Assignment-3 Linear Regression & Logistic Regression

21307289 刘森元

Exercise-1 Linear Regression

Subtask a)

参数的数量需要根据数据的特征数来决定，由 *dataForTrainingLinear.txt* 为例，参数为六个。

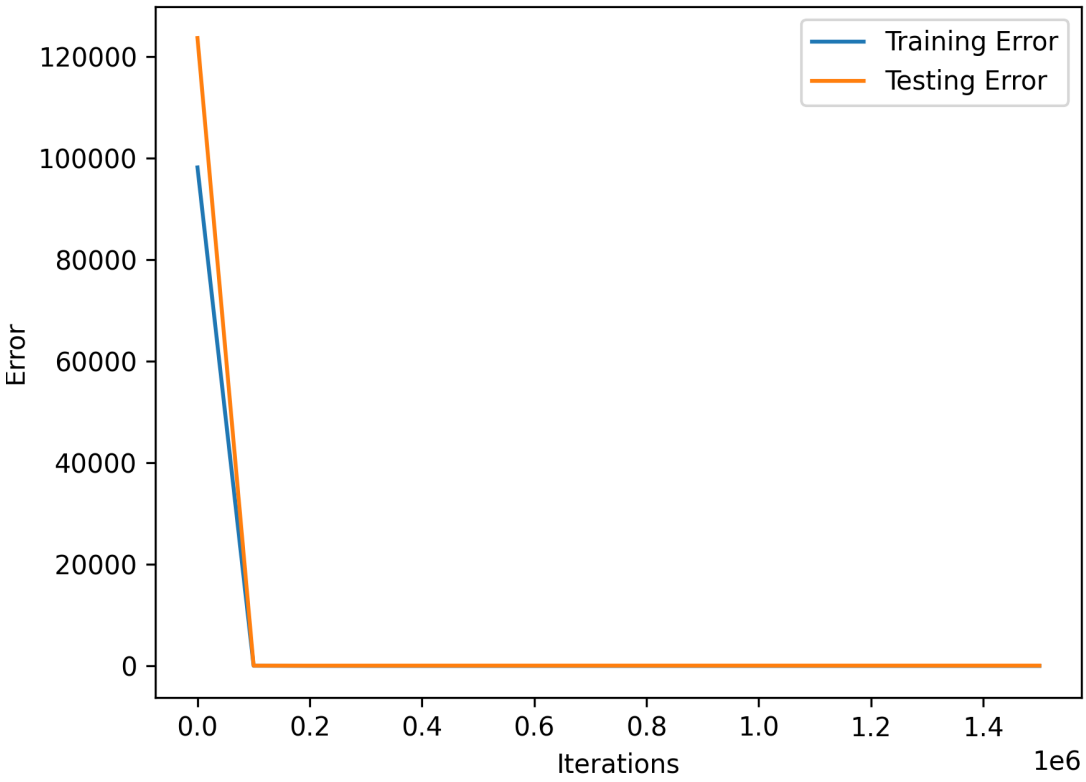
$$\theta_j := \theta_j - \frac{\alpha}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

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

根据线性回归模型的特征，可以写出以下代码：

```
def gradient_descent(X, y, parameters, learning_rate, num_iterations):
    m = len(y)
    for i in range(num_iterations):
        predictions = np.dot(X, parameters[1:]) + parameters[0]
        parameters[1:] -= (learning_rate / m) * np.dot(X.T, (predictions - y))
        parameters[0] -= (learning_rate / m) * np.sum(predictions - y)
    return parameters
```

完整代码详见附件 Solution-1a.py，训练后有如下误差结果：



分析有：

- 训练误差的下降：随着迭代次数的增加，训练误差逐渐减小。这是因为模型通过梯度下降算法不断调整参数，使得预测值与实际值之间的误差逐渐减小，模型能够更好地拟合训练数据。

但是这仅为理想状况下的情况，在应用层面，更多时候会出现

- 测试误差的变化：随着迭代次数的增加，测试误差可能会先减小后增大，形成一个 U 形曲线。这是因为模型在开始阶段过度拟合了训练数据，导致在测试数据上的表现并不好，随着迭代次数的增加，模型开始逐渐泛化，测试误差减小，但在一定程度后，模型可能开始过度泛化，导致测试误差增大。
- 过拟合和欠拟合：通过观察训练误差和测试误差的曲线，可以判断模型的拟合情况。如果训练误差和测试误差都很高，说明模型欠拟合，无法很好地拟合训练数据和测试数据。如果训练误差很低而测试误差很高，说明模型过拟合，过度拟合了训练数据，无法很好地泛化到测试数据。
- 模型的收敛速度：通过观察曲线的斜率，可以判断模型的收敛速度。如果曲线的斜率在开始阶段很陡峭，然后逐渐变缓，说明模型在开始阶段快速收敛，然后逐渐趋于稳定。如果曲线的斜率一直保持较大，说明模型收敛速度较慢。

Subtask b)

在 Subtask a) 的基础上，将学习率更改为 0.0002 时，出现以下状况：

```
Solution-1a.py:41: RuntimeWarning: invalid value encountered in subtract
  parameters[1:] -= (learning_rate / m) * np.dot(X.T, (predictions - y))
```

Debug 可知，这是因为溢出导致的，即模型出现了梯度爆炸的状况。

出现溢出导致梯度爆炸的状况可能是由于学习率过大导致的。当学习率过大时，梯度下降算法在每次参数更新时会产生较大的变化，这可能导致参数值变得非常大，超过了计算机可以表示的范围，从而导致溢出。

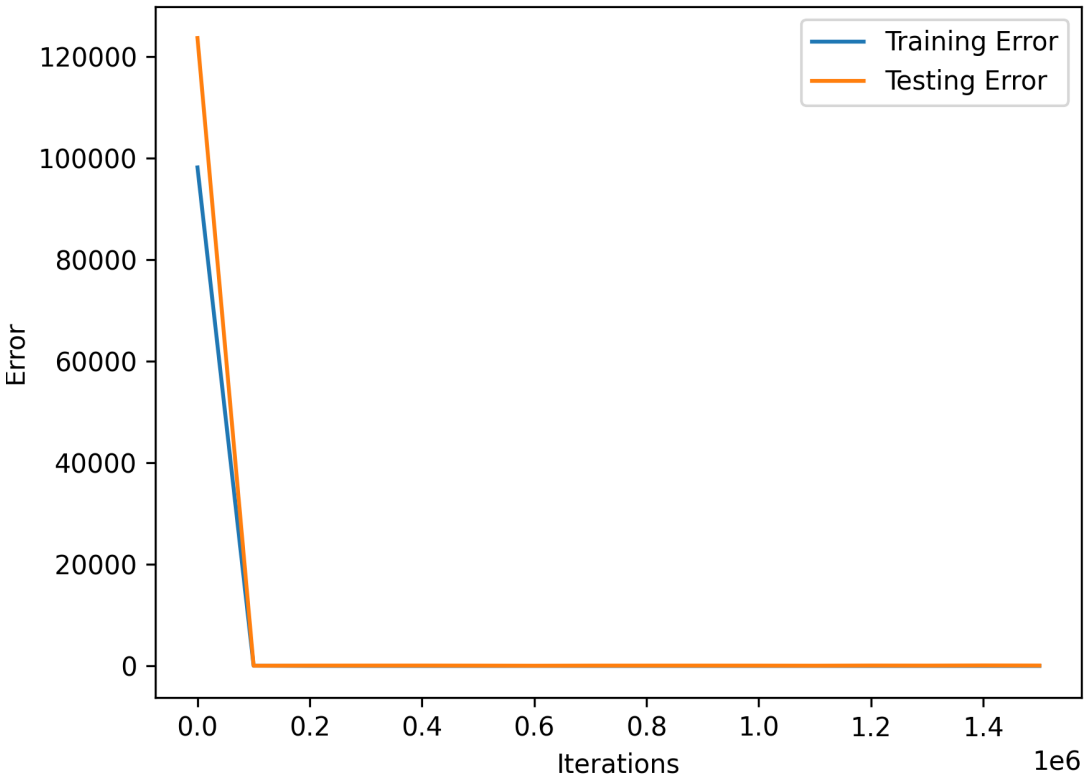
为了解决这个问题，可以尝试以下方法：

- 缩放数据：对数据进行归一化处理，将特征值缩放到一个较小的范围内，例如[0, 1]或[-1, 1]。这可以帮助避免梯度爆炸问题。
- 调整学习率：减小学习率的值，使其更接近于0。较小的学习率可以减缓参数的变化速度，从而减少梯度爆炸的风险。
- 使用其他优化算法：梯度下降算法可能对于某些数据集和学习率组合来说不稳定。尝试使用其他优化算法，如Adam、RMSProp等，这些算法可以自适应地调整学习率，从而更好地处理梯度爆炸问题。

Subtask c)

简单修改 Subtask a) 中的代码可得到：

```python
# 定义随机梯度下降函数
def stochastic_gradient_descent(X, y, parameters, learning_rate,
                                num_iterations, batch_size):
    m = len(y)
    for i in range(num_iterations):
        # 随机选择一批数据
        batch_indices = np.random.choice(m, batch_size, replace=False)
        X_batch = X[batch_indices]
        y_batch = y[batch_indices]
        # 计算预测值和误差
        predictions = np.dot(X_batch, parameters[1:]) + parameters[0]
        errors = predictions - y_batch
        # 更新参数
        parameters[1:] -= (learning_rate / batch_size) * np.dot(
            X_batch.T, errors)
        parameters[0] -= (learning_rate / batch_size) * np.sum(errors)
    return parameters
```



可见使用随机梯度下降法仍然能够获得最优的参数，对比两种方法，能够发现：

梯度下降法：

1. 收敛性好：在合适的学习率下，梯度下降法通常能够较快地收敛到局部最优解。
2. 全局最优解：对于凸函数，梯度下降法能够找到全局最优解。
3. 简单易理解：梯度下降法的原理相对简单，易于理解和实现。
4. 需要遍历所有样本：梯度下降法在每次迭代时需要遍历所有的训练样本，这在大规模数据集上可能会导致计算时间较长。
5. 可能陷入局部最优解：对于非凸函数，梯度下降法容易陷入局部最优解，而无法达到全局最优解。

随机梯度下降法：

1. 计算效率高：随机梯度下降法每次迭代只使用一个样本或一小批样本进行参数更新，因此计算效率较高。
2. 可以适用于大规模数据集：由于每次迭代只使用少量样本，随机梯度下降法适用于大规模数据集，可以更快地进行参数更新。
3. 可以逃离局部最优解：由于每次迭代使用随机样本，随机梯度下降法有机会跳出局部最优解，寻找更优的全局最优解。

4. 不稳定性：由于每次迭代使用随机样本，随机梯度下降法的更新方向可能会有较大的波动，导致参数的收敛路径不稳定。
5. 学习率调整困难：由于样本的随机性，难以确定合适的学习率，可能需要手动调整学习率以达到更好的收敛效果。
6. 可能无法收敛到最优解：由于随机性的影响，随机梯度下降法可能无法达到全局最优解，而只能接近最优解。

综上所述，梯度下降法在稳定性和收敛性方面具有优势，适用于小规模数据集和凸函数优化。而随机梯度下降法在计算效率和适用于大规模数据集方面具有优势，但可能会牺牲一定的稳定性和全局最优解的能力。选择使用哪种优化算法应根据具体的问题和数据集特点进行综合考虑。

Exercise-2 Logistic Regression

Subtask a)

对于一个包含 n 个训练示例和 p 个特征的数据集，我们可以使用以下公式表示训练数据的条件对数似然：

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{n} \left[y^{(i)} \log \left(\frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}^{(i)}}} \right) + (1 - y^{(i)}) \log \left(1 - \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}^{(i)}}} \right) \right]$$

其中，$\mathbf{w} = [w_0, w_1, \ldots, w_p]$ 是参数向量，$\mathbf{x}^{(i)} = [1, x_1^{(i)}, \ldots, x_p^{(i)}]$ 是第 i 个训练示例的特征向量，$y^{(i)}$ 是第 i 个训练示例的类别标签。

这个公式表示了在给定参数 $\mathbf{w}$ 的情况下，观察到训练数据的概率。我们的目标是最大化这个概率，即最大化对数似然函数 $\mathcal{L}(\mathbf{w})$。

Subtask b)

要计算目标函数关于 w_0 和任意 w_j 的偏导数，我们需要对条件对数似然函数进行求导。

首先，我们计算目标函数关于 w_0 的偏导数 $\frac{\partial f}{\partial w_0}$：

$$\frac{\partial f}{\partial w_0} = \sum_{i=1}^{n} \left(y^{(i)} - \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}^{(i)}}} \right) x_0^{(i)}$$

其中，$x_0^{(i)}$ 是第 i 个训练示例的第一个特征，即常数 1。

接下来，我们计算目标函数关于任意 w_j 的偏导数 $\frac{\partial f}{\partial w_j}$，其中 $j \in \{1, 2, \ldots, p\}$：

$$\frac{\partial f}{\partial w_j} = \sum_{i=1}^{n} \left(y^{(i)} - \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}^{(i)}}} \right) x_j^{(i)}$$

这两个偏导数都可以写成有限和的形式，其中每个训练示例的误差项乘以相应的特征值。这样，我们可以使用梯度上升算法来最大化条件对数似然函数。

Subtask c)

设计和训练逻辑回归分类器的一般步骤如下：

1. 数据准备：收集和准备训练数据集。确保数据集包含特征矩阵和相应的标签向量。
2. 特征工程：根据问题的特定要求，对特征进行选择、提取或转换。这可能包括特征缩放、特征选择、特征组合等操作。
3. 参数初始化：初始化逻辑回归模型的参数向量。
4. 定义模型：确定逻辑回归模型的假设函数，通常使用sigmoid函数将线性回归模型的输出映射到概率。
5. 定义损失函数：选择适当的损失函数来度量模型预测与实际标签之间的差异。在逻辑回归中，通常使用对数似然损失函数。
6. 梯度上升算法：使用梯度上升算法或其他优化算法最大化对数似然函数。通过迭代更新参数来最大化似然函数，直到达到收敛条件。
7. 模型评估：使用验证集或交叉验证来评估模型的性能。常见的评估指标包括准确率、精确率、召回率、F1分数等。
8. 超参数调优：根据模型性能选择最佳的超参数，如学习率、迭代次数等。
9. 模型应用：使用训练好的逻辑回归模型对新的未标记样本进行预测。

现在可以写出逻辑回归模型

```
# 定义逻辑回归模型
class LogisticRegression:

    def __init__(self):
        self.parameters = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def compute_cost(self, X, y, parameters):
        m = len(y)
        h = self.sigmoid(np.dot(X, parameters))
        cost = -(1 / m) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))
        return cost

    def gradient_ascent(self, X, y, learning_rate, num_iterations):
        m, n = X.shape
        self.parameters = np.zeros(n)

        for it in trange(num_iterations):
            h = self.sigmoid(np.dot(X, self.parameters))
```

```
            error = h - y
            gradient = np.dot(X.T, error)
            self.parameters -= learning_rate * gradient

    def predict(self, X):
        h = self.sigmoid(np.dot(X, self.parameters))
        predictions = np.round(h)
        return predictions
```

训练后有结果

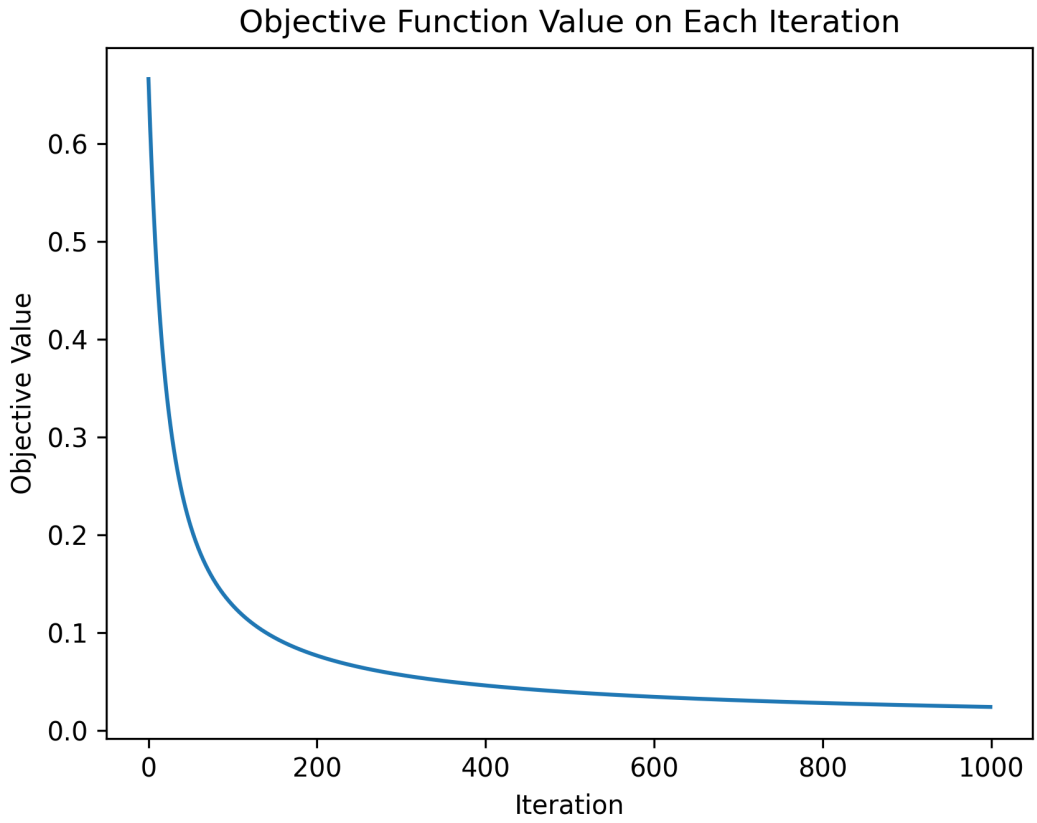
```
Parameters:
[ 0.80131003 -3.75763946  5.12608959 -3.7949055   4.92020543 -3.05452776
  0.10739291]
Errors:    0.0
Err_Rates: 0.0
```

Subtask d)

如 Subtask c) 中所示，错误分类数量为 0

Subtask e)

绘制出图像有 *(代码详见 Solution-2e.py)*



可见模型在 400 次迭代后趋于收敛

Subtask f)

进行如下训练

```
# 评估训练集大小增加时的训练误差和测试误差
train_errors = []
test_errors = []
training_sizes = np.linspace(10, len(train_data), 10, dtype=np.int32)

for k in training_sizes:
    # 随机选择训练集
    indices = np.random.choice(len(X_train), k, replace=False)
    X_train_subset = X_train[indices]
    y_train_subset = y_train[indices]

    # 训练模型
    logreg = LogisticRegression()
    learning_rate = 0.001
    num_iterations = 1000
    logreg.gradient_ascent(X_train_subset, y_train_subset, learning_rate,
```

```
                            num_iterations)

    # 计算训练误差和测试误差
    train_error = calculate_error(X_train_subset, y_train_subset,
                                  logreg.parameters)
    test_error = calculate_error(X_test, y_test, logreg.parameters)

    train_errors.append(train_error)
    test_errors.append(test_error)
```

有如下结果

随着训练集大小的增加，训练误差逐渐减少，而测试误差先减少后增加。

这种行为发生的原因是，当训练集较小时，模型可能无法捕捉到数据的所有模式和特征，导致欠拟合。

随着训练集大小的增加，模型能够更好地学习数据的特征，从而减少了训练误差。

当训练集过大时，模型可能过度拟合训练数据，导致在未见过的测试数据上表现较差，从而增加了测试误差。