逻辑回归算法实现

学号：PB 21000224　　姓名：陈鸿绪　　日期：10.28

实验内容：

使用 python 的常用普通标准库，编写代码实现逻辑回归算法，并通过给定数据集，清洗、处理、划分数据集后，进行逻辑回归模型训练，要求预测其中一列二元属性值。

实验任务：

1. 在 Logistic.py 文件中，编写 Logistic Regression 类。
2. 对于数据集，需要进行缺失值处理，通过合理的方式扔去或者用平均值替代这些空缺值。
3. 对于种类属性（非数字的），需要用数字编码这些种类。
4. 利用合理的划分手段将原数据集划分为训练集与测试集。
5. 通过调制不同的参数进行训练，并对模型的准确率（accuracy）进行比较。

实验原理：

没有正则项的逻辑回归，损失函数表达式如下：

$$\sum_{i=1}^{m}\left(-y_i\beta^T\hat{x}_i+\ln\left(1+\exp\left(\beta^T\hat{x}_i\right)\right)\right)$$

由牛顿法求解极值有：

$$\beta^{'}=\beta-\alpha\left(\frac{\partial^2 l(\beta)}{\partial\beta\partial\beta^T}\right)^{-1}\frac{\partial l(\beta)}{\partial\beta}\quad\alpha=lr$$

对于 L1 正则项的逻辑回归，损失函数表达式如下：

$$\sum_{i=1}^{m}\left(-y_i\beta^T\hat{x}_i+\ln\left(1+\exp\left(\beta^T\hat{x}_i\right)\right)\right)+C\|\omega\|_1$$

对于 L2 正则项的逻辑回归，损失函数表达式如下：

$$\sum_{i=1}^{m}\left(-y_i\beta^T\hat{x}_i+\ln\left(1+\exp\left(\beta^T\hat{x}_i\right)\right)\right)+C\|\omega\|_2$$

实验中，对于带有 L1 正则项的采用牛顿法梯度下降，带有 L2 正则项的采用普通梯度下降。

实验步骤：

1. 加载数据集"loan.csv"得类型为 DataFrame 的数据对象 df，查看数据集头部的若干条数据。
2. 对 df 进行基本信息查看（如每一属性数据类型，占用内存，数据个数）。
3. 数据处理：首先数据清洗，在观察了每一个属性的缺失值比例情况，缺失值占比均小于 1%，所以我们采用抛弃缺失值的方法；信息编码，由于属性中一些是种类且非数字属性，所以根据不同种类进行数据编码，对于相同属性相对临近的两个属性值，我们编码时尽量将其之间距离缩小。比如对于 map_Property_Atea 属性，具有三个属性值，'Rural'与'Semiurban'相近，而'Rural'与'Urban'相差甚远，所以编码如下：

   ```
   map_Property_Area={
       'Rural': -1,
       'Semiurban': 0,
       'Urban': 1
   }
   ```

   最后对数据进行标准化，这里采用的是高斯标准化模型：

   $$X=\frac{X-\mu}{\sigma}$$

4. 数据集划分：由于整个数据集体量并非很大，所以单一的一种划分可能会使

得模型的泛化性能降低，训练集过大又会导致过拟合现象，所以实验中我们采用 k-折交叉检验方法与留出法相结合的方法，将数据集等分为 4 折，每折中正负样本比例全部相等。每取出其中 3 折作为训练集，1 折作为验证集，便得到一个模型，我们考察的是每次交叉检验后得到四个模型的平均性能。

5. 模型训练：实验中为了提高模型泛化性能，采用了对损失函数加上正则项处理，有 L1、L2 两种正则项，加上 L2 正则项的逻辑回归模型采用梯度下降的方法求解，加上 L1 正则项的模型采用牛顿法进行梯度下降求解。我们需要对步长与正则项权重等参数进行调整。
6. 模型测试：调制不同参数，比较算法预测的准确率，同时考察时间性能。

主代码解释：

```
class LogisticRegression:
    def __init__(self, df,test_df,penalty="l2", gamma=0, fit_intercept=True):
        #初始化类对象
        err_msg = "penalty must be 'l1' or 'l2', but got: {}".format(penalty)
        assert penalty in ["l2", "l1"], err_msg
        self.penalty = penalty
        self.gamma = gamma
        self.fit_intercept = fit_intercept
        self.coef_ = None
        self.df=df
        self.beta=np.array([1]*(len(df.columns)))
        self.test_df=test_df

    def loss(self) : #计算损失函数
        len_=len(self.df)
        loss=0
        for i in range(len_) :
            temp_arr=np.array(self.df.iloc[i,:-1])
            temp_arr_1=np.append(temp_arr,1)
            loss=loss-
self.df['Loan_Status'][i]*np.dot(self.beta,temp_arr_1)+np.log(1+np.exp(np.dot(self.beta,
temp_arr_1)))
```

```
        if self.fit_intercept==False :
            pass
        elif self.fit_intercept==True and self.penalty=="l1" :
            for i in range(len(self.beta-1)) :
                loss+=self.gamma*abs(self.beta[i])
        elif self.fit_intercept==True and self.penalty=="l2" :
            sum=0
            for i in range(len(self.beta-1)) :
                sum+=abs(self.beta[i])**2
            loss+=self.gamma*np.sqrt(sum)
        return loss

    def sigmoid(self, x): #计算 sigmoid 函数值
        x_=np.append(x,1)
        z=np.dot(self.beta,x_)
        if z>0 :
            return 1/(1+np.exp(-z))
        else :
            return np.exp(z)/(1+np.exp(z))

    def L_beta_2(self) : #对损失函数二阶导
        len_=len(self.df.columns)
        len_index=len(self.df)
        sum_=np.array(len_*[[0]*len_])
        for i in range(len_index) :
            temp_arr=np.array(self.df.iloc[i,:-1])
            temp_arr_1=np.append(temp_arr,1)
            sum_=sum_+self.sigmoid(temp_arr)*(1-
self.sigmoid(temp_arr))*np.dot(np.array([temp_arr_1]).T,np.array([temp_arr_1]))
        return sum_

    def L_beta_1(self) : #对损失函数的一阶导
        len_=len(self.df.columns)
        len_index=len(self.df)
        sum_=np.array(len_*[0])
        for i in range(len_index) :
            temp_arr=np.array(self.df.iloc[i,:-1])
            temp_arr_1=np.append(temp_arr,1)
            sum_=sum_-(self.df['Loan_Status'][i]-self.sigmoid(temp_arr))*temp_arr_1
        if self.fit_intercept and self.penalty=="l1":
        #考虑加入 L1 正则项
            for i in range(len(sum_)-1) :
                if self.beta[i]<0 :
                    sum_[i]-=self.gamma
```

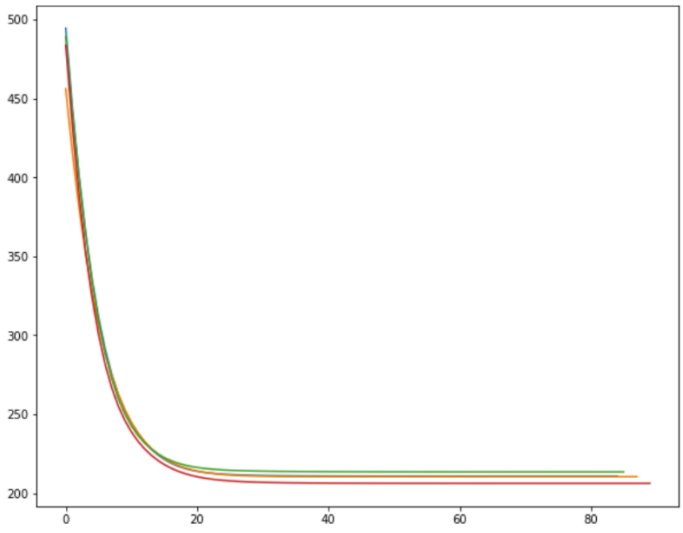
```
                else :
                    sum_[i]+=self.gamma

        elif self.fit_intercept and self.penalty=="l2":
        #考虑加入 L2 正则项
            temp=np.sqrt(np.dot((self.beta)[:-1],(self.beta)[:-1]))
            for i in range(len(sum_)-1) :
                sum_[i]+=self.gamma*(self.beta)[i]/temp
        return sum_

    def fit(self, lr=0.01, tol=1e-7, max_iter=1e7):
        self.lr=lr
        self.tol=tol
        self.max_iter=max_iter
        beta_new=np.array([0]*(len(self.df.columns)))
        time=0
        loss_list=[]
        if self.penalty =='l2' :  #L2 正则项采用牛顿法梯度下降
            while True :
                gradient=self.L_beta_1()
                beta_new=self.beta-self.lr*gradient
                if np.dot(gradient,gradient)<self.tol :
                    break
                self.beta=beta_new.copy()
                time+=1
                if time>=self.max_iter :
                    break
                loss_list.append(self.loss())
            return loss_list
        else :  #L1 正则项采用普通梯度下降
            while True :
                gradient=self.L_beta_1()
                beta_new=([self.beta]-
self.lr*np.dot(np.array([gradient]),np.linalg.inv(np.array(self.L_beta_2().T)))[0])[0]
                if np.dot(gradient,gradient)<self.tol :
                    break
                self.beta=beta_new.copy()
                if np.dot(gradient,gradient)<0.003 :
                    self.lr=0.01
                time+=1
                if time>=self.max_iter :
                    break
                loss_list.append(self.loss())
            return loss_list
```

```
def predict(self) : #得到预测的值
    len_=len(self.test_df)
    predict=[]
    for i in range(len_) :
        temp_arr=np.array(self.test_df.iloc[i,:-1])
        flag=self.sigmoid(temp_arr)
        if flag>=0.5 :
            if (self.test_df)['Loan_Status'][i] == 1 :
                predict.append([1,1])
            else :
                predict.append([1,0])
        else :
            if (self.test_df)['Loan_Status'][i] == 1 :
                predict.append([0,1])
            else :
                predict.append([0,0])
    self.predict=predict
    return predict

def get_accurate(self) : #预测结果的 accuracy
    accurate=0
    for i in range(len(self.predict)) :
        if (self.predict)[i][0] == (self.predict)[i][1] :
            accurate+=1
    return accurate/len(self.predict)

def get_R_score(self) : #预测结果的 R score
    TP,FP,FN=0,0,0
    for i in range(len(self.predict)) :
        if (self.predict)[i] == [1,1] :
            TP+=1
        elif (self.predict)[i] == [1,0] :
            FP+=1
        elif (self.predict)[i] == [0,1] :
            FN+=1
    self.R_score=TP/(TP+FN)
    return self.R_score

def get_P_score(self) : #预测结果的 P score
    TP,FP,FN=0,0,0
    for i in range(len(self.predict)) :
        if (self.predict)[i] == [1,1] :
            TP+=1
```

```
            elif (self.predict)[i] == [1,0] :
                FP+=1
            elif (self.predict)[i] == [0,1] :
                FN+=1
        self.P_score=TP/(TP+FP)
        return self.P_score

    def f1_score(self) : #预测结果的 p score
        R=self.get_R_score()
        P=self.get_P_score()
        self.f1_score=2*P*R/(P+R)
        return self.f1_score
```

实验结果与分析：

1. 对 k 折交叉验证有模型训练时损失函数的图像（4 折，所以有 4 条曲线）：



上图中横坐标为迭代次数，纵坐标为损失函数大小。对于 4 条不同颜色的曲线可以发现，最后 4 折得到的 4 个模型对应的损失函数值均不相同，每一个模型到收敛所用迭代次数也各不相同，损失函数曲线特征为一开始剧烈下降，接着趋于平缓，即梯度慢慢接近于 0 向量。最后将这 4 个模型所得的评价指标取平均得到对于该模型的最终评价结果。

2. 不同参数之间模型比较：

Tol=1e-3、lr=0.3 时 L1 正则项的加权系数与准确率的关系

加权大小	0	0.05	0.08	0.09
Accuracy	0.80417	0.80417	0.80417	0.80417
P score	0.79336	0.79336	0.79336	0.79336
R score	0.96988	0.96988	0.96988	0.96988
F1 score	0.87264	0.87264	0.87264	0.87264

以上随机种子固定为 168。对 L1 正则项的加权系数，我们只求出了加权系数在 0-0.09 范围的各项指标，在实验中发现若加权大于 0.1，lr 参数调小会导致程序时间代价大幅提升，lr 参数调大会导致该算法不收敛。所以如果希望进一步扩大 L1 正则项系数范围，固定学习率的梯度下降算法在本台实验电脑上已经行不通，此时学习率需要是一个变量，随着梯度逐渐变成 0 而变小，但本次实验并没有实现该想法。由于 L1 正则项系数变化范围过小，所以预测结果不会有改变（实际上向量 beta 的值是具有小变化的，但是不足以影响预测结果，这点可以从代码运行结果看出）。

Tol=1e-5、lr=0.002 时 L2 正则项的加权系数与准确率的关系

加权大小	1	5	15	30
Accuracy	0.80625	0.81042	0.81042	0.80833
P score	0.79385	0.79489	0.79489	0.79307
R score	0.97289	0.97892	0.97892	0.97892
F1 score	0.87419	0.87724	0.87724	0.87610

40	45	50	60
0.80833	0.80833	0.80625	0.76667
0.79307	0.79307	0.79105	0.75494
0.97892	0.97892	0.97892	0.98193
0.87610	0.87610	0.87490	0.85347

以上随机种子固定为 168。可以发现 L2 加权系数在 1-60 之间变化时，准确率成先增后减的趋势，大概在加权系数在 5—15 范围左右就会达到准确率最大。该准确率最大为 0.81042。这是因为加入 L2 正则项后，如果 L2 正则项系数适中，则模型泛化性能将达到较优的状态。

对于步长（学习率）不同值，在逻辑回归算法收敛的情况下只会影响到时间代价，而对预测结果没有任何影响（因为是凸问题）。但是如果步长较大则会使得损失函数无法有效下降而陷入死循环。所以本算法中的学习率参数都是经过多次调整比较得到的一个较优且尽可能避免死循环的值。

3. 最优情况：在实验中，实际操作中选取了多个随机种子进行了多次参数调整，最后在本代码的框架下得到采用 L2 正则项、随机种子为 168，正则系数在 5—15 范围、学习率在 0.02、精度在 1e-5 以内时刻得到较优的预测准确率，其大小为 0.81042。

   平均情况：在多次试验后发现，在 4-折交叉验证下，每次得到的平均准确率的方差并不大，准确率稳定在 0.80 左右。