迭代选取点的随机性，因而收敛曲线存在一定的震荡空间。为了进一步凸显出两种算法收敛曲线的差异，我将两条曲线置于同一张图像中，结果图如下。



从这个图像中可以看到，两条收敛曲线基本重合，但SGD有一定程度上的浮动。仔细观察图像，当收敛次数$iters = 500$时，两条曲线基本接近于收敛状态，因此我们可以认为，当收敛次数高于500时，算法处于收敛状态。为了使得随机梯度下降算法曲线的波动幅度更加明显，这里我又做了一组对比实验。即当两条曲线处于收敛状态（即$iters > 500$）时，RMSE同迭代次数的关系图如下。

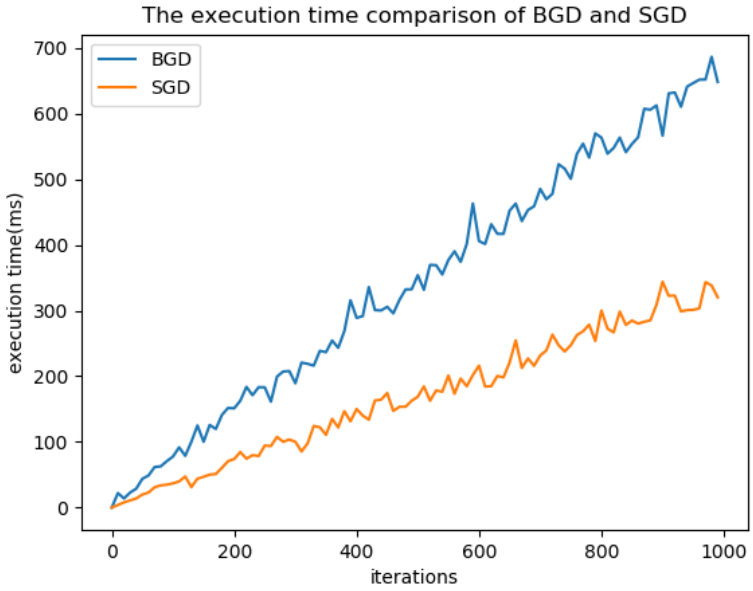
从这个图像中，可以非常明显地观察到，对于随机梯度下降算法，由于迭代样本选取的随机性所引起的收敛曲线的波动情况。

3. BGD与SGD深入对比

前文分析了对于两种梯度下降算法，RMSE与迭代次数关系的差异，本部分将从算法收敛速度和分类准确度两个方面，深入分析BGD算法同SGD算法的差异。（注：下文实验代码不再给出，详见`experiment.py`文件）

执行时间

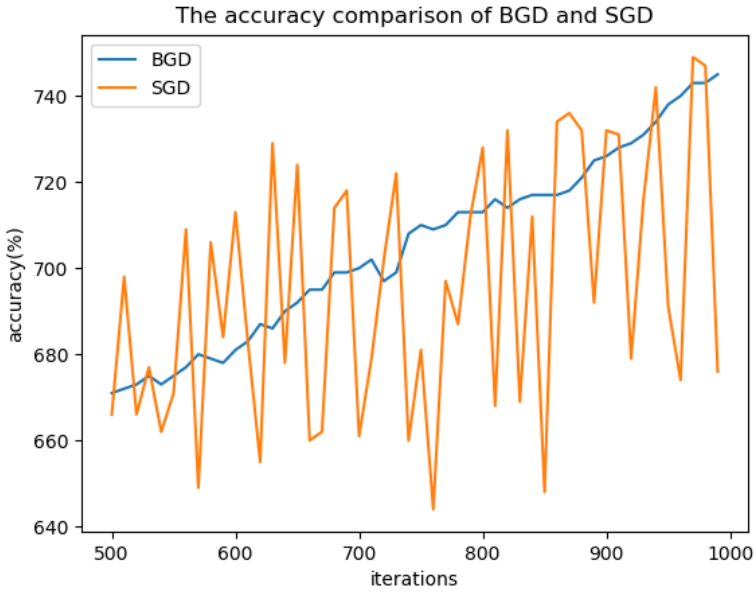
首先，对于二者的收敛速度，我使用python中的`time`包记录下对于相同训练集，两种梯度下降算法的执行时间同迭代次数的关系，实验结果图如下。



从上述实验结果图中，可以很明显地看到，对于相同数据集，BGD算法执行需要的时间成本远高于SGD，二者的执行时间大约成2倍的关系。原因其实也在前文给出了说明，即对于批量梯度下降，当样本数目非常大时，每迭代一步都需要对所有样本计算，训练过程会很慢。而随机梯度下降由于不是在全部训练数据上的损失函数，而是在每轮迭代中，随机优化某一条训练数据上的损失函数，因此每轮的参数更新速度会大大加快。

分类准确度

对于二者的分类准确度，利用前文实现的线性回归分类器，我设计了对比实验，计算了当模型算法收敛时（即收敛次数高于500时），在不同迭代次数下，分类准确度的差异情况。实验结果图如下。

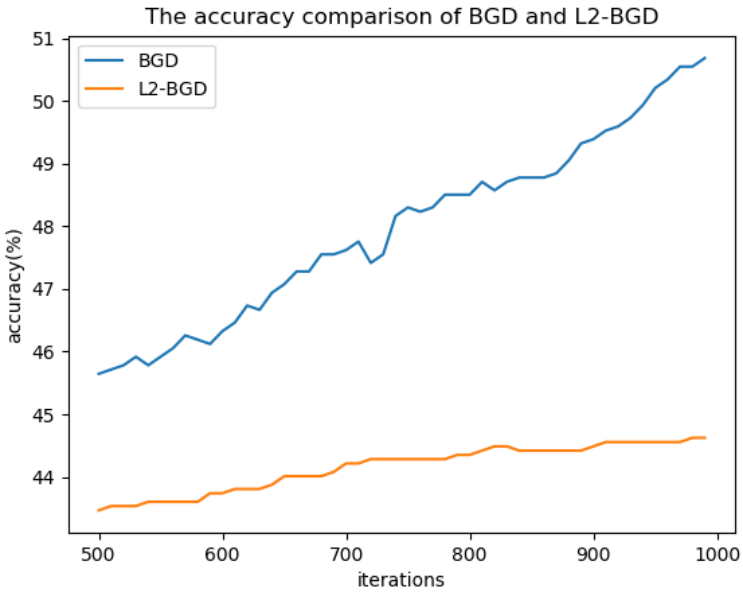


从图中我们不难发现，整体而言，BGD算法的分类准确度在稳定程度上要高于SGD算法的分类准确度，且在分类准确度的平均水平上，前者也要优于后者。

4. BGD与L2正则化后的BGD对比

前文实现了L2正则化后的批量梯度下降算法，为了比较两者的性能差异，和前文相同，我选择了分类准确度这一指标进行度量，通过比较对两种算法在收敛时的分类准确度，进而得出实验结果。

实验结果图如下。



需要指出，这里，我定义的拉格朗日常数的值为$\lambda = 100$。从图像中可以观察到，使用了L2正则化后的批量梯度下降，在分类准确度上要明显低于未使用正则化的批量梯度下降算法。出现这种状况的原因是，L2正则化使得线性回归函数中特征参数θ的值减小，降低了模型的过拟合，使得线性回归函数的拟合曲线更为平滑，因此自然在分类准确率上有所降低。但这种做法使得模型结构风险最小化，提高了模型的普适性，即使对于更大的外部数据集，也能取得较好的效果，避免了过拟合的发生。

以上是本次实验报告内容，感谢批阅！
