Exercise 1

如果你有一个具有数百万个特征的训练集，你可以使用哪种线性回归训练算法？

答案：

如果你有一个包含数百万个特征的训练集，你可以使用随机梯度下降或小批量梯度下降，如果训练集适合内存，也可以使用批量梯度下降。但是您不能使用 标准方程 或 SVD 方法，因为计算复杂度会随着特征数量的增加而快速增长（超过二次增长）。

Exercise 2

假设你的训练集中的特征有非常不同的尺度。哪一种算法可能会失效，以及如何失效呢？你能做些什么呢？

答案：

如果训练集中的特征具有非常不同的尺度，则成本函数将具有细长碗的形状，因此梯度下降算法将需要很长时间才能收敛。要解决这个问题，您应该在训练模型之前缩放数据。请注意，标准方程或 SVD 方法无需缩放即可正常工作。此外，如果特征未缩放，则正则化模型可能会收敛到次优解决方案：由于正则化会惩罚大权重，因此与具有较大值的特征相比，具有较小值的特征往往会被忽略。

Exercise 3

当训练逻辑回归模型时，梯度下降会陷入局部最小值吗？

答案：

梯度下降在训练逻辑回归模型时不会陷入局部最小值，因为成本函数是凸的。 凸意味着如果你在曲线上的任意两点之间画一条直线，这条直线永远不会与曲线相交。

Exercise 4

如果你让梯度下降算法运行的时间足够长，那么所有的梯度下降算法都能产生相同的模型吗？

答案：

如果优化问题是凸的（例如线性回归或逻辑回归），并且假设学习率不是太高，那么所有梯度下降算法都会接近全局最优并最终产生非常相似的模型。但是，除非你逐渐降低学习率，否则 Stochastic GD 和 Mini-batch GD 永远不会真正收敛；相反，他们将继续围绕全局最优值来回跳跃。这意味着即使你让它们运行很长时间，这些梯度下降算法也会产生略有不同的模型。

Exercise 5

假设您使用批梯度下降并每个历元绘制验证误差。如果您注意到验证误差持续上升，那么可能发生了什么呢？你该如何解决这个问题呢？

答案：

如果验证误差在每个 epoch 之后都持续上升，那么一种可能性是学习率太高并且算法正在发散。 如果训练误差也上升，那么这显然是问题所在，您应该降低学习率。但是，如果训练误差没有上升，那么你的模型对训练集过度拟合，你应该停止训练。

Exercise 6

当验证错误上升时，立即停止小批量梯度下降是一个好主意吗？

答案：

由于它们的随机性，随机梯度下降和小批量梯度下降都不能保证在每次训练迭代中都能取得进展。 因此，如果您在验证误差上升时立即停止训练，您可能会在达到最佳值之前停止得太早。 更好的选择是定期保存模型； 然后，当它长时间没有改进时（这意味着它可能永远不会打破记录），您可以恢复到保存最好的模型。

Exercise 7

哪种梯度下降算法（在我们讨论过的那些算法中）将最快地到达最优解的附近？哪些会真正收敛？如何才能让其他算法也收敛起来呢？

答案：

随机梯度下降具有最快的训练迭代，因为它一次只考虑一个训练实例，所以它通常最先到达全局最优（或 mini-batch 大小非常小的 Mini-batch GD）附近。 然而，如果有足够的训练时间，只有批量梯度下降会真正收敛。 如前所述，Stochastic GD 和 Mini-batch GD 会在最优值附近反弹，除非你逐渐降低学习率。

Exercise 8

假设你正在使用多项式回归。你绘制学习曲线，你会注意到训练错误和验证错误之间有很大的差距。发生了什么？有三种解决这个问题的方法是什么？

答案：

如果验证误差远高于训练误差，这可能是因为您的模型过度拟合了训练集。 尝试解决此问题的一种方法是降低多项式次数：自由度较低的模型不太可能过度拟合。 您可以尝试的另一件事是对模型进行正则化——例如，通过向成本函数添加 ℓ_2 惩罚 (Ridge) 或 ℓ_1 惩罚 (Lasso)。 这也会降低模型的自由度。 最后，您可以尝试增加训练集的大小。

Exercise 9

假设您使用岭回归，您注意到训练误差和验证误差几乎相等，而且相当高。你会说这个模型存在高偏差或高方差吗？你应该增加正则化超参数 α 还是减少它？

答案：

如果训练误差和验证误差几乎相等且相当高，则该模型可能欠拟合训练集，这意味着它具有高偏差。 您应该尝试减少正则化超参数 α 。

Exercise 10

您为什么要使用：

1. 岭回归而不是普通的线性回归（即，没有任何正则化）？
2. 套索回归而不是岭回归？
3. 弹性网络回归而不是套索回归？

答案：

- 具有一些正则化的模型通常比没有任何正则化的模型表现更好，因此您通常应该更喜欢岭回归而不是普通线性回归。
- 套索回归使用 ℓ_1 惩罚，它倾向于将权重降低到恰好为零。这会导致稀疏模型，其中除最重要的权重外所有权重均为零。这是一种自动执行特征选择的方法，如果您怀疑只有少数特征真正重要，这很好。当你不确定时，你应该更喜欢岭回归。
- 弹性网络回归通常比套索回归更受青睐，因为套索回归在某些情况下可能表现不稳定（当多个特征强相关或特征多于训练实例时）。但是，它确实添加了一个额外的超参数来调整。如果你想要套索回归没有不稳定的行为，你可以只使用 **l1_ratio** 接近 1 的弹性网络回归。

Exercise 11

假设您想将图片分类为 outdoor/indoor 和 daytime/nighttime 。你应该实现两个逻辑回归分类器还是一个softmax回归分类器？

答案：

如果你想将图片分类为户外/室内和白天/夜间，因为这些不是排他性的类别（即所有四种组合都是可能的）你应该训练两个逻辑回归分类器。

Exercise 12

实现批梯度下降与早期停止的softmax回归不使用 Scikit-Learn ，只有 NumPy 。使用它的分类任务，如鸢尾花数据集。

答案：

让我们从加载数据开始。 我们将重用我们之前加载的 Iris 数据集。

In [61]:
```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

In [2]:
```
from sklearn.datasets import load_iris

iris = load_iris(as_frame=True)

X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = iris["target"].values
```

我们需要为每个实例添加偏差项($x_0 = 1$)。最简单的选择是使用 Scikit-Learn 的 add_dummy_feature() 函数，但本练习的目的是通过手动实现算法来更好地理解算法。所以这是一种可能的实现：

In [7]:
```
X_with_bias = np.c_[np.ones(len(X)), X]
```

```
X_with_bias[:5]
```

Out[7]:
```
array([[1. , 1.4, 0.2],
       [1. , 1.4, 0.2],
       [1. , 1.3, 0.2],
       [1. , 1.5, 0.2],
       [1. , 1.4, 0.2]])
```

将数据集拆分为训练集、验证集和测试集的最简单方法是使用 Scikit-Learn 的 train_test_split() 函数，但同样，我们希望手动完成：

In [8]:
```
test_ratio = 0.2
validation_ratio = 0.2
total_size = len(X_with_bias)

test_size = int(total_size * test_ratio)
validation_size = int(total_size * validation_ratio)
train_size = total_size - test_size - validation_size

np.random.seed(42)

rnd_indices = np.random.permutation(total_size)

X_train = X_with_bias[rnd_indices[:train_size]]
y_train = y[rnd_indices[:train_size]]

X_valid = X_with_bias[rnd_indices[train_size:-test_size]]
y_valid = y[rnd_indices[train_size:-test_size]]

X_test = X_with_bias[rnd_indices[-test_size:]]
y_test = y[rnd_indices[-test_size:]]
```

当前目标是类别索引（0、1 或 2），但我们需要目标类别概率来训练 Softmax 回归模型。 除了目标类的概率为 1.0（换句话说，任何给定实例的类概率向量是一个独热向量）之外，每个实例的目标类概率对于所有类都将等于 0.0。让我们编写一个小函数，将类索引向量转换为包含每个实例的单热向量的矩阵。要理解这段代码，您需要知道 `np.diag(np.ones(n))` 创建了一个 $n \times n$ 矩阵，除了主对角线上的 1 之外全是 0。此外，如果 a 是 NumPy 数组，则 a[[1, 3, 2]] 返回一个包含 3 行等于 a[1]、a[3] 和 a[2] 的数组（这是高级 NumPy 索引）。

In [15]:
```
def to_one_hot(y):
    return np.diag(np.ones(y.max() + 1))[y]
```

In [16]:
```
y_train[:10]
```

Out[16]:
```
array([1, 0, 2, 1, 1, 0, 1, 2, 1, 1])
```

In [17]:
```
to_one_hot(y_train[:10])
```

Out[17]:
```
array([[0., 1., 0.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 1., 0.],
       [1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 1., 0.]])
```

看起来不错，让我们为训练集和测试集创建目标类概率矩阵：

In [18]:
```
Y_train_one_hot = to_one_hot(y_train)
Y_valid_one_hot = to_one_hot(y_valid)
Y_test_one_hot = to_one_hot(y_test)
```

现在让我们缩放输入。我们计算训练集上每个特征的均值和标准差（偏差特征除外），然后我们对训练集、验证集和测试集中的每个特征进行居中和缩放：

In [19]:
```
mean = X_train[:, 1:].mean(axis=0)
std = X_train[:, 1:].std(axis=0)

X_train[:, 1:] = (X_train[:, 1:] - mean) / std
X_valid[:, 1:] = (X_valid[:, 1:] - mean) / std
X_test[:, 1:] = (X_test[:, 1:] - mean) / std
```

现在让我们来实现 Softmax 函数。 回想一下，它由以下等式定义：

$$\sigma(\boldsymbol{s}(\boldsymbol{x}))_k = \frac{exp(s_k(\boldsymbol{x}))}{\sum_{j=1}^{K} exp(s_j(\boldsymbol{x}))}$$

In [21]:
```
def softmax(logits):
    exps = np.exp(logits)
    exp_sums = exps.sum(axis=1, keepdims=True)
    return exps / exp_sums
```

我们几乎准备好开始训练了。 让我们定义输入和输出的数量：

In [22]:
```
n_inputs = X_train.shape[1]  # == 3 (2 features plus the bias term)

n_outputs = len(np.unique(y_train))  # == 3 (there are 3 iris classes)
```

现在最难的部分来了：训练！从理论上讲，这很简单：只需将数学方程式转换为 Python 代码即可。但在实践中，这可能非常棘手：特别是，很容易混淆项或索引的顺序。您甚至可以得到看起来可以正常工作但实际上计算不正确的代码。当不确定时，您应该记下等式中每一项的形状，并确保代码中的相应项紧密匹配。 它还可以帮助独立评估每个项并将其打印出来。好消息是你不必每天都这样做，因为所有这些都由 Scikit-Learn 很好地实现了，但它会帮助你了解幕后发生的事情。

所以我们需要的方程是成本函数：

$$J(\boldsymbol{\Theta}) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

以及梯度方程：

$$\nabla_{\theta^{(k)}} J(\boldsymbol{\Theta}) = \frac{1}{m}\sum_{i=1}^{m}(\hat{p}_k^{(i)} - y_k^{(i)})\boldsymbol{x}^{(i)}$$

注意：如果 $\hat{p}_k^{(i)} = 0$ 的话 $log(\hat{p}_k^{(i)})$ 可能无法计算，所以我们对 $log(\hat{p}_k^{(i)})$ 加一个极小项 ϵ 以避免获得 `nan` 值。

In [51]:
```
eta = 0.5
n_epochs = 5001
m = len(X_train)
epsilon = 1e-5

np.random.seed(42)
```

```
Theta = np.random.randn(n_inputs, n_outputs)

for epoch in range(n_epochs):
    logits = X_train @ Theta
    Y_proba = softmax(logits)
    if epoch % 1000 == 0:
        Y_proba_valid = softmax(X_valid @ Theta)
        xentropy_losses = -(Y_valid_one_hot * np.log(Y_proba_valid + epsilon))
        print(epoch, xentropy_losses.sum(axis=1).mean())
    error = Y_proba - Y_train_one_hot
    gradients = 1 / m * X_train.T @ error
    Theta = Theta - eta * gradients
```

```
0 3.7085808486476917
1000 0.14519367480830644
2000 0.1301309575504088
3000 0.12009639326384539
4000 0.11372961364786884
5000 0.11002459532472425
```

就是这样！训练了 Softmax 模型。让我们看一下模型参数：

In [52]:
```
Theta
```

Out[52]:
```
array([[ 0.41931626,  6.11112089, -5.52429876],
       [-6.53054533, -0.74608616,  8.33137102],
       [-5.28115784,  0.25152675,  6.90680425]])
```

让我们对验证集进行预测并检查准确度分数：

In [53]:
```
logits = X_valid @ Theta

Y_proba = softmax(logits)

y_predict = Y_proba.argmax(axis=1)

accuracy_score = (y_predict == y_valid).mean()

accuracy_score
```

Out[53]:
```
0.9333333333333333
```

嗯，这个模型看起来还不错。为了练习，让我们添加一些 l_2 正则化。 下面的训练代码和上面的类似，但是损失现在多了一个 l_2 惩罚，并且梯度有适当的附加项（请注意，我们没有对 Theta 的第一个元素进行正则化，因为它对应于偏置项）。另外，让我们尝试增加学习率 eta。

In [54]:
```
eta = 0.5
n_epochs = 5001
m = len(X_train)
epsilon = 1e-5
alpha = 0.01  # regularization hyperparameter

np.random.seed(42)
Theta = np.random.randn(n_inputs, n_outputs)

for epoch in range(n_epochs):
    logits = X_train @ Theta
    Y_proba = softmax(logits)
    if epoch % 1000 == 0:
        Y_proba_valid = softmax(X_valid @ Theta)
        xentropy_losses = -(Y_valid_one_hot * np.log(Y_proba_valid + epsilon))
        l2_loss = 1 / 2 * (Theta[1:] ** 2).sum()
        total_loss = xentropy_losses.sum(axis=1).mean() + alpha * l2_loss
```

```
        print(epoch, total_loss.round(4))
    error = Y_proba - Y_train_one_hot
    gradients = 1 / m * X_train.T @ error
    gradients += np.r_[np.zeros([1, n_outputs]), alpha * Theta[1:]]
    Theta = Theta - eta * gradients
```

```
0 3.7372
1000 0.3259
2000 0.3259
3000 0.3259
4000 0.3259
5000 0.3259
```

因为额外的 l_2 惩罚，损失似乎比之前更大，但也许这个模型会表现得更好？让我们看看：

In [55]:
```
logits = X_valid @ Theta

Y_proba = softmax(logits)

y_predict = Y_proba.argmax(axis=1)

accuracy_score = (y_predict == y_valid).mean()

accuracy_score
```

Out[55]:
```
0.9333333333333333
```

在这种情况下 l_2 惩罚并没有改变测试的准确性。 也许尝试微调 alpha？

In [56]:
```
eta = 0.5
n_epochs = 50_001
m = len(X_train)
epsilon = 1e-5
C = 100  # regularization hyperparameter
best_loss = np.infty

np.random.seed(42)
Theta = np.random.randn(n_inputs, n_outputs)

for epoch in range(n_epochs):
    logits = X_train @ Theta
    Y_proba = softmax(logits)
    Y_proba_valid = softmax(X_valid @ Theta)
    xentropy_losses = -(Y_valid_one_hot * np.log(Y_proba_valid + epsilon))
    l2_loss = 1 / 2 * (Theta[1:] ** 2).sum()
    total_loss = xentropy_losses.sum(axis=1).mean() + 1 / C * l2_loss
    if epoch % 1000 == 0:
        print(epoch, total_loss.round(4))
    if total_loss < best_loss:
        best_loss = total_loss
    else:
        print(epoch - 1, best_loss.round(4))
        print(epoch, total_loss.round(4), "early stopping!")
        break
    error = Y_proba - Y_train_one_hot
    gradients = 1 / m * X_train.T @ error
    gradients += np.r_[np.zeros([1, n_outputs]), 1 / C * Theta[1:]]
    Theta = Theta - eta * gradients
```

```
0 3.7372
281 0.3256
282 0.3256 early stopping!
```

In [57]:
```
logits = X_valid @ Theta
Y_proba = softmax(logits)
```

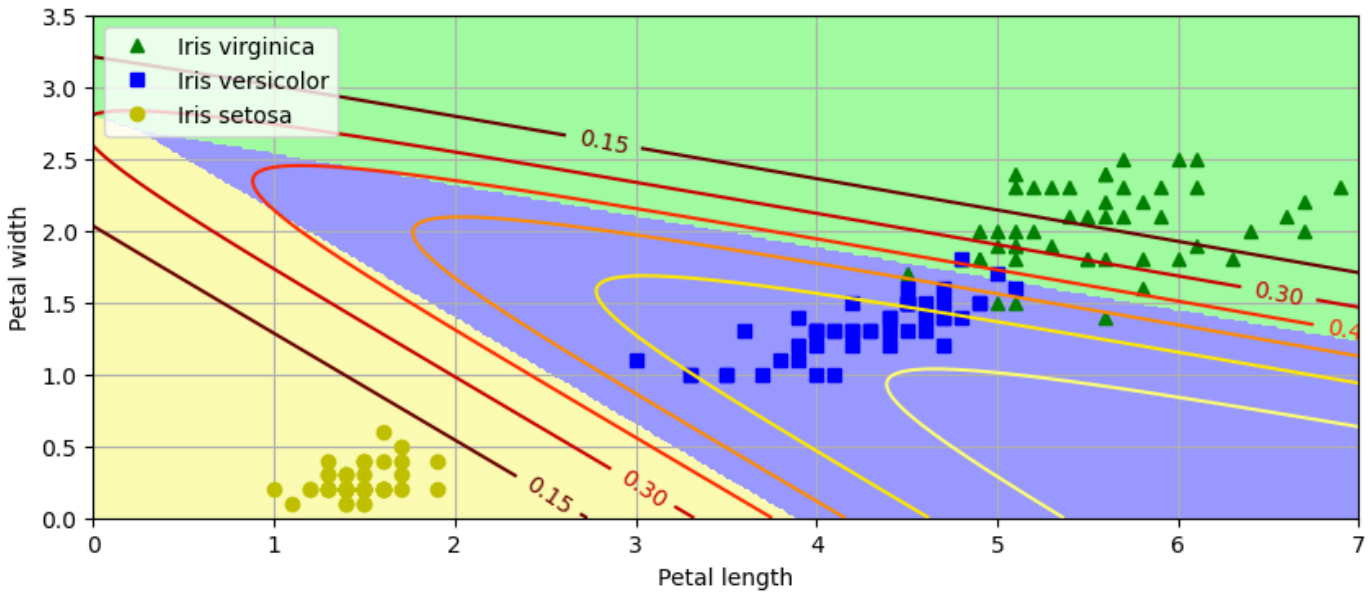
```
y_predict = Y_proba.argmax(axis=1)

accuracy_score = (y_predict == y_valid).mean()
accuracy_score
```

Out[57]:
```
0.9333333333333333
```

哦，好吧，验证准确性仍然没有变化，但至少提前停止缩短了一点训练时间。

现在让我们绘制模型对整个数据集的预测（记住缩放提供给模型的所有特征）：

In [62]:
```
custom_cmap = mpl.colors.ListedColormap(['#fafab0', '#9898ff', '#a0faa0'])

x0, x1 = np.meshgrid(np.linspace(0, 8, 500).reshape(-1, 1),
                     np.linspace(0, 3.5, 200).reshape(-1, 1))
X_new = np.c_[x0.ravel(), x1.ravel()]
X_new = (X_new - mean) / std
X_new_with_bias = np.c_[np.ones(len(X_new)), X_new]

logits = X_new_with_bias @ Theta
Y_proba = softmax(logits)
y_predict = Y_proba.argmax(axis=1)

zz1 = Y_proba[:, 1].reshape(x0.shape)
zz = y_predict.reshape(x0.shape)

plt.figure(figsize=(10, 4))
plt.plot(X[y == 2, 0], X[y == 2, 1], "g^", label="Iris virginica")
plt.plot(X[y == 1, 0], X[y == 1, 1], "bs", label="Iris versicolor")
plt.plot(X[y == 0, 0], X[y == 0, 1], "yo", label="Iris setosa")

plt.contourf(x0, x1, zz, cmap=custom_cmap)
contour = plt.contour(x0, x1, zz1, cmap="hot")
plt.clabel(contour, inline=1)
plt.xlabel("Petal length")
plt.ylabel("Petal width")
plt.legend(loc="upper left")
plt.axis([0, 7, 0, 3.5])
plt.grid()
plt.show()
```



现在让我们测量最终模型在测试集上的准确性：

In [63]:
```
logits = X_test @ Theta
```

```
Y_proba = softmax(logits)
y_predict = Y_proba.argmax(axis=1)

accuracy_score = (y_predict == y_test).mean()
accuracy_score
```

Out[63]:
```
0.9666666666666667
```

好吧，我们在测试集上获得了更好的性能。 这种可变性可能是由于数据集的大小非常小：根据您对训练集、验证集和测试集的采样方式，您可能会得到截然不同的结果。 尝试更改随机种子并再次运行代码几次，您会发现结果会有所不同。