实验二——回归计算

基本原理

回归是监督学习的一个重要问题，回归用于预测输入变量和输出变量之间的关系，在进行多元回归任务时，可以将整个流程划分为多个阶段：数据准备，损失函数定义，数据拟合与评估；在这一任务中，首先将会依照8：2的比例将数据集划分为训练集与测试集，而后定义损失函数为MSE函数，其具有以下形式：

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \theta_0 - \theta_i x^{(i)})^2$$

以此结果以及其它指标作为对模型的评估指标。

而对于训练过程，则使用SGD方法进行，其目标为为最小化所有训练样本的损失函数，它每一次从训练样本中取出某一batch，而后根据这一batch的平均梯度确定该epoch的优化方向，并根据预先设定的学习率进行梯度的更新过程，最终得到可以较好拟合数据的模型。

实现代码

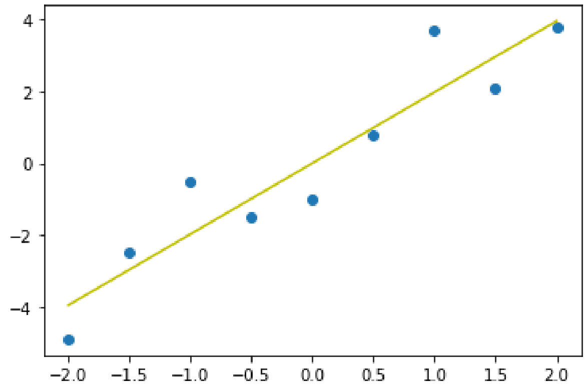
在实现过程中，首先对最小二乘解回归方程进行了实现，其基本代码如下：

```
def compute_cost(w,b,X,Y):
    total_cost=0
    M =len(X)
    for i in range(M):
        x=X[i]
        y=Y[i]
        total_cost += (y-w*x-b)**2
    return total_cost/M
def fit(X,Y):
    M = len(X)
    x_bar=np.mean(X)
    sum_pro= 0
    sum_squ=0
    sum_delta =0
    for i in range(M):
        x=X[i]
        y=Y[i]
        sum_pro += y*(x-x_bar)
        sum_squ += x**2
    w = sum_pro/(sum_squ-M*(x_bar**2))
    for i in range(M):
        x=X[i]
        y=Y[i]
        sum_delta += (y-w*x)
    b = sum_delta / M
    return w,b
```

其计算结果如下：

```
w=1.9766666666666667     b=-2.4671622769447922e-17
cost= 0.9257592592592595
```

而对其作图的结果为;



在本次实验中，针对winequality-white.csv中的各个数据进行回归计算，首先对数据集进行划分操作，如下：

```
# data prepared
quality_factor = pd.read_csv('winequality-white.csv')
data,label=quality_factor.iloc[:,:-2],quality_factor.iloc[:,-1]
data,label=np.array(data),np.array(label)
data=np.hstack((np.ones((data.shape[0],1)), data))
x_train, x_test,  y_train, y_test = train_test_split(data, label, test_size =
0.2, random_state = 7)
```

在这里，将数据集划分为80%的训练集以及20%的测试集，并对各个部分进行了相应的标签，数据的处理，完成了对数据集的基本封装处理：

随后对损失函数以及梯度计算进行处理，得到了以下结果：

```
def loss_fuc(theta, X_b, y):
    return np.sum((y - X_b.dot(theta))**2) / len(X_b)#除以样本数（多少行）
```

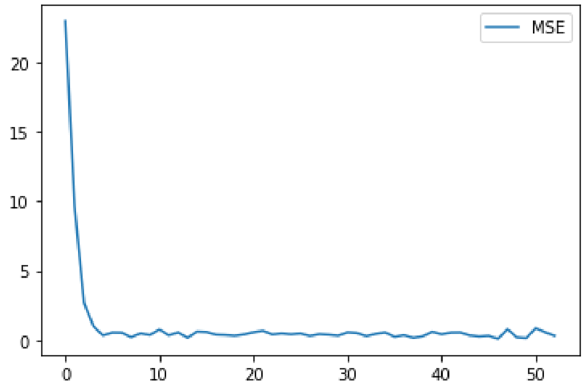
```
def gradient_cal(theta, X_b, y):
    res = np.empty(len(theta))
    #res[0] = np.sum(X_b.dot(theta) - y)
    for i in range(0, len(theta)):
        res[i] = (X_b.dot(theta) - y).dot(X_b[:,i])
    return res * 2 / len(X_b)
```

在这一操作下，对单个批量的基本数据计算进行矩阵计算，得到了各个batch中的损失函数以及梯度的计算方法，为SGD的计算提供了素材。

```
def
sgd(initial_weight,train_set,train_label,epsilon=0.01,epoch=10,batch_size=8,lear
ning_rate=0.00001):
    weights=initial_weight
    for i in range(epoch):
        input_x=np.empty((batch_size,train_set.shape[1]))
        input_y=[]
```

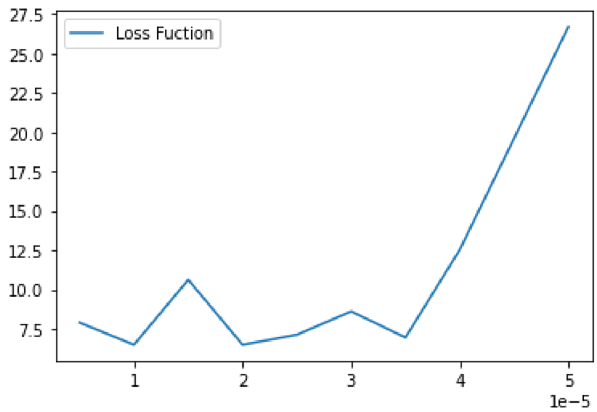
```
        for item in range(batch_size):
            select_data_index=np.random.randint(len(train_set))
            input_x[item,:]=train_set[select_data_index]
            input_y.append(train_label[select_data_index])
        gradient_set=gradient_cal(weights,input_x,input_y)
        if(abs(loss_fuc(weights, input_x, input_y) - loss_fuc(weights-
gradient_set*learning_rate, input_x, input_y)) < epsilon):
            break
        loss_store.append(loss_fuc(weights-gradient_set*learning_rate, input_x,
input_y))
        weights=weights-gradient_set*learning_rate
    return weights
```

在SGD中，完成了单个批次数据的生成以及对所选择batch进行优化，迭代该过程即可得到最终结果以及完成对MSE函数的可视化操作：



可以看出，整个过程中，MSE保持着下降的趋势，但是在个别点，其损失函数会上升，这是由于SGD算法自有的随机性造成的，这一特性也可以帮助其脱离局部最优解，该实验结果是符合一贯认知的。

而对于学习率的选择，可以看到：



经过多次实验，可以发现当学习率的取值接近$2 * 10^{-5}$时，取得了较优结果，而随着学习率的增大，MSE将会在不断的"震荡"过程中，达到非常大的值，这也是SGD算法中可以进行进一步优化的地方。

最终，通过sklearn的回归模型进行计算，完成了对比试验：

```
LinearRegression()
model_ore=np.around (model.predict(x_test))
print(model_ore)
print(y_test)
count_model=0
for i in range(len(x_test)):
    if(model_ore[i]==y_test[i]):
        count_model=count_model+1
print(count_model/len(model_ore))
```

```
0.5217687074829932
```

对比可知，在对最终结果进行四舍五入的操作时，使用现有模型可以达到超过50%的准确率，明显高于现有手写模型，这一方面是由于`SGD`方法具有一定的不稳定性，另一方面则是对手写模型的超参数的选择有待进一步优化。

对于`learning-rate`的搜索，除了基本的多次查找外，还可以通过诸如自适应方法，或者引入Moteum的概念进行进一步的优化，这也是回归算法在后续操作中值得注意的地方。

总结

在这一实验中，完成了基本，中级要求；实现了基本的最小二乘法完成回归任务；对所给定数据集的回归任务实现了随机梯度下降算法，均取得了较为良好的结果，而针对所提出的要求，求出了给定数据集的最小二乘解并计算出训练误差，而针对多元回归任务，构造完成了线性回归模型并对多种学习率进行检验，画出了MSE曲线，选择出最佳的学习率