机器学习 HW1: 线性模型和支持向量机

姓名：聂礼昂
学号：2022012097

2024 年 10 月 27 日

目录

1 引言

本次作业通过实现线性模型与梯度下降，Softmax 回归，支持向量机来加深对机器学习基本知识的了解。

2 线性模型与梯度下降

在本题中，使用梯度下降法（Gradient Descent）实现岭回归（Ridge Regression）算法。

2.1 特征归一化

2.1.1 split_data

补全函数 split_data，将数据集划分为训练集和测试集我们可以根据 split_size 参数来计算各个部分所需的样本数，然后将数据集划分为对应的子集。

```
def split_data(X, y, split_size=[0.8, 0.2], shuffle=False, random_seed=
    ↪ None):
    """
    对数据集进行划分

    Args:
        X - 特征向量，一个大小为 (num_instances, num_features) 的二维 numpy
            ↪  数组
        y - 标签向量，一个大小为 (num_instances) 的一维 numpy 数组
        split_size - 划分比例，期望为一个浮点数列表，如[0.8, 0.2]表示将数据
            ↪ 集划分为两部分，比例为80%和20%
        shuffle - 是否打乱数据集
        random_seed - 随机种子

    Return:
        X_list - 划分后的特征向量列表
        y_list - 划分后的标签向量列表
    """
    assert sum(split_size) == 1
    num_instances = X.shape[0]
    if shuffle:
        rng = np.random.RandomState(random_seed)
        indices = rng.permutation(num_instances)
        X = X[indices]
        y = y[indices]

    # TODO 2.1.1
```

```
    # 计算训练集的大小
    train_size = int(num_instances * split_size[0])

    # 分割数据集
    X_train, X_test = X[:train_size], X[train_size:]
    y_train, y_test = y[:train_size], y[train_size:]

    # 返回分割后的数据
    X_list = [X_train, X_test]
    y_list = [y_train, y_test]

    return X_list, y_list
```

2.1.2 feature_normalization

补全函数 feature_normalization，实现特征的归一化。我们需要将训练集 train 的所有特征值归一化到 [0, 1] 范围内，并对测试集 test 使用相同的归一化变换。通常，归一化是通过对每个特征应用以下公式来完成的：

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

为了避免除以零的情况（当某一列的 train_max 和 train_min 相等时，特征不变），我们将差值 diff 中的 0 替换为 1。

```
def feature_normalization(train, test):
    """将训练集中的所有特征值映射至[0,1]，对测试集上的每个特征也需要使用相
       ↪ 同的仿射变换

    Args:
        train - 训练集，一个大小为 (num_instances, num_features) 的二维
           ↪ numpy 数组
        test - 测试集，一个大小为 (num_instances, num_features) 的二维
           ↪ numpy 数组
    Return:
        train_normalized - 特征归一化后的训练集
        test_normalized - 特征归一化后的测试集

    """
    # TODO 2.1.2

    # 计算训练集中的最小值和最大值
    train_min = np.min(train, axis=0)  # 对每个特征的最小值
    train_max = np.max(train, axis=0)  # 对每个特征的最大值
```

```
    # 防止除以零，避免最大值等于最小值的情况
    diff = train_max - train_min
    diff[diff == 0] = 1  # 避免出现除以 0 的情况

    # 归一化训练集
    train_normalized = (train - train_min) / diff

    # 使用训练集的最小值和最大值来归一化测试集
    test_normalized = (test - train_min) / diff

    return train_normalized, test_normalized
```

2.2 目标函数与梯度

2.2.1 $J(\theta)$ 的矩阵形式

$$J(\theta) = \frac{1}{m}(X\theta - y)^T(X\theta - y) + \lambda\theta^T\theta$$

2.2.2 compute_regularized_square_loss

```
def compute_regularized_square_loss(X, y, theta, lambda_reg):
    """
    给定一组 X, y, theta，计算用 X*theta 预测 y 的岭回归损失函数

    Args:
        X - 特征向量，数组大小 (num_instances, num_features)
        y - 标签向量，数组大小 (num_instances)
        theta - 参数向量，数组大小 (num_features)
        lambda_reg - 正则化系数

    Return:
        loss - 损失函数，标量
    """
    # TODO 2.2.2
    m = X.shape[0]  # 样本数量
    h_theta = X @ theta  # 预测值
    loss = np.sum((h_theta - y) ** 2) / (2 * m)  # 均方误差
    regularization = lambda_reg * np.sum(theta ** 2) / 2  # 正则化项
    J = (loss + regularization)
    return J
```

2.2.3 $J(\theta)$ θ 梯度的矩阵形式

$$\nabla_\theta J(\theta) = \frac{1}{m} X^T (X\theta - y) + \lambda\theta$$

2.2.4 compute_regularized_square_loss_gradient

```python
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    """
    计算岭回归损失函数的梯度

    参数:
        X - 特征向量，数组大小 (num_instances, num_features)
        y - 标签向量，数组大小 (num_instances)
        theta - 参数向量，数组大小 (num_features)
        lambda_reg - 正则化系数

    返回:
        grad - 梯度向量，数组大小 (num_features)
    """
    # TODO 2.2.4
    m = X.shape[0]
    h_theta = X @ theta  # 预测值
    error = h_theta - y
    grad = (X.T @ error) / m + lambda_reg * theta  # 计算梯度
    # grad = grad * 2
    return grad
```

2.3 梯度下降 (12pt)

2.3.1 θ 更新表达式

在最小化 $J(\theta)$ 时，假设从 θ 迈出一步到 $\theta + \eta h$，其中 $h \in R^{d+1}$ 是前进方向，$\eta \in (0, \infty)$ 是步长。目标函数的变化可以用泰勒展开的近似公式表示为：

$$J(\theta + \eta h) - J(\theta) \approx \eta \nabla J(\theta)^\top h.$$

当 $h = -\nabla J(\theta)$ 时，目标函数的下降最快，因为此时 $\nabla J(\theta)^\top h$ 为负值且最大。因此，梯度下降算法中更新 θ 的表达式为：

$$\theta := \theta - \eta \nabla J(\theta).$$

2.3.2 gradient_descent

main 函数已将数据加载并拆分为训练集和测试集。你需要实现梯度下降算法，通过以下伪代码来更新参数 θ:

while not converged:

$$\theta := \theta - \eta \nabla J(\theta)$$

```
    def grad_descent(X, y, lambda_reg, alpha=0.1, num_iter=1000,
        ↪ check_gradient=False):
"""
全批量梯度下降算法

Args:
    X - 特征向量，数组大小 (num_instances, num_features)
    y - 标签向量，数组大小 (num_instances)
    lambda_reg - 正则化系数，可自行调整为默认值以外的值
    alpha - 梯度下降的步长，可自行调整为默认值以外的值
    num_iter - 要运行的迭代次数，可自行调整为默认值以外的值
    check_gradient - 更新时是否检查梯度

Return:
    theta_hist - 存储迭代中参数向量的历史，大小为 (num_iter+1,
        ↪ num_features) 的二维 numpy 数组
    loss_hist - 全批量损失函数的历史，大小为 (num_iter) 的一维 numpy 数
        ↪ 组
"""
num_instances, num_features = X.shape[0], X.shape[1]
theta_hist = np.zeros((num_iter + 1, num_features))  # Initialize
    ↪ theta_hist
theta_hist[0] = theta = np.zeros(num_features)  # Initialize theta
loss_hist = np.zeros(num_iter)  # Initialize loss_hist

# TODO 2.3.3

for i in range(num_iter):
    # 计算当前梯度
    grad = compute_regularized_square_loss_gradient(X, y, theta,
        ↪ lambda_reg)

    # 如果需要检查梯度
    if check_gradient:
        grad_diff = grad_checker(X, y, theta, lambda_reg)
        print(f"Iteration {i}, Gradient Check Diff: {grad_diff}")
```

```
        # 更新 theta
        theta = theta - alpha * grad
        # print(f"theta: {theta}, alpha: {alpha}, grad: {grad}")
        # 判断theta是否是NaN
        if np.isnan(theta).any():
            print("theta is NaN")
            exit(1)


        # 记录 theta 和损失
        theta_hist[i + 1] = theta
        loss_hist[i] = compute_regularized_square_loss(X, y, theta,
            ↪ lambda_reg)

    return theta_hist, loss_hist
```

该算法在训练集上迭代更新参数，直到目标函数收敛。

2.3.3 步长选择

选择不同的步长 η（例如：0.5, 0.1, 0.05, 0.01），在训练过程中记录不同步长的表现。通过试验发现：

- 较大的步长（如 $\eta = 0.5$）会导致发散，因为步长过大会导致参数更新幅度过大，目标函数值反而增加 (NaN)。
- 较小的步长（如 $\eta = 0.01, 0.05$）虽然能保证收敛，但收敛速度较慢。
- 步长 $\eta = 0.1$ 是一个折中方案，在本次实验中表现出最快的收敛速度。

绘制不同步长下的目标函数值随训练时间的变化曲线，可以直观地看到各步长的效果差异。

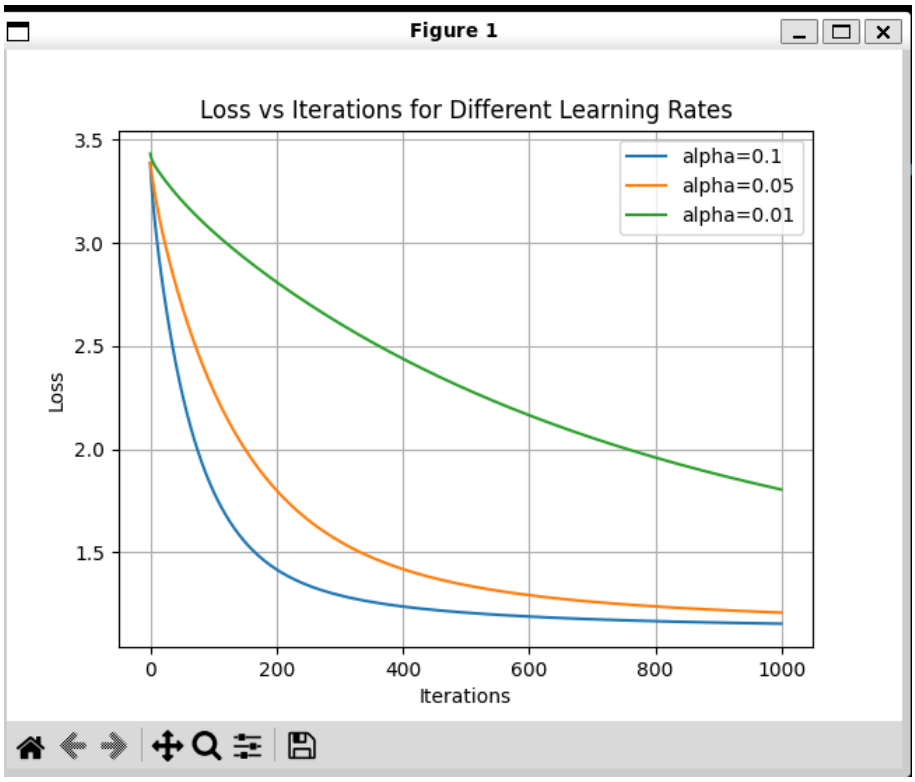

图 1: Loss Curve

2.4 模型选择 (8pt)

2.4.1 K_fold_split_data

我们可以通过验证集上的均方误差 $J_{\text{MSE}}(\theta)$ 来选择合适的模型训练超参数。由于当前还没有验证集，我们可以通过补全函数 `K_fold_split_data`，将训练集分为 K 组（假设 $K = 5$），进行交叉验证，分别形成训练集和验证集。具体步骤如下：

- 将整个训练集划分为 K 个子集，每次选择其中一个子集作为验证集，剩余 $K - 1$ 个子集作为训练集。
- 通过训练集训练模型后，在验证集上计算均方误差 $J_{\text{MSE}}(\theta)$。
- 取 K 次验证集误差的平均值作为该模型在此超参数下的性能评价指标。

```
        cross_validation_K = 5
def K_fold_split_data(X, y, K=cross_validation_K, shuffle=False,
    ↪ random_seed=None):
    """

    """
    num_instances = X.shape[0]
```

```
    if shuffle:
        rng = np.random.RandomState(random_seed)
        indices = rng.permutation(num_instances)
        X = X[indices]
        y = y[indices]
    X_train_list, y_train_list = [], []
    X_valid_list, y_valid_list = [], []

    # TODO 2.5.1
    # 计算每折的大小
    fold_size = num_instances // K
    X_train_list, y_train_list = [], []
    X_valid_list, y_valid_list = [], []

    # 划分数据集
    for k in range(K):
        # 划分验证集
        start_valid = k * fold_size
        end_valid = (k + 1) * fold_size if k < K - 1 else num_instances

        X_valid = X[start_valid:end_valid]
        y_valid = y[start_valid:end_valid]

        # 划分训练集
        X_train = np.concatenate((X[:start_valid], X[end_valid:]), axis=0)
        y_train = np.concatenate((y[:start_valid], y[end_valid:]), axis=0)

        X_train_list.append(X_train)
        y_train_list.append(y_train)
        X_valid_list.append(X_valid)
        y_valid_list.append(y_valid)

    return X_train_list, y_train_list, X_valid_list, y_valid_list
```

2.4.2 K__fold__cross__validation

为了进一步优化模型超参数，我们可以补全函数 `K_fold_cross_validation`，实现 K 折交叉验证。在交叉验证中，使用表格记录不同超参数下模型在验证集上的均方误差 $J_{\text{MSE}}(\theta)$，并通过网格搜索的方式寻找最优的模型训练超参数。搜索范围至少应包括以下几种步长和正则化系数：

- 步长 η 的搜索范围可以包括：$\{0.5, 0.1, 0.05, 0.04, 0.03, 0.02, 0.01\}$。
- 正则化系数 λ 的搜索范围可以包括：$\{10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100\}$。

```
def K_fold_cross_validation(X, y, alphas, lambdas, num_iter=1000, K=
    ↪ cross_validation_K, shuffle=False, random_seed=None):
"""
K 折交叉验证

Args:
    X - 特征向量，一个大小为 (num_instances, num_features) 的二维 numpy
        ↪ 数组
    y - 标签向量，一个大小为 (num_instances) 的一维 numpy 数组
    alphas - 搜索的步长列表
    lambdas - 搜索的正则化系数列表
    num_iter - 要运行的迭代次数
    K - 折数
    shuffle - 是否打乱数据集
    random_seed - 随机种子

Return:
    alpha_best - 最佳步长
    lambda_best - 最佳正则化系数
"""
# alpha_best, lambda_best = None, None
# X_train_list, y_train_list, X_valid_list, y_valid_list =
    ↪ K_fold_split_data(X, y, K, shuffle, random_seed)

# TODO 2.5.2

# 分割数据集
X_train_list, y_train_list, X_valid_list, y_valid_list =
    ↪ K_fold_split_data(X, y, K, shuffle, random_seed)

best_alpha, best_lambda = None, None
min_mse = float('inf')  # 用于记录最小的均方误差

# 用于存储不同超参数组合下的 MSE 值
results_table = []

# 迭代所有可能的 alpha 和 lambda 组合
for alpha in alphas:
    for lambda_reg in lambdas:
        mse_total = 0

        # 对每一折进行交叉验证
        for k in range(K):
            X_train, y_train = X_train_list[k], y_train_list[k]
```

```
                X_valid, y_valid = X_valid_list[k], y_valid_list[k]

                # 训练模型
                theta_hist, _ = grad_descent(X_train, y_train, lambda_reg,
                    ↪ alpha=alpha, num_iter=num_iter)

                # 获取最终的参数 theta
                theta_final = theta_hist[-1]

                # 在验证集上计算均方误差
                h_theta = X_valid @ theta_final
                mse = np.mean((h_theta - y_valid) ** 2)
                mse_total += mse

            # 计算当前超参数组合下的平均 MSE
            avg_mse = mse_total / K
            results_table.append((alpha, lambda_reg, avg_mse))

            # 如果当前组合的 MSE 最小，更新最佳超参数
            if avg_mse < min_mse:
                min_mse = avg_mse
                best_alpha = alpha
                best_lambda = lambda_reg

    # 打印超参数搜索结果
    for alpha, lambda_reg, avg_mse in results_table:
        print(f"alpha={alpha}, lambda={lambda_reg}, avg_mse={avg_mse}")

    return best_alpha, best_lambda
```

实现交叉验证时，每一组超参数组合都会在训练集和验证集上进行训练和评估。最终记录的表格汇报每一组超参数在验证集上的均方误差，选择验证误差最小的超参数作为最优参数。

最后，使用选择出的最优超参数训练模型，并计算模型在测试集上的均方误差 $J_{\text{MSE}}(\theta)$，评估其泛化能力。

最佳超参数组合为 $\alpha = 0.05$ 和 $\lambda = 1 \times 10^{-7}$。

表 1: 不同 α 和 λ 值下的均方误差 (MSE)

α	λ	**avg MSE**
0.05	1×10^{-7}	2.733450
0.05	1×10^{-5}	2.733509
0.05	1×10^{-3}	2.739926
0.05	0.1	4.247802
0.05	1	6.335040
0.05	10	6.807665
0.04	1×10^{-7}	2.773171
0.04	1×10^{-5}	2.773252
0.04	1×10^{-3}	2.781772
0.04	0.1	4.250356
0.04	1	6.335040
0.04	10	6.807665
0.03	1×10^{-7}	2.862424
0.03	1×10^{-5}	2.862530
0.03	1×10^{-3}	2.873326
0.03	0.1	4.262074
0.03	1	6.335040
0.03	10	6.807665
0.02	1×10^{-7}	3.103656
0.02	1×10^{-5}	3.103775
0.02	1×10^{-3}	3.115776
0.02	0.1	4.319813
0.02	1	6.335040
0.02	10	6.807665
0.01	1×10^{-7}	3.883932
0.01	1×10^{-5}	3.884016
0.01	1×10^{-3}	3.892441
0.01	0.1	4.633958
0.01	1	6.335051
0.01	10	6.807665

2.5 随机梯度下降

2.5.1 $\nabla J_{\mathbf{SGD}}(\theta)$

在随机梯度下降中，目标函数 $J_{\text{SGD}}(\theta)$ 是小批量的损失函数。定义如下：

$$J_{\text{SGD}}(\theta) = \frac{1}{n}\sum_{k=1}^{n} f_{i_k}(\theta) + \frac{\lambda}{2}\theta^T\theta$$

其中，$f_{i_k}(\theta)$ 是单个数据点的损失。

对应的梯度可以通过对 $J_{\text{SGD}}(\theta)$ 求导得到：

$$\nabla J_{\text{SGD}}(\theta) = \frac{1}{n}\sum_{k=1}^{n} \nabla f_{i_k}(\theta) + \lambda\theta$$

这里，$\nabla f_{i_k}(\theta)$ 是小批量内第 k 个样本的梯度，$\lambda\theta$ 是正则化项的梯度。

2.5.2 无偏估计证明

假设随机选择的小批量数据点来自数据集中所有样本的独立同分布（i.i.d.）采样。期望值运算可以应用到每个样本的梯度上。即：

$$E_{i_1,i_2,\ldots,i_n}[\nabla J_{\text{SGD}}(\theta)] = E\left[\frac{1}{n}\sum_{k=1}^{n} \nabla f_{i_k}(\theta) + \lambda\theta\right]$$

由于每个 i_k 是独立同分布的，因此：

$$E[\nabla J_{\text{SGD}}(\theta)] = \nabla J(\theta)$$

这表明随机梯度下降中的梯度是目标函数梯度的无偏估计。

2.5.3 stochastic_grad_descent

```
def stochastic_grad_descent(X_train, y_train, X_val, y_val, lambda_reg,
    ↪ alpha=0.1, num_iter=1000, batch_size=1):
    """
    随机梯度下降，并随着训练过程在验证集上验证

    参数：
        X_train - 训练集特征向量，数组大小（num_instances, num_features）
        y_train - 训练集标签向量，数组大小（num_instances）
        X_val - 验证集特征向量，数组大小（num_instances, num_features）
        y_val - 验证集标签向量，数组大小（num_instances）
        alpha - 梯度下降的步长，可自行调整为默认值以外的值
        lambda_reg - 正则化系数，可自行调整为默认值以外的值
        num_iter - 要运行的迭代次数，可自行调整为默认值以外的值
```

```
    batch_size - 批大小，可自行调整为默认值以外的值

返回：
    theta_hist - 参数向量的历史，大小的 2D numpy 数组 (num_iter+1,
        ↪ num_features)
    loss hist - 小批量正则化损失函数的历史，数组大小(num_iter)
    validation hist - 验证集上全批量均方误差（不带正则化项）的历史，数
        ↪ 组大小(num_iter)
"""
num_instances, num_features = X_train.shape[0], X_train.shape[1]
theta_hist = np.zeros((num_iter + 1, num_features))  # Initialize
    ↪ theta_hist
theta_hist[0] = theta = np.zeros(num_features)  # Initialize theta
loss_hist = np.zeros(num_iter)  # Initialize loss_hist
validation_hist = np.zeros(num_iter)  # Initialize validation_hist\

for i in range(num_iter):
    # 随机选择一个小批量数据
    indices = np.random.choice(num_instances, batch_size, replace=False
        ↪ )
    X_batch = X_train[indices]
    y_batch = y_train[indices]

    # 计算梯度
    gradient = compute_regularized_square_loss_gradient(X_batch,
        ↪ y_batch, theta, lambda_reg)

    # 更新参数
    theta = theta - alpha * gradient

    # 保存当前参数向量
    theta_hist[i + 1] = theta

    # 计算当前训练集上的损失
    loss_hist[i] = compute_regularized_square_loss(X_batch, y_batch,
        ↪ theta, lambda_reg)

    # 计算验证集上的损失（不带正则化项）
    validation_errors = X_val.dot(theta) - y_val
    validation_hist[i] = (1 / (2 * len(y_val))) * np.sum(
        ↪ validation_errors**2)

return theta_hist, loss_hist, validation_hist
```

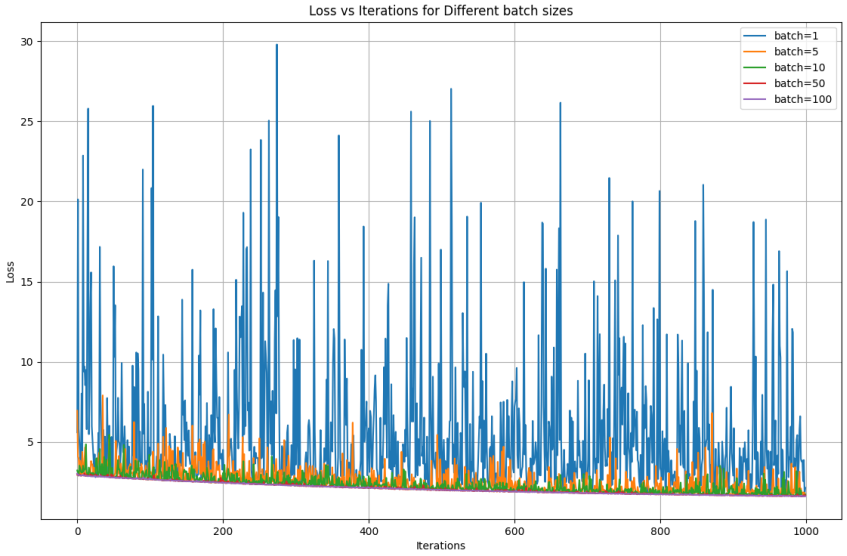

图 2: test_batch_size

2.5.4 选择合适的批大小

从批大小 1 开始，尝试各种不同的批大小，记录随着批大小逐渐增大时的训练曲线

```
def test_batch_sizes(X, y, batch_sizes, lambda_reg=0, alpha=0.01, num_iter
    ↪ =1000):
    """
    测试不同批大小对模型收敛情况的影响，并记录验证集上的全批量损失

    参数:
        X - 输入特征矩阵
        y - 输出标签向量
        batch_sizes - 批大小列表
        lambda_reg - 正则化系数
        alpha - 学习率
        num_iter - 迭代次数

    返回:
        results - 批大小和对应的验证集损失
    """
    # 划分数据集
    (X_train, X_val), (y_train, y_val) = split_data(X, y,shuffle=True,
        ↪ random_seed=0) # 划分数据集,shuffle=True,否则会导致验证集上的损失
        ↪ 越来越大

    print(X_train.shape)
    print(X_val.shape)
    print(y_train.shape)
```

```
    print(y_val.shape)

    # 存储结果
    results = {}

    for batch_size in batch_sizes:
        print(f"Training with batch size: {batch_size}")

        # 进行随机梯度下降
        theta_hist, loss_hist, validation_hist = stochastic_grad_descent(
            X_train, y_train, X_val, y_val, lambda_reg, alpha, num_iter,
            ↪ batch_size)

        # 记录验证集上的最后一次全批量损失
        final_validation_loss = validation_hist[-1]
        results[batch_size] = validation_hist

    # print(results)

    return results


def run_test_batch_sizes(X,y):
    # 测试不同批大小的效果
    batch_sizes = [1, 5, 10, 50, 100]
    results = test_batch_sizes(X, y, batch_sizes)

    # # 绘制验证集损失与批大小的关系

    for batch_size in batch_sizes:
        plt.plot(results[batch_size], label=f"batch={batch_size}")

    plt.title("Loss vs Iterations for Different batch sizes")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.legend()
    plt.grid(True)
    plt.show()
```

2.6 岭回归解析解

岭回归的解析解公式为:

$$\hat{\beta} = (X^T X + \lambda I)^{-1} X^T y \tag{1}$$

其中：

- $\hat{\beta}$ 是模型参数
- X 是特征矩阵
- y 是目标变量
- λ 是正则化系数
- I 是单位矩阵

2.6.1 实现分析解函数

可以使用 NumPy 实现此函数：

```
def analytical_solution(X, y, lambda_reg):
    """
    岭回归解析解

    Args:
        X - 特征向量，一个大小为 (num_instances, num_features) 的二维 numpy
            ↪ 数组
        y - 标签向量，一个大小为 (num_instances) 的一维 numpy 数组
        lambda_reg - 正则化系数

    Return:
        theta - 参数向量
    """
    assert lambda_reg > 0
    # TODO 2.6.1


    # 计算岭回归解析解
    return np.linalg.inv(X.T @ X + lambda_reg * np.eye(X.shape[1])) @ X.T @
        ↪ y
```

2.6.2 不同正则化系数的均方误差记录

使用以下方式记录不同正则化系数的岭回归模型在测试集上的均方误差：

```
def run_test_analytical_solution(X, y):
    # 划分数据集
    (X_train, X_val), (y_train, y_val) = split_data(X, y, shuffle=True,
        ↪ random_seed=0)
    # 假设 X_train, y_train 是训练集, X_test, y_test 是测试集
```

```
lambdas = [0.01, 0.1, 1, 10, 100]
mse_results = []

# 特征归一化
X_train, X_val = feature_normalization(X_train, X_val)

# 增加偏置项
X_train = np.hstack((X_train, np.ones((X_train.shape[0], 1))))
X_val = np.hstack((X_val, np.ones((X_val.shape[0], 1))))

for lambda_ in lambdas:
    theta = analytical_solution(X_train, y_train, lambda_)
    h_theta = X_val @ theta
    mse = np.mean((h_theta - y_val) ** 2)
    mse_results.append(mse)
    print(f"Lambda: {lambda_}, MSE: {mse}")
results_df = pd.DataFrame({'Lambda': lambdas, 'MSE': mse_results})
print(results_df)
```

运行结果为

表 2: 不同正则化系数的均方误差

Lambda	MSE
1.000000e-07	2.583219
1.000000e-03	2.492926
0.10	2.426063
1.00	2.320294
10.00	2.660115
100.00	4.037493

由此可见，在 0.01 到 1.00 的时候，随着正则化系数 λ 的增大，均方误差逐渐减少。

2.6.3 机器学习方法与解析解的比较

1. 计算时间

   - 解析解的计算时间主要依赖于矩阵运算，尤其是矩阵的逆，这通常是较为耗时的。
   - 机器学习方法（例如梯度下降）通常具有更好的时间复杂性，能够逐步收敛到解。

2. 测试集均方误差

- 解析解在数学上是准确的，能提供全局最优解，但在某些情况下可能由于数据噪声或多重共线性而表现不佳。
- 机器学习方法（例如 Lasso、随机森林等）通常具有更强的泛化能力，尤其是在数据较少时，能够有效避免过拟合。

3. 总结 机器学习方法在计算时间和泛化能力上相较于解析解具有一定优势，特别是在处理大规模数据集时。解析解则更适合于特征数量较小且对精确性有较高要求的场景。

3 Softmax 回归

3.1 梯度计算

设交叉熵损失函数 L 为:

$$L = -\log y_k = -y^T \log \hat{y}$$

其中，$\hat{y}_i = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$。

对 z 求偏导数，即可得到 L 关于 z 的梯度:

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i$$

所以，梯度为:

$$\nabla_z L = \hat{y} - y$$

3.2 梯度计算

假设 $z = Wx + b$，则

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial W} = (\hat{y} - y)x^T$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial b} = \hat{y} - y$$

因此，梯度为:

$$\nabla_W L = (\hat{y} - y)x^T$$

$$\nabla_b L = \hat{y} - y$$

3.3 Hessian 矩阵 H。

Hessian 矩阵 H 的元素 $H_{ij} = \frac{\partial^2 L}{\partial z_i \partial z_j}$，可由如下公式给出：

$$H_{ij} = \begin{cases} \hat{y}_i(1 - \hat{y}_i), & \text{if } i = j \\ -\hat{y}_i \hat{y}_j, & \text{if } i \neq j \end{cases}$$

因此，Hessian 矩阵为：

$$H = \text{diag}(\hat{y}) - \hat{y}\hat{y}^T$$

3.4 ）证明矩阵 H 是半正定的

矩阵 $H = \text{diag}(\hat{y}) - \hat{y}\hat{y}^T$ 是一个半正定矩阵。因为 $\hat{y}$ 是 Softmax 概率分布，所有的元素 $\hat{y}_i \geq 0$ 且 $\sum_i \hat{y}_i = 1$。根据定义，Hessian 矩阵是半正定的。

4 支持向量机

4.1 次梯度

对于合页损失函数（Hinge Loss）：

$$J(w) = \max\{0, 1 - yw^T x\}$$

根据次梯度定义，对于 $g \in R^d$，满足：

$$f(z) \geq f(x) + g^T(z - x)$$

我们可以求出 $J(w)$ 的次梯度。

当 $1 - yw^T x > 0$ 时，损失函数为 $1 - yw^T x$，次梯度为：

$$\frac{\partial J(w)}{\partial w} = -yx$$

当 $1 - yw^T x \leq 0$ 时，损失函数为 0，次梯度为：

$$\frac{\partial J(w)}{\partial w} = 0$$

因此，合页损失函数的次梯度为：

$$\frac{\partial J(w)}{\partial w} = \begin{cases} -yx, & \text{if } 1 - yw^T x > 0 \\ 0, & \text{if } 1 - yw^T x \leq 0 \end{cases}$$

4.2 硬间隔支持向量机

4.2.1 KKT 条件与证明

首先给出带约束的优化问题：

$$\min_{w\in R^d, b\in R} \quad \frac{1}{2}\|w\|^2$$

$$\text{s.t.} \quad y_i(w^T x_i + b) \geq 1, \quad 1 \leq i \leq m$$

拉格朗日函数为：

$$L(w, b, \mu) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{m} \mu_i [y_i(w^T x_i + b) - 1]$$

根据 KKT 条件，需要满足以下条件：

1. 梯度条件（stationarity）：

$$\frac{\partial L(w, b, \mu)}{\partial w} = w - \sum_{i=1}^{m} \mu_i y_i x_i = 0$$

$$\frac{\partial L(w, b, \mu)}{\partial b} = -\sum_{i=1}^{m} \mu_i y_i = 0$$

2. 可行性条件（primal feasibility）：

$$y_i(w^T x_i + b) \geq 1, \quad 1 \leq i \leq m$$

3. 对偶可行性条件（dual feasibility）：

$$\mu_i \geq 0, \quad 1 \leq i \leq m$$

4. 互补松弛条件（complementary slackness）：

$$\mu_i [y_i(w^T x_i + b) - 1] = 0, \quad 1 \leq i \leq m$$

4.2.2 证明

根据梯度条件：

$$w = \sum_{i=1}^{m} \mu_i y_i x_i$$

可以得出权向量 w 是训练数据 $x_1, x_2, ..., x_m$ 的线性组合。

同时，互补松弛条件：

$$\mu_i [y_i(w^T x_i + b) - 1] = 0$$

说明当 $\mu_i \neq 0$ 时，必须有 $y_i(w^T x_i + b) = 1$，这表明支持向量位于分离超平面 $w \cdot x + b = \pm 1$ 上。

因此，满足 KKT 条件的解 w 一定是训练数据的线性组合，且支持向量位于超平面 $w \cdot x + b = \pm 1$ 上。

4.3 软间隔支持向量机

线性可分是一种理想的情况，现实数据集经常会有噪声，无法约束所有的数据点都能被正确分类，因此实际的支持向量机会引入软间隔，目标是让不满足约束的数据点尽可能少，即：

$$\min_{w\in R^d, b\in R, \xi\in R^m} \quad \frac{\lambda}{2}\|w\|^2 + \frac{1}{p}\sum_{i=1}^{m}\xi_i^p$$

$$\text{s.t.} \quad y_i(w^T x_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0, \quad 1 \leq i \leq m$$

其中，参数 $p \geq 1$ 用于控制对数据点不满足约束的惩罚力度。

4.3.1 拉格朗日形式

要求 $p = 1$ 时，该问题的拉格朗日 (Lagrange) 形式为：

$$L(w, b, \xi, \mu, \alpha) = \frac{\lambda}{2}\|w\|^2 + \sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i[y_i(w^T x_i + b) - 1 + \xi_i] - \sum_{i=1}^{m}\mu_i\xi_i$$

其中，$\alpha_i \geq 0$ 是对分类约束的拉格朗日乘子，$\mu_i \geq 0$ 是对松弛变量非负性的拉格朗日乘子。

4.3.2 对偶形式

当 $p = 1$ 时，问题的对偶形式 (Dual Form) 为：

$$\max_{\alpha} \quad \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha_i\alpha_j y_i y_j (x_i^T x_j)$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq 1, \quad \sum_{i=1}^{m}\alpha_i y_i = 0$$

4.3.3 次梯度下降法

在求出对偶问题后，使用 Sequential Minimal Optimization (SMO) 算法，可以实现软间隔支持向量机的实际求解。对于 $p = 1$，上文中的优化问题也可以等价表示为：

$$\min_{w,b} \frac{\lambda}{2}\|w\|^2 + \sum_{i=1}^{m}\max\{0, 1 - y_i(w^T x_i + b)\}$$

此时，SVM 求解还可以通过另一种基于 ** 次梯度下降法 ** 的算法。证明此时软间隔支持向量机中：

$$J(w) = \frac{\lambda}{2}\|w\|^2 + \sum_{i=1}^{m}\max\{0, 1 - y_i(w^T x_i + b)\}$$

的次梯度可以表示为：

$$\partial J_w|_b = \begin{cases} \lambda w - \sum_{i=1}^{m} y_i x_i & \text{for} \quad y_i(w^T x_i + b) < 1, \\ \lambda w & \text{for} \quad y_i(w^T x_i + b) \geq 1 \end{cases}$$

对于 $J_b(w)$ 的梯度为：

$$\partial J_b|_b = \begin{cases} -\sum_{i=1}^{m} y_i & \text{for} \quad y_i(w^T x_i + b) < 1, \\ 0 & \text{for} \quad y_i(w^T x_i + b) \geq 1 \end{cases}$$

4.3.4 梯度表达式

对于 $p > 1$，次梯度表达式为：

$$J(w) = \frac{\lambda}{2}\|w\|^2 + \left(\sum_{i=1}^{m} \max\{0, 1 - y_i(w^T x_i + b)\}^p\right)^{1/p}$$

其次梯度如下：

$$\partial J_w|_b = \lambda w + \left(\sum_{i=1}^{m} \max\{0, 1 - y_i(w^T x_i + b)\}^{p-1}\right)$$

4.4 核方法

4.4.1 基函数

函数 $k(x, x') = \cos\angle(x, x')$ 可以视作两个向量之间的余弦相似度。由于 $\cos\angle(x, x')$ 满足对称性，且任意两个向量的余弦相似度的加权和非负，因此矩阵 $K = [k(x_i, x_j)]_{i,j}$ 是对称半正定的。

对于核函数 $k(x_1, x_2) = \cos\angle(x_1, x_2)$，我们可以取基函数 $\Phi(x) = \frac{x}{\|x\|}$，从而有：

$$k(x_1, x_2) = \Phi(x_1) \cdot \Phi(x_2)$$

4.4.2 证明

核函数之和 $k_s(x, x') = k_1(x, x') + k_2(x, x')$ 仍为核函数。因为 k_1 和 k_2 对应的核矩阵都是对称半正定矩阵 K_1, K_2，所以 $K_s = K_1 + K_2$ 也对称半正定。因此，k_s 是核函数。

类似地，核函数之积 $k_p(x, x') = k_1(x, x') \cdot k_2(x, x')$ 也是核函数。因为 $K_p = K_1 \circ K_2$（Hadamard 积）仍为对称半正定矩阵，因此 k_p 也是核函数。

4.4.3 核函数

支持向量机的优化问题形式为：

$$\min_{\alpha} \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j k(x_i, x_j) - \sum_{i=1}^{m} \alpha_i$$

约束条件为：

$$\sum_{i=1}^{m} \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, m$$

对于测试样本 x'，使用核函数的支持向量机分类决策函数为：

$$f(x') = \sum_{i=1}^{m} \alpha_i y_i k(x_i, x') + b$$

如果 $f(x') > 0$，则分类为正类；否则分类为负类。

4.5 情绪检测

4.5.1 线性 SVM 的随机次梯度下降

在 linear_svm_subgrad_descent 函数中实现线性 SVM 的随机次梯度下降算法。主要思路是使用随机选择的样本来更新模型参数。

```
def linear_svm_subgrad_descent(X, y, alpha=0.05, lambda_reg=0.0001,
    ↪ num_iter=60000, batch_size=1):
    num_instances, num_features = X.shape
    theta = np.zeros(num_features)  # Initialize theta
    loss_hist = np.zeros(num_iter)  # Initialize loss_hist

    for i in range(num_iter):
        indices = np.random.choice(num_instances, batch_size, replace=False
            ↪ )  # 随机选择batch_size个样本
        X_batch = X[indices]
        y_batch = y[indices]

        # 计算损失和梯度
        margins = 1 - y_batch * (X_batch @ theta)
        loss = np.maximum(0, margins)
        loss_mean = np.mean(loss) + (lambda_reg / 2) * np.dot(theta, theta)
        loss_hist[i] = loss_mean

        # 计算梯度
        gradient = -np.mean((y_batch * (margins > 0)).reshape(-1, 1) *
            ↪ X_batch, axis=0) + lambda_reg * theta
        theta -= alpha * gradient  # 更新参数
```

```
    return theta, loss_hist
```

4.5.2 调参

调整超参数，如步长、正则化参数等，并记录训练和验证集的准确率

```
def tune_hyperparameters(X_train, y_train, X_val, y_val):
    results = []

    for alpha in [0.01, 0.05, 0.1]:
        for lambda_reg in [0.0001, 0.001]:
            for batch_size in [1, 10, 50]:
                theta, loss_hist = linear_svm_subgrad_descent(X_train,
                    ↪ y_train, alpha=alpha, lambda_reg=lambda_reg, num_iter
                    ↪ =60000, batch_size=batch_size)

                # 计算训练集和验证集准确率
                train_accuracy = np.mean(np.sign(X_train @ theta) ==
                    ↪ y_train)
                val_accuracy = np.mean(np.sign(X_val @ theta) == y_val)
                results.append((alpha, lambda_reg, batch_size,
                    ↪ train_accuracy, val_accuracy))

    return results
```

调参结果如下

步长	正则化	批大小	训练准确率	验证准确率
0.01	0.0001	1	0.9006	0.8395
0.01	0.0001	10	0.8877	0.8344
0.01	0.0001	50	0.8908	0.8359
0.01	0.001	1	0.9025	0.8366
0.01	0.001	10	0.8730	0.8214
0.01	0.001	50	0.8672	0.8142
0.05	0.0001	1	0.9546	0.8713
0.05	0.0001	10	0.9564	0.8648
0.05	0.0001	50	0.9574	0.8641
0.05	0.001	1	0.9258	0.8474
0.05	0.001	10	0.9212	0.8568
0.05	0.001	50	0.9273	0.8576
0.1	0.0001	1	0.9660	0.8612
0.1	0.0001	10	0.9709	0.8590
0.1	0.0001	50	0.9721	0.8597
0.1	0.001	1	0.8902	0.8286
0.1	0.001	10	0.9276	0.8576
0.1	0.001	50	0.9276	0.8590

最终验证集准确率: 0.8706

4.5.3 基于核函数的 SVM

在 kernel_svm_subgrad_descent 中实现基于线性核或高斯核的非线性 SVM。以高斯核为例。

```
def gaussian_kernel(X, sigma=1.0):
    K = np.exp(-np.linalg.norm(X[:, None] - X, axis=2) ** 2 / (2 * sigma **
        ↪ 2))
    return K

def kernel_svm_subgrad_descent(X, y, alpha=0.1, lambda_reg=1, num_iter
    ↪ =1000, batch_size=1, sigma=1.0):
    num_instances = X.shape[0]
    K = gaussian_kernel(X, sigma)
    theta = np.zeros(num_instances)  # Initialize theta
    loss_hist = np.zeros(num_iter)  # Initialize loss_hist

    for i in range(num_iter):
```

```
        indices = np.random.choice(num_instances, batch_size, replace=False
            ↪ )  # 随机选择batch_size个样本
        y_batch = y[indices]
        K_batch = K[indices][:, indices]

        margins = 1 - y_batch * (theta @ K_batch)  # 使用核函数计算边际
        loss = np.maximum(0, margins)
        loss_mean = np.mean(loss) + (lambda_reg / 2) * np.dot(theta, theta)
        loss_hist[i] = loss_mean

        # 计算梯度
        gradient = -np.mean((y_batch * (margins > 0))[:, None] * K_batch,
            ↪ axis=0) + lambda_reg * theta
        theta -= alpha * gradient  # 更新参数

    return theta, loss_hist
```

4.5.4 模型性能

计算模型在验证集上的准确率、F1-Score 和混淆矩阵。

```
def evaluate_model(X_val, y_val, theta):
    predictions = np.sign(X_val @ theta)
    accuracy = np.mean(predictions == y_val)
    f1 = f1_score(y_val, predictions)
    cm = confusion_matrix(y_val, predictions)
    return accuracy, f1, cm
```

计算结果如下最终验证集准确率: 0.8706
F1-Score: 0.8765

$$\text{混淆矩阵:} \quad \begin{bmatrix} 569 & 107 \\ 72 & 635 \end{bmatrix}$$

4.5.5 逻辑斯特回归

逻辑斯特回归的目标函数和梯度如下:

- 目标函数: $J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\|\theta\|^2$
- 梯度: $\nabla J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta(x^{(i)})\right)x^{(i)} + \frac{\lambda}{m}\theta$

5 总结

在函数 test_batch_sizes 中，分割数据的时候，使用了 split_data(X, y) 忘记了设置 shuffle=True, random_seed=0，导致验证集的 loss 随着迭代次数的增加而增大。

在训练过程中计算时间很长。