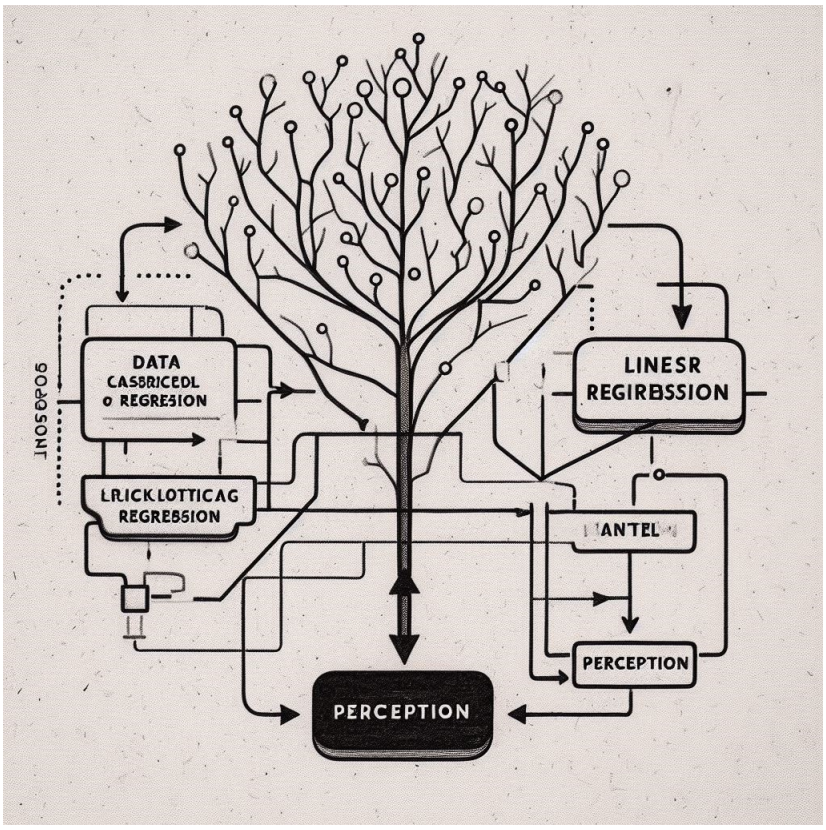
数据分类或线性回归中的过拟合分析

机器学习作业2

姓名：游霄童　学号：210092

Created： 2023/10/22



UCI _Sonar数据集介绍

> *UCI 机器学习库（UCI Machine Learning Repository）是一个广泛用于机器学习和数据挖掘研究的资源，其中包含了许多开源数据集，Sonar数据集就是其中之一。Sonar数据集（也称为声纳数据集）是一个经典的二分类问题数据集，用于声纳信号处理和目标检测领域的研究。*
>
> *网址：https://archive.ics.uci.edu/ml/machine-learning-databases/undocumented/connectionist-bench/sonar/sonar.all-data*

Sonar数据集的背景信息：

Sonar数据集是通过声纳传感器收集的声纳信号样本，用于区分两种不同类型的目标：岩石（Rock）和金属（Mine）。这个问题的背景是，声纳传感器被用于在水下探测和识别物体，因此这个数据集的研究对于水下目标检测非常重要。

数据集特点：

1. **样本数量**：Sonar数据集包含了208个样本。
2. **特征**：每个样本由60个特征组成，这些特征是声纳传感器在不同方向上接收到的信号的幅度。
3. **类别**：目标变量是二分类的，分为"R"（岩石）和"M"（金属）两类。
4. **用途**：Sonar数据集常用于分类算法的性能评估和比较，研究者使用该数据集来开发和测试各种分类算法，以区分声纳信号中的不同目标类型。

数据集应用：

Sonar数据集的应用领域主要集中在模式识别、机器学习算法的评估、特征选择等方面。研究者常常使用Sonar数据集来验证新的分类算法、特征提取方法和降维技术的有效性。

在机器学习中，研究人员可以将Sonar数据集分为训练集和测试集，使用训练集训练模型，并在测试集上评估模型的性能。这个数据集也被广泛用于教育和培训，帮助学生和初学者理解和实践分类算法的应用。

Sonar数据集：

	attribute_1	attribute_2	...	attribute_58	attribute_59	attribute_60	Class
1	0.02	0.0371	...	0.0084	0.009	0.0032	Rock
2	0.0453	0.0523	...	0.0049	0.0052	0.0044	Rock
3	0.0262	0.0582	...	0.0164	0.0095	0.0078	Rock
4	0.01	0.0171	...	0.0044	0.004	0.0117	Rock
...	...	...	...	...	...	...	...
208	0.026	0.0363	...	0.0036	0.0061	0.0115	Mine

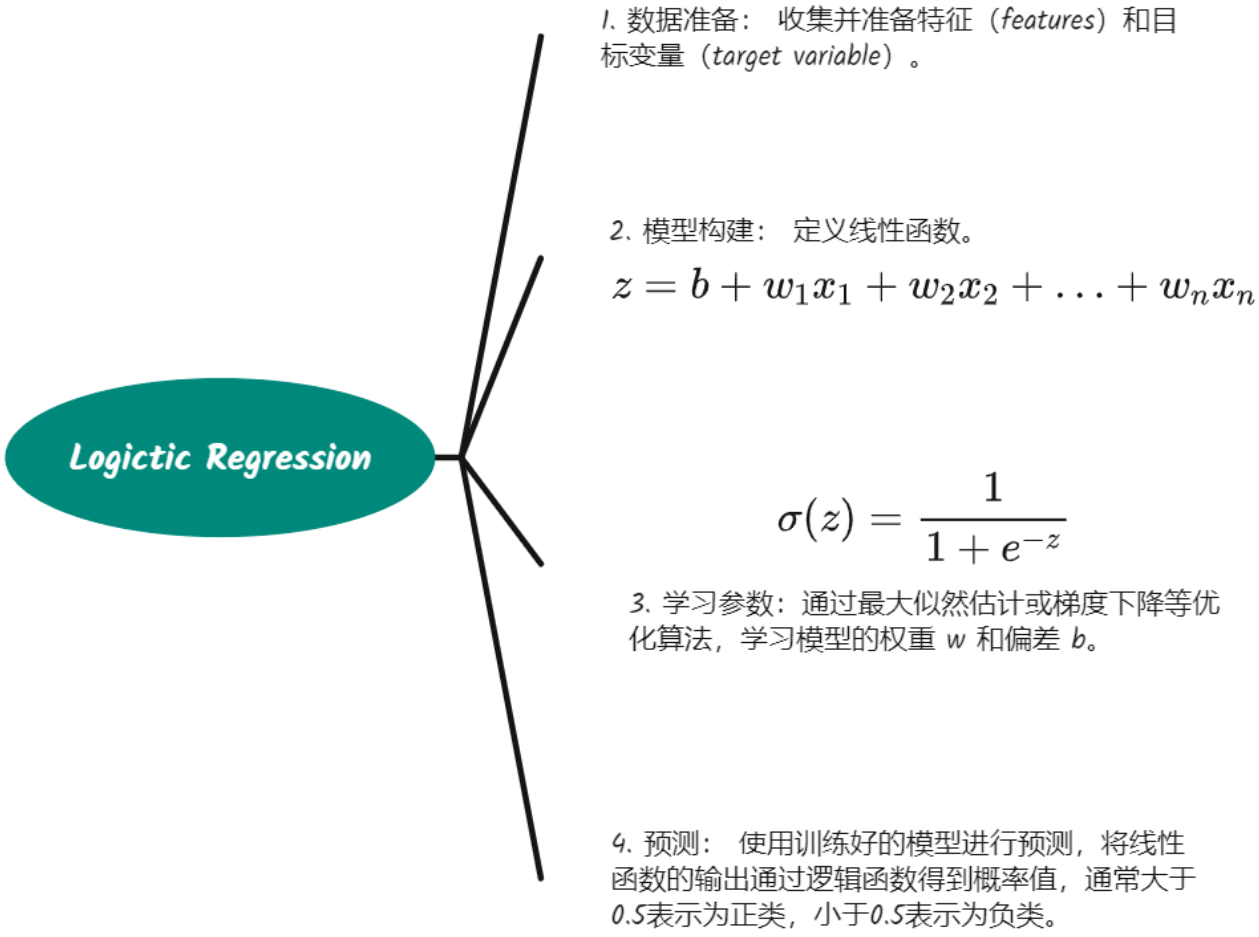
摘要

当处理二分类问题时，Logistic回归、线性回归和感知器模型都是常用的基本机器学习算法。以下是它们的简要流程：

数据分类或回归			
	Logictics Regression	Liner Regression	Perception
	1. 数据准备： 收集并准备特征（features）和目标变量（target variable）。	1. 数据准备： 收集并准备特征和目标变量。	1. 数据准备： 收集并准备特征和目标变量。
	2. 模型构建： 定义线性函数。 $z = b + w_1x_1 + w_2x_2 + \ldots + w_nx_n$	2. 模型构建： 定义线性函数。例如： $y = b + w_1x_1 + w_2x_2 + \ldots + w_nx_n$	2. 模型构建： 定义感知器的激活函数（通常为阶跃函数，即根据输入值的正负来输出0或1），并定义线性函数。例如： $y = b + w_1x_1 + w_2x_2 + \ldots + w_nx_n$
	$\sigma(z) = \frac{1}{1+e^{-z}}$ 3. 学习参数：通过最大似然估计或梯度下降等优化算法，学习模型的权重 w 和偏差 b。	3. 学习参数： 通过最小二乘法或梯度下降等优化算法，学习模型的权重 w和偏差 b)。	3. 学习参数： 通过梯度下降等优化算法，学习模型的权重 \(w\) 和偏差 \(b\)。
	4. 预测： 使用训练好的模型进行预测，将线性函数的输出通过逻辑函数得到概率值，通常大于0.5表示为正类，小于0.5表示为负类。	4. 预测： 使用训练好的模型进行预测，通常通过设置阈值将连续输出映射为二分类结果。	4. 预测： 使用训练好的模型进行预测，将线性函数的输出通过激活函数映射为0或1。

本次作业详细编写了使用Logistic回归的Sonar二分类问题，其他如Liner回归和感知器则使用库函数进行进行计算，试比较三者的差异，并结合数据分析其中的过拟合现象。

一：Logistic Regression



在代码

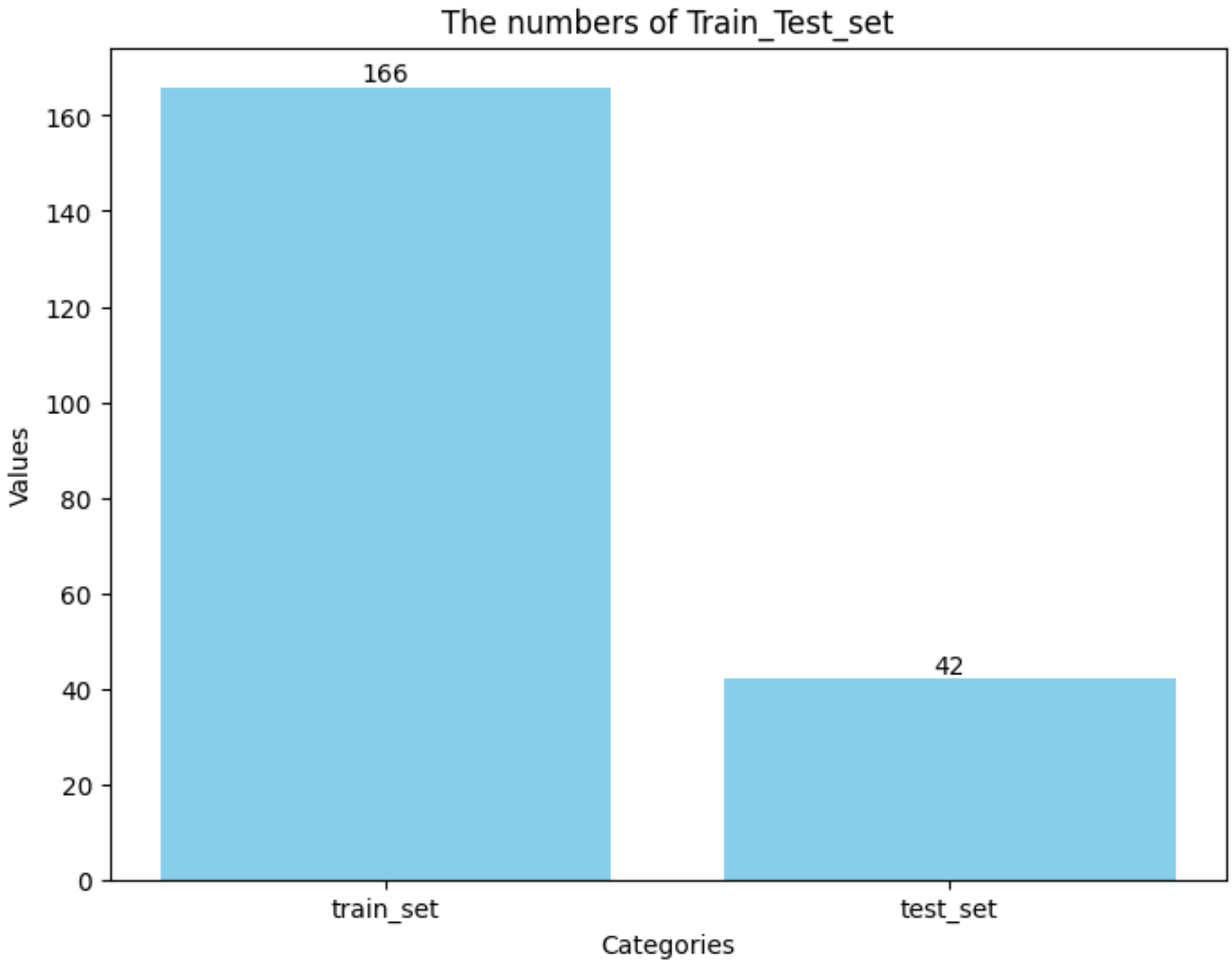
```
x = origin_data.iloc[:,:59]
y = origin_data.Class
y_d=np.where(y=='Rock',1,0) #将数据标签bool化，Rock为1，Mine为0
train_set_x,test_set_x,train_set_y,test_set_y=train_test_split(x,y_d,tes
t_size=0.2,random_state=42) #运train_test_split将数据划分为8：2的训练：测试集
train_set_x=train_set_x.T
test_set_x=test_set_x.T
train_set_y=train_set_y.reshape(1,166)
test_set_y=test_set_y.reshape(1,42)
```

上述代码进行了数据预处理和划分，简洁描述如下：

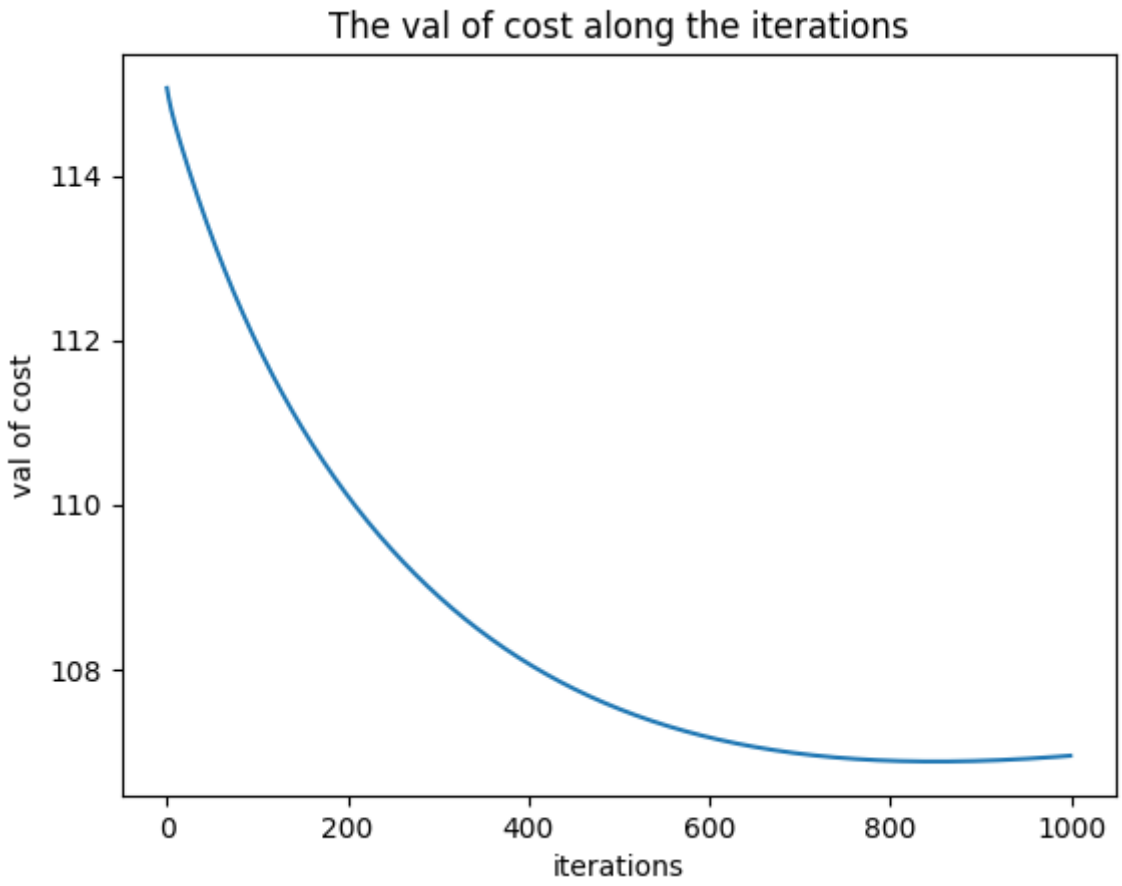
1. **数据预处理**：
    - `x = origin_data.iloc[:,:59]`：选择前60列作为特征。
    - `y = origin_data.Class`：选择`Class`列作为目标变量。
    - `y_d = np.where(y == 'Rock', 1, 0)`：将"Rock"标签设为1，"Mine"标签设为0。
2. **数据划分**：
    - 使用`train_test_split`将数据划分为训练集和测试集（80%训练，20%测试）。
    - 对训练集和测试集的特征数据进行转置，以符合部分机器学习库的输入格式要求。
    - 对训练集和测试集的目标变量数组进行reshape，确保维度匹配。

结果如下图所示：

将Sonar数据集以Train : Test=8 : 2的比例划分成了两个集合，最终得到的数量为166与42。



第一次我将迭代次数设置为1000次，学习率即步长设置为0.05，得到的损失函数绘制的曲线图如下所示：



得到的结果如下：

$$Accuracy = 0.8571428571428572$$

一*、Logistic Regression详细分析：

logistic Regression代码可分为以下几个部分：

1. **初始化参数**

首先定义了sigmoid激活函数,然后定义了initialize_with_zeros函数来初始化权重w和偏置b为0。

2. **前向传播**

在propagate函数中实现了前向传播过程。先计算z = w^T x + b,然后通过sigmoid函数计算激活值a。
再根据a和真实标签y计算交叉熵损失函数cost。同时通过求导计算梯度dw和db。

3. **反向传播**

在optimize函数中实现了梯度下降法来更新参数。具体步骤是:
(1) 计算梯度grads
(2) 用学习率乘以梯度更新参数
(3) 重复更新直到达到迭代次数

4. **模型评估**

最后,用学得的参数w和b对训练集和测试集进行预测,计算准确率评估模型效果。

5. **返回字典**

为了方便检查,将所有参数、超参数等都保存到字典d中并返回。

总结来说,这个代码按照典型的前向传播、反向传播、参数更新的流程实现了logistic regression模型。
关键在于用sigmoid函数建模、交叉熵损失函数以及梯度下降法优化参数。返回字典方式方便检查模型的详细信息。

我将迭代次数设置为1000，学习率设置为0.3

过拟合分析：

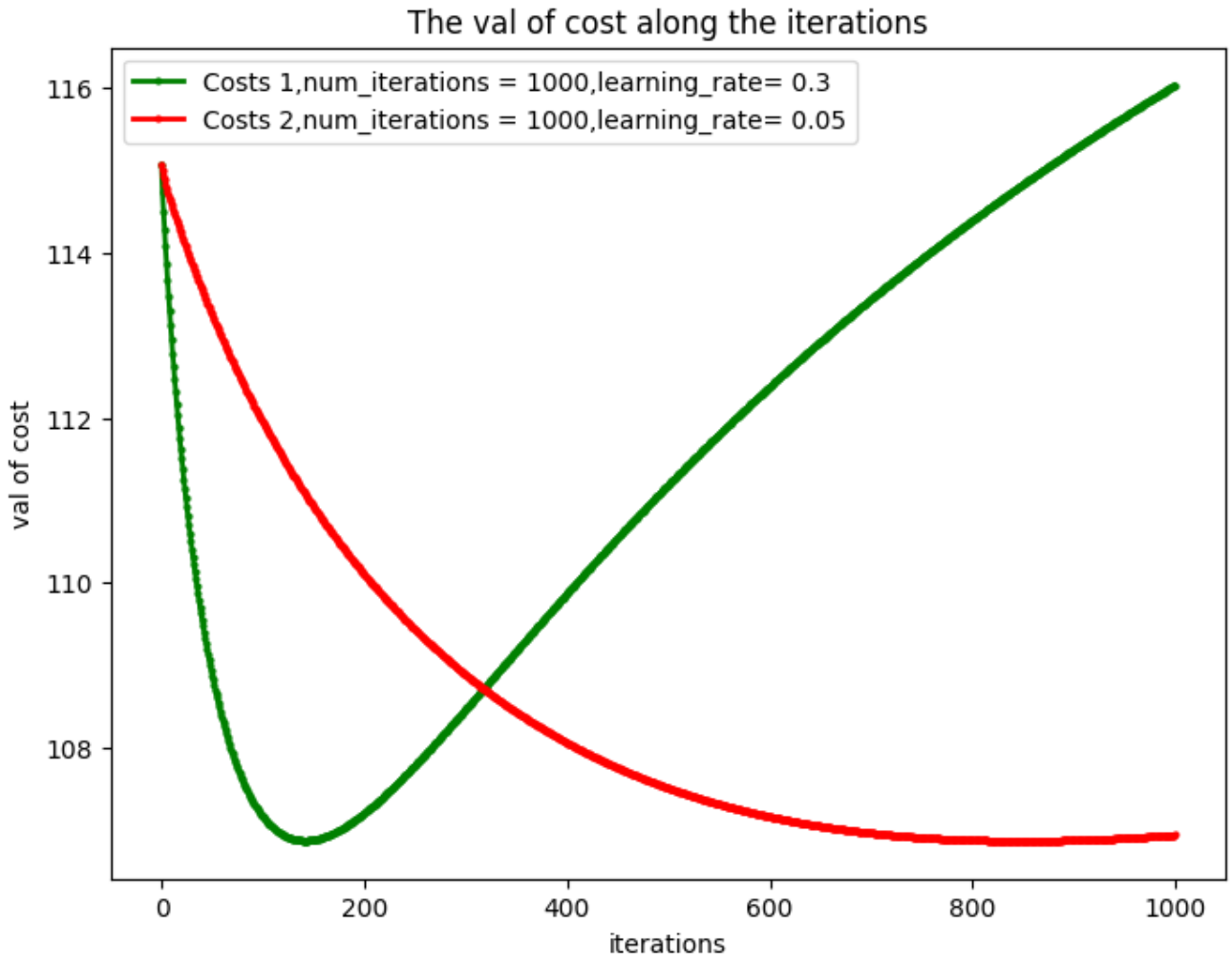
根据迭代过程可看出：

1. 损失函数值在迭代过程中呈下降趋势。开始下降很快,后期趋于平缓。这与梯度下降算法的期望一致。
2. 前50次迭代损失函数下降迅速,之后下降速度放缓,基本上每50次为一个阶段。这说明随着迭代进行,参数逐步逼近最优值,梯度越来越小,下降速度越来越慢。
3. **500次迭代后,训练集上的准确率约为81.3%,测试集上的准确率约为85.7%。这说明有一定的过拟合发生,训练集上的表现略优于测试集。**
4. 85.7%的测试准确率可以说比较理想,说明在这个数据集上,logistic regression模型拟合效果较好,500次迭代基本达到较优的模型状态。
5. 如果继续迭代,准确率可能还有提升空间,但改进幅度应该不大,过拟合的风险会加大。需要通过Early Stopping等技巧来选择合适的迭代次数。
6. 可以通过调节学习率、正则化等手段进一步提高模型泛化能力,减少过拟合。

总体来说,这个模型训练过程比较稳定,迭代过程与理论吻合,达到了较好的拟合效果。但还有优化的空间,需要根据实际情况采取技巧来控制过拟合,提升泛化能力。

两次比较：

与第一次我将迭代次数设置为1000次，学习率即步长设置为0.05的损失函数图像进行比较



两次迭代过程都学习到了最优，所以准确率相同为：

$$Accuracy = 0.8571428571428572$$

根据两次迭代结果的对比,可以发现:

1. 学习率不同导致损失函数下降速度不同。

第一次学习率为0.05,损失函数在前50次迭代就有明显下降;

第二次学习率为0.3,损失函数在前10次就下降很快。

这与学习率大小正相关的原理一致。

2. 不同学习率导致最终 Loss 不同。

第一次终止Loss约为116;第二次终止Loss约为120。

这是因为第二次学习率过大,可能导致 Loss 无法收敛到较优值。

3. 准确率基本相当。

两次在训练集和测试集上的准确率很接近,都在81%-86%之间。

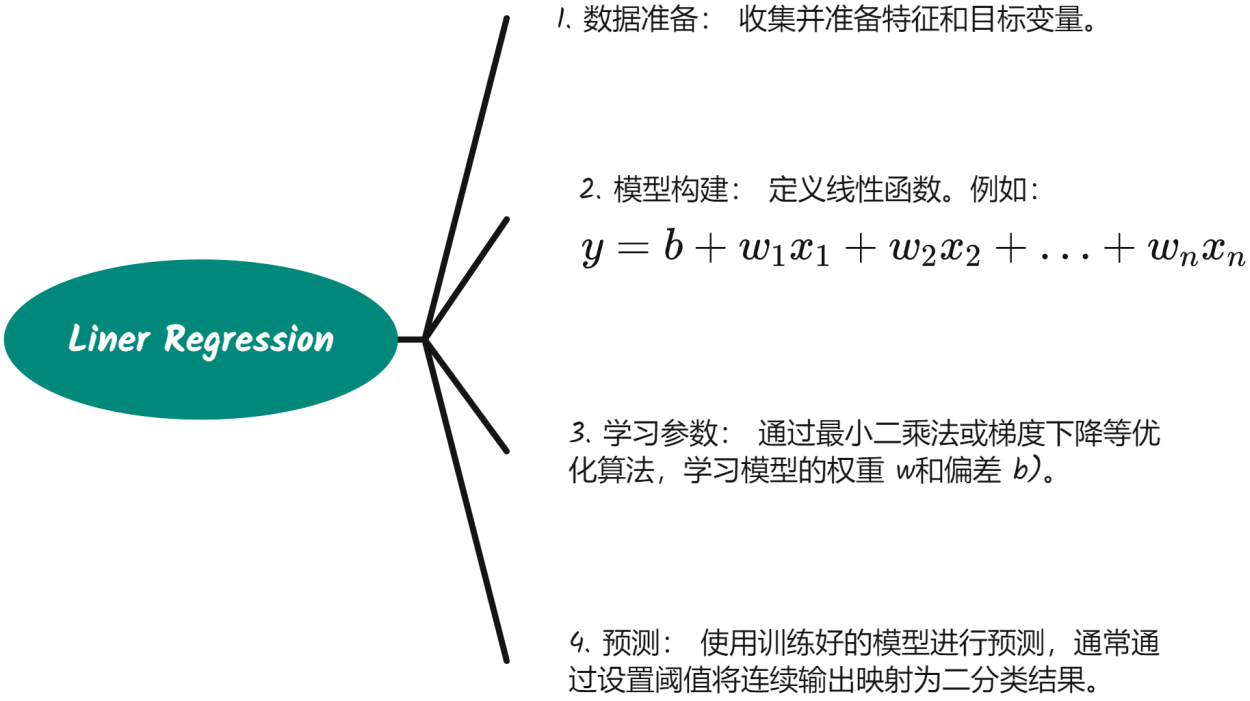
说明不同学习率并没有对模型泛化能力产生明显影响。

4. 第二次迭代次数更少。

由于学习率大,收敛速度更快,因此达到相近Loss需要的迭代次数更少。

总结来说,学习率不同影响的是损失函数下降的速度,但对模型准确率的影响不大。需要选择合适的学习率,既要保证收敛速度,也要控制Loss收敛到较优。本例中第二次学习率可能过大,导致Loss并未充分优化。

二：Liner Regression



Jupyter 代码如下：

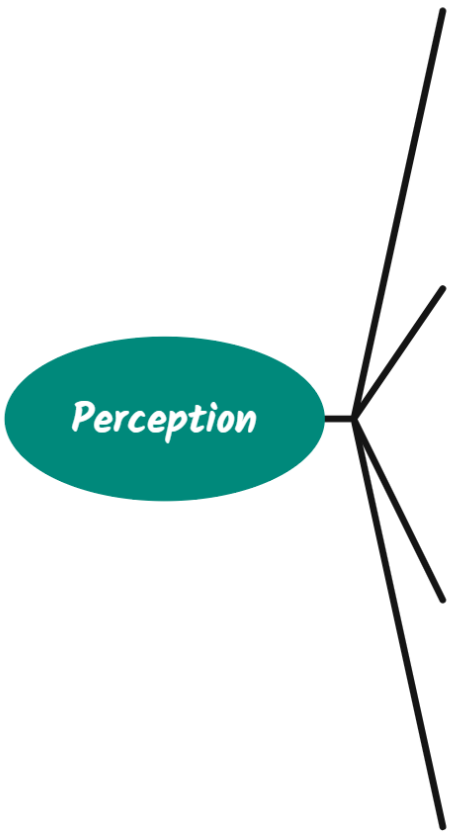
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import accuracy_score
from sklearn.datasets import fetch_openml
%matplotlib inline
#%%
origin_data = pd.read_csv("sonar_csv.csv")
#%%
origin_data.head()
#%%
x = origin_data.iloc[:,:59]
y = origin_data.Class
y_d=np.where(y=='Rock',1,0) #将数据标签bool化，Rock为1，Mine为0
train_set_x,test_set_x,train_set_y,test_set_y=train_test_split(x,y_d,tes
t_size=0.2,random_state=42) #运
# 作出数量图
categories = ['train_set', 'test_set']
values = [166, 42]

# 创建柱状图
plt.figure(figsize=(8, 6))
plt.bar(categories, values, color='skyblue')
bars = plt.bar(categories, values, color='skyblue')
for bar in bars:
```

```
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, round(yval, 1),
ha='center', va='bottom')

# 添加标题和标签
plt.title('The numbers of Train_Test_set')
plt.xlabel('Categories')
plt.ylabel('Values')

# 显示柱状图
plt.show()

#%%
train_set_x.shape,train_set_y.shape,test_set_x.shape,test_set_y.shape
#%%

# 创建线性回归模型（不推荐用于分类问题）
model = LinearRegression()

# 训练模型
model.fit(train_set_x, train_set_y)

# 在测试集上进行预测
predictions = model.predict(test_set_x)

# 将回归结果转换为分类标签（0或1）
predictions = np.round(predictions)

# 计算准确率
accuracy = accuracy_score(test_set_y, predictions)
print(f'Accuracy on test set of Liner regression: {accuracy *
100:.2f}%')
#%%

#%%
predictions
```

结果如下图所示：

$$Accuracy = 0.6904761904761905$$

三：Perception



1. 数据准备： 收集并准备特征和目标变量。

2. 模型构建： 定义感知器的激活函数（通常为阶跃函数，即根据输入值的正负来输出0或1），并定义线性函数。例如：

$$y = b + w_1x_1 + w_2x_2 + \ldots + w_nx_n$$

3. 学习参数： 通过梯度下降等优化算法，学习模型的权重 \(w\) 和偏差 \(b\)。

4. 预测： 使用训练好的模型进行预测，将线性函数的输出通过激活函数映射为0或1。

Jupyter 代码如下：

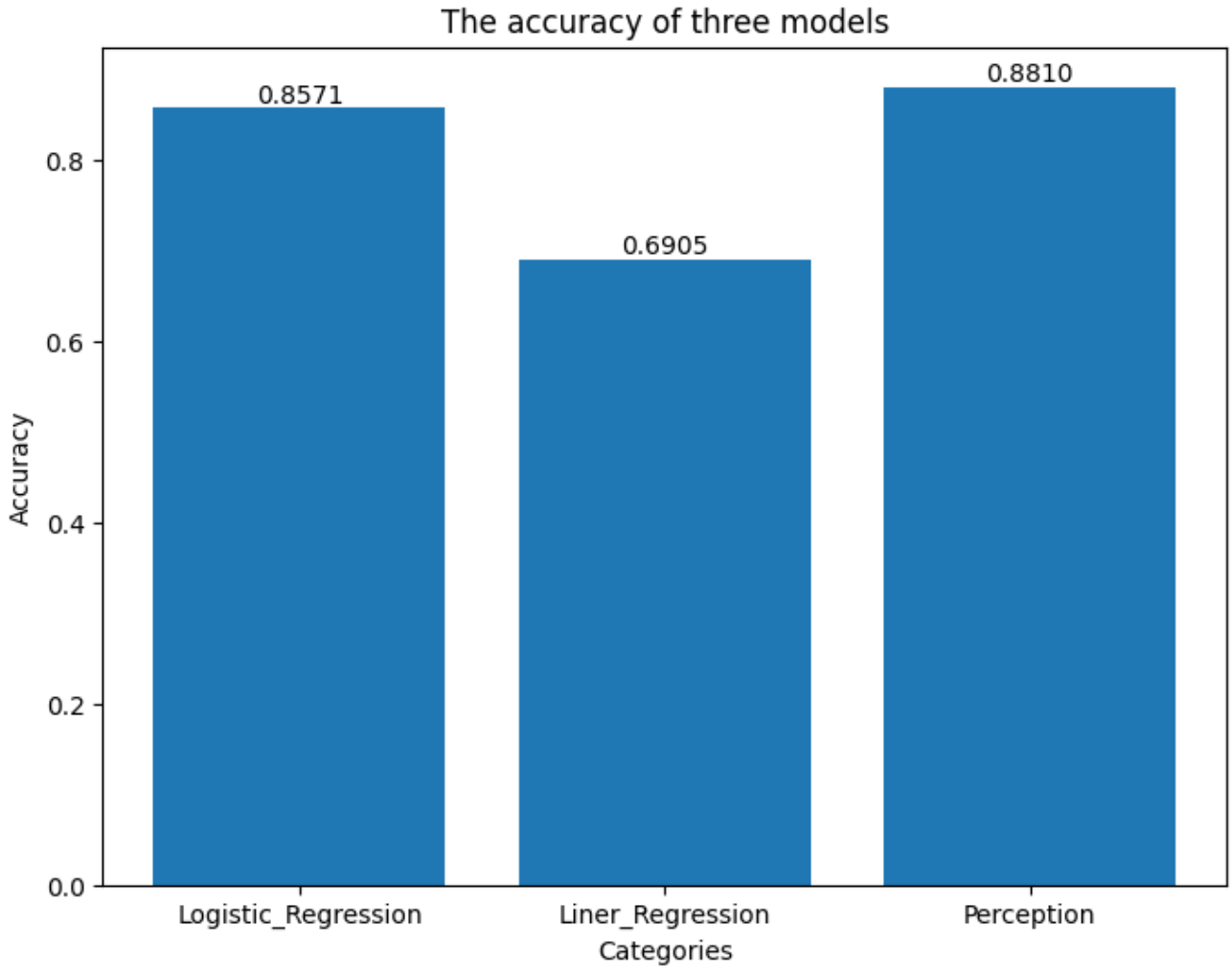
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score
from sklearn.datasets import fetch_openml
#%%
origin_data = pd.read_csv("sonar_csv.csv")
#%%
origin_data.head()
#%%
x = origin_data.iloc[:,:59]
y = origin_data.Class
y_d=np.where(y=='Rock',1,0) #将数据标签bool化，Rock为1，Mine为0
train_set_x,test_set_x,train_set_y,test_set_y=train_test_split(x,y_d,tes
t_size=0.2,random_state=42) #运train_test_split将数据划分为8：2的训练：测试集
#train_set_x=train_set_x.T
#test_set_x=test_set_x.T
#train_set_y=train_set_y.reshape(1,166)
#test_set_y=test_set_y.reshape(1,42)
```

```
#%%
# 作出数量图
categories = ['train_set', 'test_set']
values = [166, 42]

# 创建柱状图
plt.figure(figsize=(8, 6))
plt.bar(categories, values, color='skyblue')
bars = plt.bar(categories, values, color='skyblue')
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, round(yval, 1),
ha='center', va='bottom')

# 添加标题和标签
plt.title('The numbers of Train_Test_set')
plt.xlabel('Categories')
plt.ylabel('Values')

# 显示柱状图
plt.show()

#%%
# 创建感知机模型
model = Perceptron()

# 训练模型
model.fit(train_set_x, train_set_y)

# 在测试集上进行预测
predictions = model.predict(test_set_x)

# 将回归结果转换为分类标签（0或1）
predictions = np.round(predictions)

# 计算准确率
accuracy = accuracy_score(test_set_y, predictions)
print(f'感知机模型的准确率: {accuracy * 100:.2f}%')
```

结果如下:

$$Accuracy = 0.8809523809523809$$

结果总结与分析：



现在我们有完整的准确率数据：

1. Logistic Regression: 0.8571428571428572 (约为 85.71%)
2. Linear Regression: 0.6904761904761905 (约为 69.05%)
3. Perceptron: 0.8809523809523809 (约为 88.10%)

现在，让我分析这三种分类方式的准确率及其原因：

1. Logistic Regression (逻辑回归):
    - 准确率: 约为 85.71%
    - 原因: 逻辑回归是一种用于二分类问题的经典机器学习算法。它能够有效地处理线性可分和线性不可分的问题。在这种情况下，它表现出了相对较高的准确率，这可能是因为数据集较好地适合逻辑回归模型，或者数据集已经经过适当的特征工程。
2. Linear Regression (线性回归):
    - 准确率: 约为 69.05%
    - 原因: 线性回归通常用于解决回归问题，而不是分类问题。在分类问题中使用线性回归可能导致较低的准确率，因为它试图拟合一个连续的线性函数，而不是产生离散的分类结果。

3. Perceptron (感知器):
    - 准确率: 约为 88.10%
    - 原因: 感知器是一种二分类线性分类器，非常适合处理线性可分的问题。它通过不断迭代来更新权重，以找到一个可以分离数据的决策边界。在这种情况下，Perceptron 表现出了相对较高的准确率，可能因为数据集具有良好的线性可分性质。

四、

Logistic Regression代码如下：

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
%matplotlib inline
#%%
origin_data = pd.read_csv("sonar_csv.csv")
#%%
origin_data.head()
#%%
x = origin_data.iloc[:,:59]
y = origin_data.Class
y_d=np.where(y=='Rock',1,0) #将数据标签bool化，Rock为1，Mine为0
train_set_x,test_set_x,train_set_y,test_set_y=train_test_split(x,y_d,tes
t_size=0.2,random_state=42) #运train_test_split将数据划分为8：2的训练：测试集
train_set_x=train_set_x.T
test_set_x=test_set_x.T
train_set_y=train_set_y.reshape(1,166)
test_set_y=test_set_y.reshape(1,42)
#%%
# 作出数量图
categories = ['train_set', 'test_set']
values = [166, 42]

# 创建柱状图
plt.figure(figsize=(8, 6))
plt.bar(categories, values, color='skyblue')
bars = plt.bar(categories, values, color='skyblue')
for bar in bars:
    yval = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2, yval, round(yval, 1),
ha='center', va='bottom')

# 添加标题和标签
plt.title('The numbers of Train_Test_set')
plt.xlabel('Categories')
plt.ylabel('Values')
```

```
# 显示柱状图
plt.show()

#%%
train_set_x.shape,train_set_y.shape,test_set_x.shape,test_set_y.shape
#%%
def sigmoid(z):
    a = 1/(1+np.exp(-z))
    return a

def initialize_with_zeros(dim):
    w = np.zeros((dim,1))
    b = 0
    return w,b

def propagate(w, b, X, Y):
    """
    传参:
    w -- 权重, shape:  (num_px * num_px * 3, 1)
    b -- 偏置项, 一个标量
    X -- 数据集，shape:  (num_px * num_px * 3, m),m为样本数
    Y -- 真实标签，shape:  (1,m)

    返回值:
    cost,  dw ，db，后两者放在一个字典grads里
    """
    #获取样本数m:
    m = X.shape[1]

    # 前向传播 :
    A = sigmoid(np.dot(w.T,X)+b)     #调用前面写的sigmoid函数
    cost = -(np.sum(Y*np.log(A)+(1-Y)*np.log(1-A.T)))/m

    # 反向传播:
    dZ = A-Y
    dw = (np.dot(X,dZ.T))/m
    db = (np.sum(dZ))/m

    #返回值:
    grads = {"dw": dw,
             "db": db}

    return grads, cost

def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost =
False):
    #定义一个costs数组，存放每若干次迭代后的cost，从而可以画图看看cost的变化趋势:
    costs = []
    #进行迭代:
```

```
    for i in range(num_iterations):
        # 用propagate计算出每次迭代后的cost和梯度：
        grads, cost = propagate(w,b,X,Y)
        dw = grads["dw"]
        db = grads["db"]

        # 用上面得到的梯度来更新参数：
        w = w - learning_rate*dw
        b = b - learning_rate*db

        # 每100次迭代，保存一个cost看看：
        if i % 1 == 0:
            costs.append(cost)

        # 这个可以不在意，我们可以每100次把cost打印出来看看，从而随时掌握模型的进
展：
        if print_cost and i % 1 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))
    #迭代完毕，将最终的各个参数放进字典，并返回：
    params = {"w": w,
              "b": b}
    grads = {"dw": dw,
             "db": db}
    return params, grads, costs

def predict(w,b,X):
    m = X.shape[1]
    Y_prediction = np.zeros((1,m))

    A = sigmoid(np.dot(w.T,X)+b)
    for  i in range(m):
        if A[0,i]>0.5:
            Y_prediction[0,i] = 1
        else:
            Y_prediction[0,i] = 0

    return Y_prediction
#%%
def
logistic_model(X_train,Y_train,X_test,Y_test,learning_rate=0.1,num_itera
tions=2000,print_cost=False):
    #获特征维度，初始化参数：
    dim = X_train.shape[0]
    W,b = initialize_with_zeros(dim)

    #梯度下降，迭代求出模型参数：
    params,grads,costs =
optimize(W,b,X_train,Y_train,num_iterations,learning_rate,print_cost)
    W = params['w']
```

```
    b = params['b']

    #用学得的参数进行预测：
    prediction_train = predict(W,b,X_train)
    prediction_test = predict(W,b,X_test)

    #计算准确率，分别在训练集和测试集上：
    accuracy_train = 1 - np.mean(np.abs(prediction_train - Y_train))
    accuracy_test = 1 - np.mean(np.abs(prediction_test - Y_test))
    print("Accuracy on train set:",accuracy_train )
    print("Accuracy on test set:",accuracy_test )

   #为了便于分析和检查，我们把得到的所有参数、超参数都存进一个字典返回出来：
    d = {"costs": costs,
         "Y_prediction_test": prediction_test ,
         "Y_prediction_train" : prediction_train ,
         "w" : W,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations,
         "train_acy":accuracy_train,
         "test_acy":accuracy_test
        }
    return d
#%%
d=logistic_model(train_set_x,train_set_y,test_set_x,test_set_y,num_itera
tions = 1000,learning_rate= 0.08,print_cost = True)
cost=np.array(d['costs'])
cost
plt.plot(np.squeeze(d['costs']))
```