

作业1：线性回归与梯度下降

任务：用 Python 实现线性回归模型，并手动编写梯度下降算法。

要求：

1. 生成带噪声的线性数据（例如： $y = 2x + 1 + \text{噪声}$ ）。
2. 用 NumPy 实现梯度下降优化权重和偏置。
3. 绘制损失函数随迭代次数下降的曲线。
4. 比较不同学习率（如 0.01、0.1、0.5）对收敛速度的影响。

1. 导入需要用到的库

```
In [1]: import numpy as np
import matplotlib.pyplot as plt

# 显示中文
plt.rcParams['font.sans-serif'] = ['Microsoft YaHei', 'SimHei', 'DejaVu Sans']
plt.rcParams['axes.unicode_minus'] = False

np.random.seed(42)
```

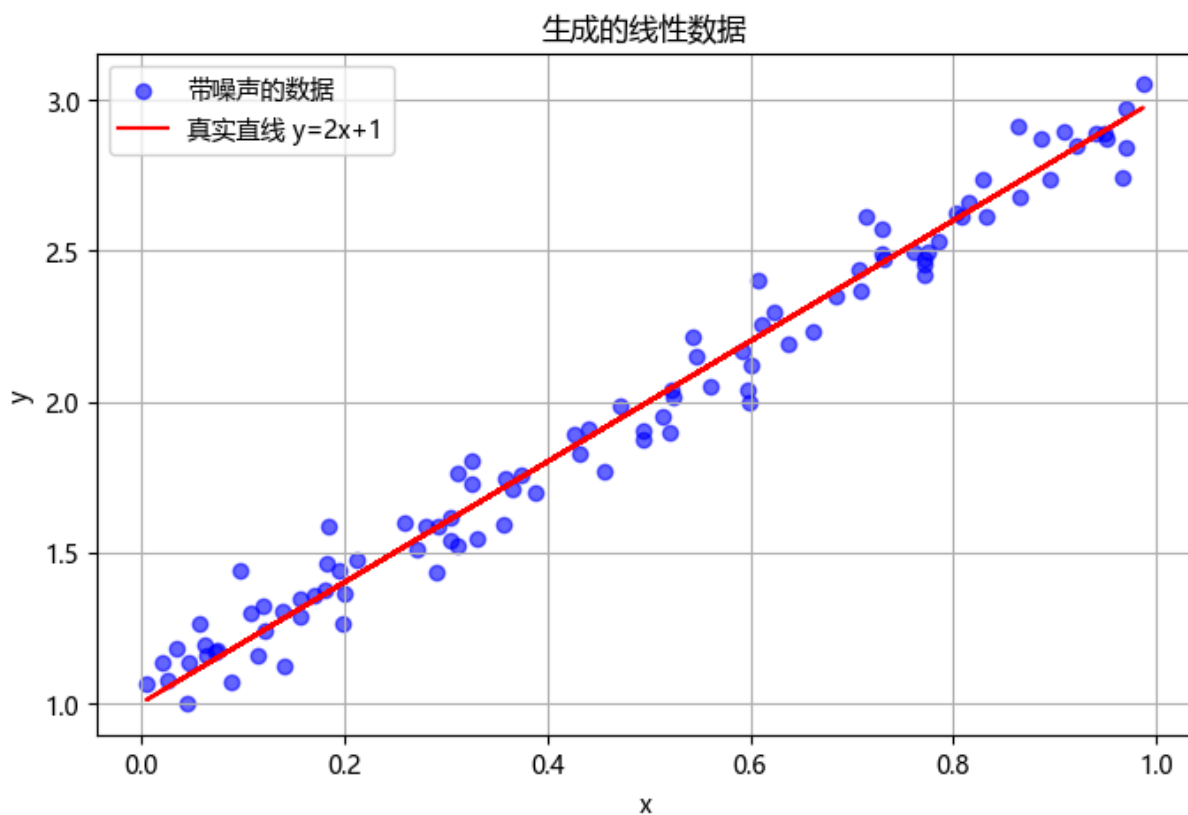
2. 生成带噪声的线性数据

生成的数据满足公式： $y = 2x + 1 + \varepsilon$ ，其中 ε 是噪声。

```
In [2]: # 真实参数
true_w = 2.0
true_b = 1.0

# 生成数据: x 取 0~1, 方便不同学习率都能收敛
n = 100
x = np.random.uniform(0, 1, n)
noise = np.random.randn(n) * 0.1
y = true_w * x + true_b + noise

# 画出来看看
plt.figure(figsize=(8, 5))
plt.scatter(x, y, color='blue', alpha=0.6, label='带噪声的数据')
plt.plot(x, true_w * x + true_b, color='red', label='真实直线 y=2x+1')
plt.xlabel('x')
plt.ylabel('y')
plt.title('生成的线性数据')
plt.legend()
plt.grid(True)
plt.show()
```



3. 手动实现梯度下降

```
In [3]: def gradient_descent(x, y, lr, n_iter):  
    w, b = 0.0, 0.0  
    loss_list = []  
    n = len(x)  
  
    for i in range(n_iter):  
        y_pred = w * x + b  
        error = y_pred - y  
        loss = np.mean(error ** 2)  
        loss_list.append(loss)  
  
        # 梯度  
        dw = (2 / n) * np.sum(error * x)  
        db = (2 / n) * np.sum(error)  
  
        # 更新参数  
        w -= lr * dw  
        b -= lr * db  
  
    return w, b, loss_list
```

4. 用学习率 0.1 训练一次看看效果

```
In [4]: lr = 0.1  
        n_iter = 1000
```

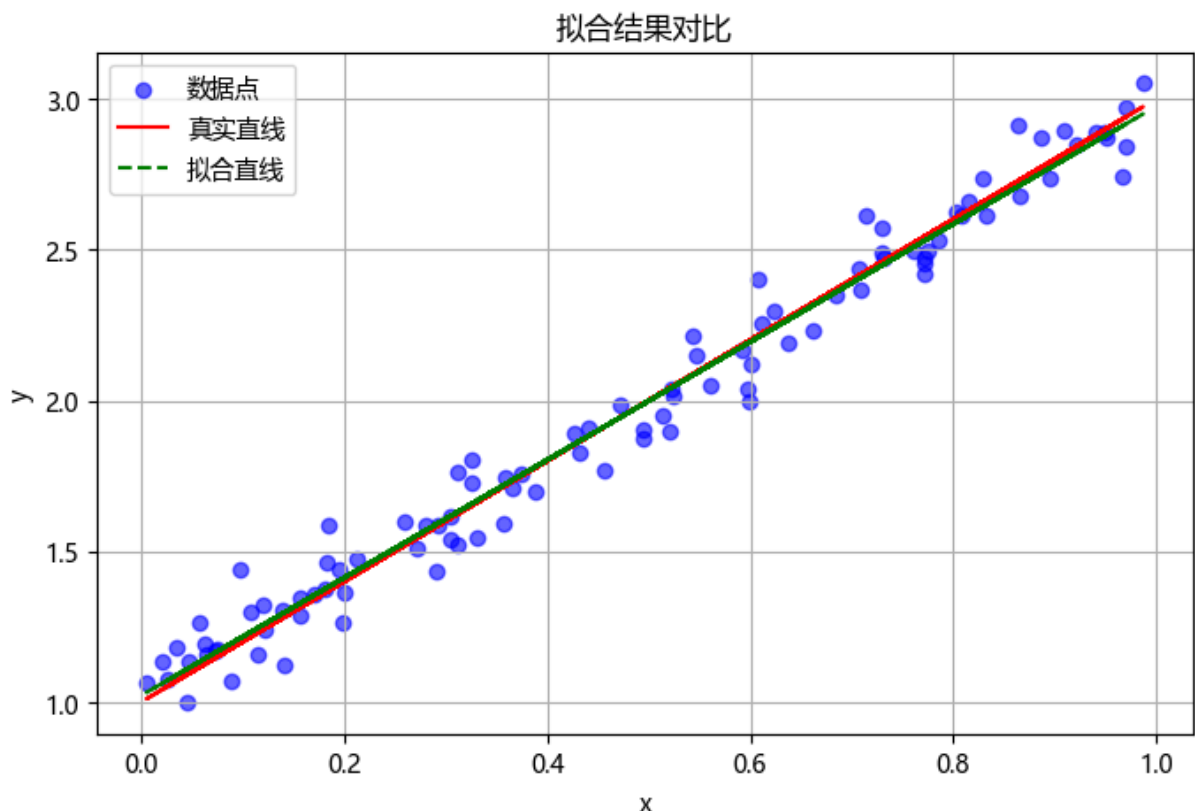
```
w, b, loss_list = gradient_descent(x, y, lr, n_iter)

print('真实的 w =', true_w, ', 真实的 b =', true_b)
print('学到的 w =', round(w, 4), ', 学到的 b =', round(b, 4))
print('最终的损失值 =', round(loss_list[-1], 6))
```

真实的 $w = 2.0$, 真实的 $b = 1.0$
 学到的 $w = 1.954$, 学到的 $b = 1.0215$
 最终的损失值 = 0.008066

5. 把拟合的直线和数据画在一起看看

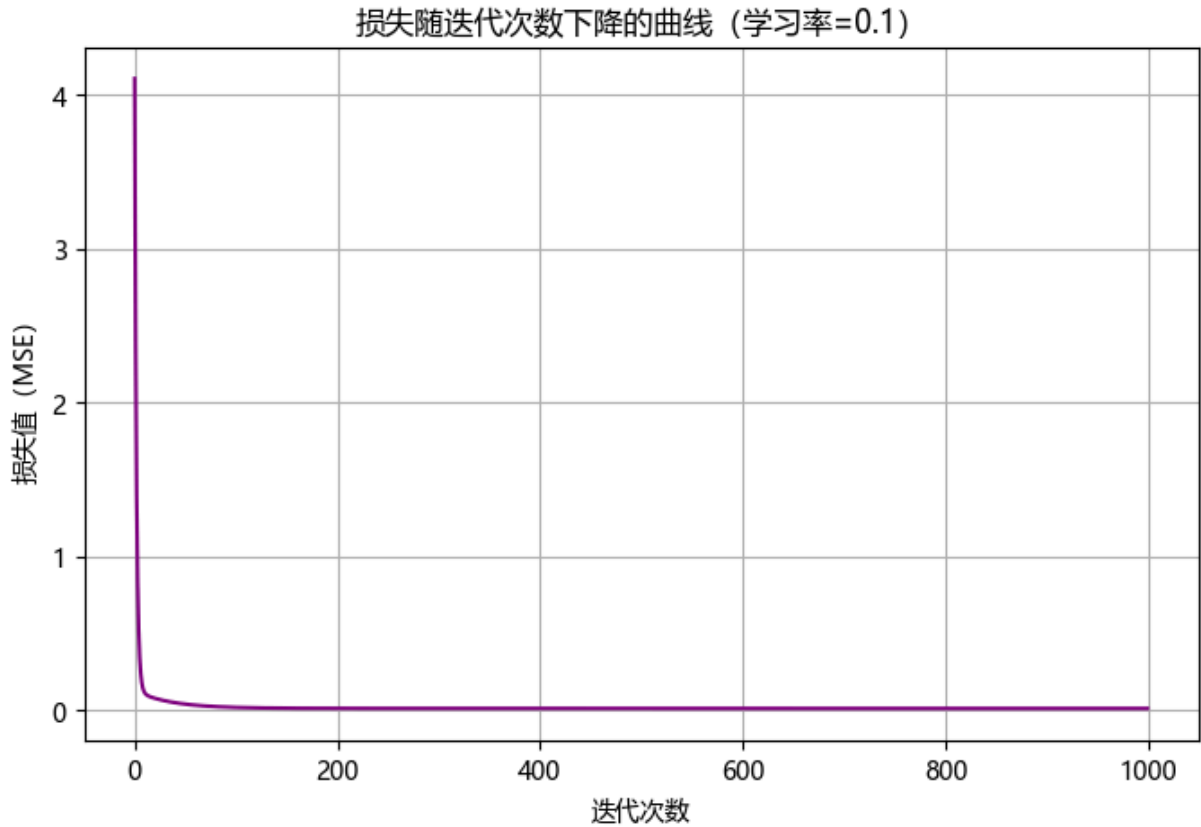
```
In [5]: plt.figure(figsize=(8, 5))
plt.scatter(x, y, color='blue', alpha=0.6, label='数据点')
plt.plot(x, true_w * x + true_b, color='red', label='真实直线')
plt.plot(x, w * x + b, color='green', linestyle='--', label='拟合直线')
plt.xlabel('x')
plt.ylabel('y')
plt.title('拟合结果对比')
plt.legend()
plt.grid(True)
plt.show()
```



6. 绘制损失函数下降的曲线

```
In [6]: plt.figure(figsize=(8, 5))
plt.plot(range(n_iter), loss_list, color='purple')
```

```
plt.xlabel('迭代次数')
plt.ylabel('损失值 (MSE)')
plt.title('损失随迭代次数下降的曲线 (学习率=0.1)')
plt.grid(True)
plt.show()
```



7. 比较不同学习率对收敛速度的影响

我们试一下三个学习率：0.01、0.1、0.5，看看哪个收敛得快。

```
In [7]: lr_list = [0.01, 0.1, 0.5]
n_iter = 1000

results = {}
for lr in lr_list:
    w, b, loss_list = gradient_descent(x, y, lr, n_iter)
    results[lr] = loss_list
    print('学习率 =', lr, ', 学到的 w =', round(w, 4),
          ', 学到的 b =', round(b, 4),
          ', 最终损失 =', round(loss_list[-1], 6))
```

学习率 = 0.01 , 学到的 $w = 1.6765$, 学到的 $b = 1.1619$, 最终损失 = 0.014932

学习率 = 0.1 , 学到的 $w = 1.954$, 学到的 $b = 1.0215$, 最终损失 = 0.008066

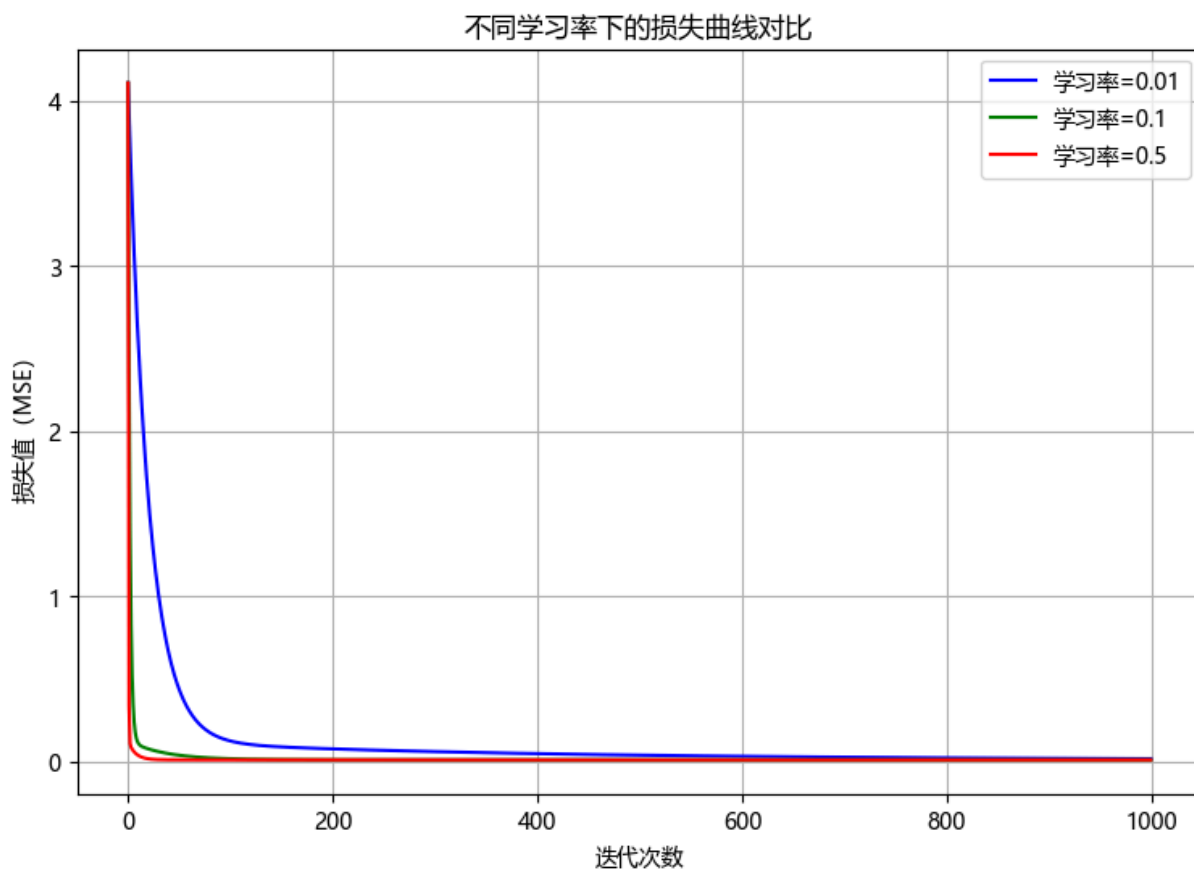
学习率 = 0.5 , 学到的 $w = 1.954$, 学到的 $b = 1.0215$, 最终损失 = 0.008066

```
In [8]: plt.figure(figsize=(9, 6))
colors = ['blue', 'green', 'red']

for i, lr in enumerate(lr_list):
```

```
plt.plot(range(n_iter), results[lr], color=colors[i], label='学习率=' + str(lr))

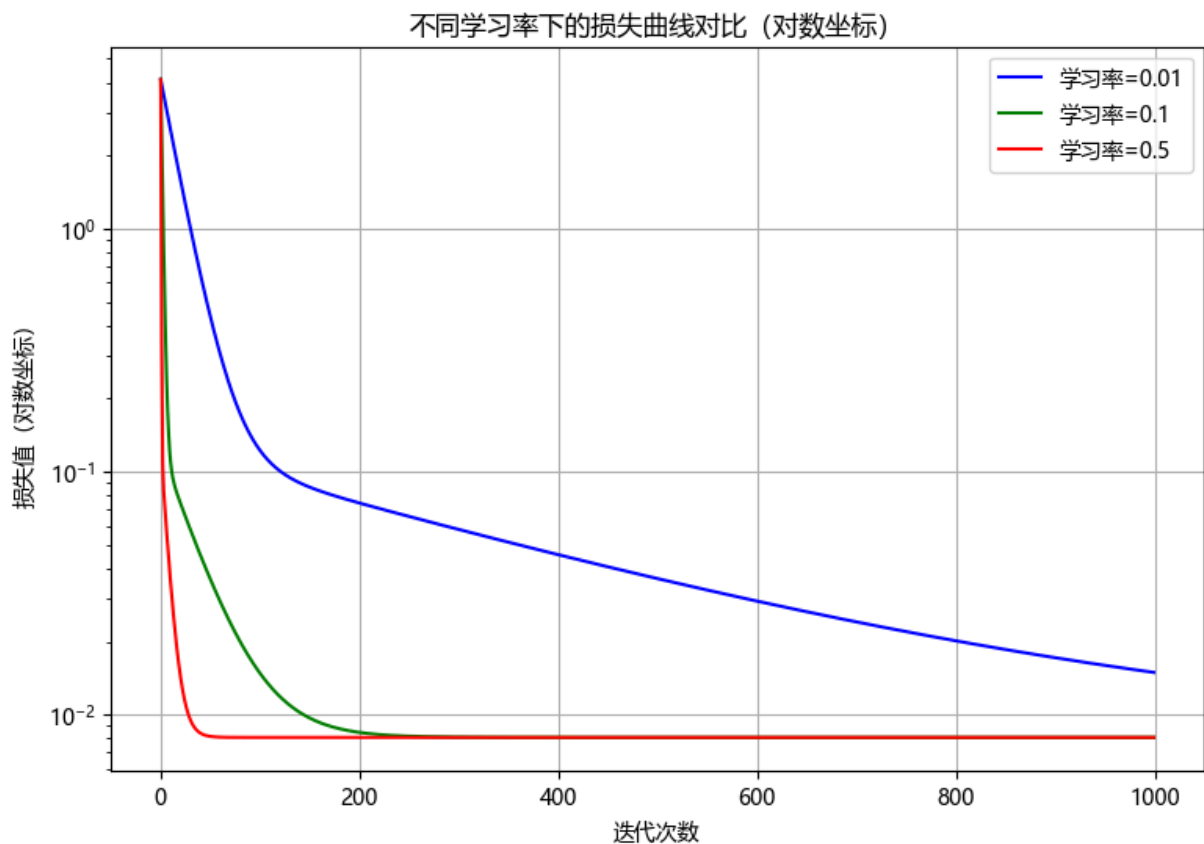
plt.xlabel('迭代次数')
plt.ylabel('损失值 (MSE) ')
plt.title('不同学习率下的损失曲线对比')
plt.legend()
plt.grid(True)
plt.show()
```



```
In [9]: # 用对数坐标再画一次，三条曲线的差异看得更清楚
plt.figure(figsize=(9, 6))

for i, lr in enumerate(lr_list):
    plt.plot(range(n_iter), results[lr], color=colors[i], label='学习率=' + str(lr))

plt.yscale('log')
plt.xlabel('迭代次数')
plt.ylabel('损失值 (对数坐标) ')
plt.title('不同学习率下的损失曲线对比 (对数坐标) ')
plt.legend()
plt.grid(True)
plt.show()
```



8. 结论

通过上面的实验可以看到：

- **学习率 = 0.01**：收敛比较慢，迭代 1000 次后 w 还没完全到 2.0，损失也比另外两个大很多。
- **学习率 = 0.1**：收敛速度合适，几百次迭代就能让损失降到接近最小值。
- **学习率 = 0.5**：收敛最快，损失曲线一开始就飞快下降，最后和 0.1 收敛到同一个值。

总结：

- 学习率太小，收敛速度慢，需要更多迭代次数。
- 学习率太大可能会让训练震荡甚至发散（如果把上面的 x 范围改成 $0 \sim 10$ ，学习率 0.1 和 0.5 就会直接发散）。
- 实际中要根据数据情况选一个合适的学习率。