

1 线性回归

模块一

一、代码填空

请补全下列代码，完成数据的标准化处理和线性回归模型的调用。

Python

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

# 1. 数据归一化处理
scaler = StandardScaler()
# [空 1] 注意：为了遵循机器学习的数据划分原则，这里只能传入训练集
scaler.fit(train)
train = scaler.transform(train)
test = scaler.transform(test)

# 2. 训练与预测
linreg = LinearRegression()
# 训练模型
linreg.fit(x_train, y_train)
# [空 2] 在测试集上生成预测结果
y_pred = linreg.predict(x_test)
```

二、简答题

题目：代码中最后计算了 RMSE（均方根误差）。请结合线性回归的任务目标，解释为什么通常使用 RMSE 或 MSE 作为损失函数？如果 RMSE 的值为 0 代表什么含义？

模块二：数学原理篇（矩阵运算）

一、代码填空

请补全下列代码，使用 Numpy 矩阵运算实现线性回归的闭式解（Closed-form Solution）。

Python

```
import numpy as np

# [空 1] 为了引入截距项(bias)，需要在 X 矩阵最后拼接一列全为 1 的向量
# 提示：请填入拼接的方向参数
X = np.concatenate([x_train, np.ones((len(x_train), 1))], axis=-1)
```

```
# [空 2] 正规方程公式:  $\theta = (X^T X)^{-1} X^T y$ 
```

```
# 提示: 使用 numpy.linalg 模块计算矩阵的逆
```

```
 $\theta = \text{np.linalg.}^{-1}\text{inv}(X.T @ X) @ X.T @ y_{\text{train}}$ 
```

二、简答题

题目: 上述代码使用了正规方程法直接求解。但在处理极大规模数据集(例如特征维度达到百万级)时, 工业界通常不使用这种方法, 而是改用梯度下降法(SGD)。请从计算复杂度或内存消耗的角度简要解释原因。

避免巨型矩阵求逆

模块三: 核心算法篇 (手写 SGD)

一、代码填空

请补全 SGD 函数中关于批量生成和梯度更新的关键步骤。

Python

```
def batch_generator(x, y, batch_size, shuffle=True):
```

```
# ... (省略循环与索引逻辑) ...
```

```
# [空 1] Python 生成器关键字, 用于按需返回数据
```

```
yield x[start:end], y[start:end]
```

```
def SGD(num_epoch, learning_rate, batch_size):
```

```
# ... (省略初始化) ...
```

```
for i in range(num_epoch):
```

```
    batch_g = batch_generator(X, y_train, batch_size)
```

```
    for x_batch, y_batch in batch_g:
```

```
        # [空 2] 计算梯度
```

```
        # 提示: 根据矩阵求导公式:  $X.T * (X * \theta - y)$ 
```

```
        grad = x_batch.T @ (_____ @ theta - y_batch)
```

```
        # [空 3] 更新参数 theta
```

```
        # 提示: 沿着梯度的反方向移动
```

```
        theta = theta - _____ * grad / len(x_batch)
```

learning-rate.

参考答案

模块一答案:

- 代码: [空 1] fit ; [空 2] predict。
- 简答: RMSE 衡量了预测值与真实值的偏差程度, 对大误差更敏感。RMSE 为 0 代表模型完美拟合了测试数据(通常意味着过拟合或数据异常简单)。

模块二答案:

- 代码: [空 1] -1 (或 1) ; [空 2] inv 。
- 简答: 正规方程需要计算矩阵 $(X.T @ X)$ 的逆, 其计算复杂度约为 $O(n^3)$ 。当特征维度 n 很大时, 计算逆矩阵非常耗时且内存开销巨大, 而梯度下降法通过迭代逼近, 避免了巨型矩阵求逆。

模块三答案:

- 代码: [空 1] yield ; [空 2] x_batch ; [空 3] learning_rate

2 逻辑回归

第一部分: 代码填空题

1. 激活函数与模型预测

逻辑回归的核心是将线性输出映射到 (0, 1) 区间。

Python

逻辑斯谛 (Sigmoid) 函数实现

def logistic(z):

[空 1] 补全 Sigmoid 公式: $1 / (1 + e^{-z})$

return 1 / (1 + np.exp(-z))

预测阶段

[空 2] 计算线性部分: $X * \theta$

z = X_test @ theta

将结果转化为概率值

test_pred = logistic(z)

2. 带 L2 正则化的梯度下降 (GD)

为了防止过拟合, 代码在损失函数和梯度中加入了 L2 正则项。

Python

循环迭代

for i in range(num_steps):

pred = logistic(X @ theta)

[空 3] 计算梯度

原始梯度为 $-X.T * (y - \text{pred})$, 需加上 L2 正则项的导数

提示: $l2_coef * \theta$ 即为正则项导数

grad = -X.T @ (y_train - pred) + l2_coef * theta

[空 4] 更新参数

提示: 沿着梯度反方向移动

theta = theta - learning_rate * grad

计算损失函数 (Log-Loss + L2 Penalty)

[空 5] 补全 L2 正则化在损失函数中的部分

提示: L2 范数的平方除以 2

```
train_loss = ... + l2_coef * np.linalg.norm(theta) ** 2 / 2
```

3. AUC 指标计算逻辑

代码手动实现了一个基于排序和累加的 AUC 计算方法。

Python

```
def auc(y_true, y_pred):
    # [空 6] 对预测概率进行排序
    # 提示: argsort 返回索引, [::-1] 表示降序, 保证概率大的排前面
    idx = np.argsort(y_pred)[::-1]
    y_true = y_true[idx]

    # [空 7] 计算真正例 (TP) 和假正例 (FP) 的累积分布
    # 提示: 使用 cumsum 函数
    tp = np.cumsum(y_true)
    fp = np.cumsum(1 - y_true)

    # ... (计算面积逻辑) ...
    return s
```

4. 决策边界绘制

为了在二维平面画出分类直线 $\theta_0 x_1 + \theta_1 x_2 + \theta_2 = 0$, 我们需要推导 x_2 关于 x_1 的关系。

Python

```
# 直线方程转换: 求 x2 (即 plot_y)
plot_x = np.linspace(-1.1, 1.1, 100)
# [空 8] 根据  $\theta_0 x + \theta_1 y + \theta_2 = 0$  推导 y
plot_y = -(theta[0] * plot_x + theta[2]) / theta[1]
```

5. Sklearn 库调用

使用标准库验证手写代码的结果。

Python

```
from sklearn.linear_model import LogisticRegression
# 初始化模型, 指定优化求解器
lr_clf = LogisticRegression(solver='liblinear')
# [空 9] 拟合模型
lr_clf.fit(x_train, y_train)
# [空 10] 预测类别
y_pred = lr_clf.predict(x_test)
```

第二部分：简答与分析题

1. 关于 L2 正则化 (Regularization)

防止过拟合。

在 GD 函数的梯度计算代码 $\text{grad} = -X.T @ (y_{\text{train}} - \text{pred}) + l2_coef * \theta$ 中，引入 $l2_coef * \theta$ 这一项的目的是什么？如果 $l2_coef$ 设置得非常大（例如 1000），会对模型的决策边界产生什么影响（欠拟合还是过拟合）？

欠拟合。

2. 关于 AUC 与 Accuracy 的区别

代码中同时计算了 `acc` (准确率) 和 `auc`。

(1) 在 `acc` 函数中，我们通常需要设定一个阈值（代码中使用了 `0.5: pred >= 0.5`）。如果我们将阈值提高到 0.9，`acc` 会发生变化吗？

会

(2) `auc` 指标的计算是否依赖于这个 0.5 的阈值？为什么在样本分布不平衡（例如正样本极少）的情况下，我们更倾向于看 AUC 而不是 Accuracy？

不依赖。

3. 关于决策边界的几何意义

在绘制决策边界的代码中，使用了公式 $-(\theta[0] * \text{plot_x} + \theta[2]) / \theta[1]$ 。

请解释为什么这里涉及 $\theta[2]$ ？在数据预处理阶段，哪一行代码为 $\theta[2]$ 的存在提供了数据基础？

参考答案

第一部分：代码填空

1. `exp` — 指数函数。
2. `theta`
3. `theta` — 正则化项的导数。
4. `learning_rate`
5. `2` — L2 Loss 定义通常包含系数 1/2 以便求导抵消。
6. `[:, -1]` — AUC 计算需按预测概率从大到小排序。
7. `cumsum`
8. `theta[1]` — 解方程移项，分母为 x_2 的系数。
9. `fit` — Sklearn 训练方法。
10. `predict`

第二部分：简答题

1. L2 正则化的作用：

- 目的：惩罚过大的参数权重，限制模型复杂度，防止过拟合。

- **大系数影响**：如果 $l2_coef$ 极大，模型会倾向于让所有参数 θ 接近于 0，导致模型极其简单，从而发生**欠拟合**（决策边界可能变成一条忽略数据的直线）。

2. AUC vs Accuracy:

- (1) **会变化**。Accuracy 强依赖于阈值，阈值改变，预测的类别（0 或 1）就会变，准确率随之改变。
- (2) **不依赖**。AUC 衡量的是模型对正负样本的排序能力（即正样本的预测分值高于负样本的概率），它遍历了所有可能的阈值。在样本不平衡时，Accuracy 容易产生误导（例如全猜 0 也有高准确率），而 AUC 能更真实反映模型区分两类样本的能力。

3. 截距项与决策边界:

- **几何意义**： $\theta_0 x_1 + \theta_1 x_2 + \theta_2 \cdot 1 = 0$ 中的常数项（截距/偏置）。如果没有它，决策直线将被强制经过原点 (0,0)，限制了模型的拟合能力。
- **代码对应**：对应数据预处理中的 $X = \text{np.concatenate}([x_train, \text{np.ones(...)}, \text{axis}=1])$ 。这一行在特征矩阵最后添加了一列全为 1 的数据，使得 θ 的最后一个分量成为偏置项。

3 KNN (K 近邻)

模块一：KNN 算法原理与手动实现

考察点：欧氏距离计算、排序逻辑、多数表决机制。

一、代码填空

请阅读 KNN 类中的方法实现，补全下列代码。

Python

```
def distance(a, b):
    # [空 1] 计算两个向量之间的欧氏距离
    # 公式:  $\sqrt{\sum((a - b)^2)}$ 
    return np.sqrt(np.sum(np.__(a - b)))

class KNN:
    def get_knn_indices(self, x):
        # 计算已知样本的距离
        dis = list(map(lambda a: distance(a, x), self.x_train))
        # [空 2] 对距离进行排序，并返回排序后的索引 (indices)
        # 提示：我们需要的是下标，不是排序后的距离值
        knn_indices = np.__(dis)
        # 取最近的 K 个
        knn_indices = knn_indices[:self.k]
        return knn_indices
```

```
def get_label(self, x):
    knn_indices = self.get_knn_indices(x)
    label_statistic = np.zeros(shape=[self.label_num])
    for index in knn_indices:
        label = int(self.y_train[index])
        label_statistic[label] += 1
    # [空 3] 返回数量最多的类别
    # 提示：找出数组中最大值的索引
    return np._____ (label_statistic)
```

二、简答题

argmax

题目：在 KNN 类的 fit 方法中，代码仅仅执行了 `self.x_train = x_train` 和 `self.y_train = y_train`，并没有进行任何复杂的数学运算或参数迭代。

(1) 这种特性被称为什么类型的学习方法？

懒惰学习。

(2) 这种方法在预测阶段 (Predict) 的计算复杂度通常是高还是低？为什么？

高 复杂度与数据集大小成正比。

模块二：Scikit-learn 应用与可视化 (高斯数据集)

考察点：网格生成、决策边界绘制、K 值对模型的影响。

一、代码填空

请补全下列代码，用于生成可视化的背景网格并进行预测。

Python

```
from sklearn.neighbors import KNeighborsClassifier

# 构造网格点用于画图
# xx, yy 分别代表网格的横纵坐标矩阵
xx, yy = np.meshgrid(np.arange(x_min, x_max, step), np.arange(y_min, y_max, step))

# [空 4] 将 xx 和 yy 拉平并拼接，构造出所有网格点的坐标矩阵
# 提示：reshape(-1, 1) 将数组转为列向量，axis=1 表示横向拼接
grid_data = np.concatenate([xx.reshape(-1, 1), yy.reshape(-1, 1)], axis=_____)

# 循环测试不同的 K 值
for i, k in enumerate([1, 3, 10]):
    # [空 5] 初始化 KNN 分类器
    # 提示：参数名通常为 n_neighbors
    knn = KNeighborsClassifier(_____=k)
    knn.fit(x_train, y_train)
```

n_neighbors

```
# 预测网格点的类别，用于绘制背景颜色
```

```
z = knn.predict(grid_data)
```

二、简答题

题目：观察代码中提到的可视化部分。当 $K=1$ 时，决策边界通常非常曲折且破碎（容易受噪声影响）；当 $K=10$ 时，决策边界变得更加平滑。

请用机器学习中的“过拟合 (Overfitting)”和“欠拟合 (Underfitting)”概念，简要分析 K 值过小（如 $K=1$ ）和 K 值过大分别容易导致什么问题？

参考答案

模块一答案

- 代码填空：
 - [空 1] square (平方)
 - [空 2] argsort (参数排序返回索引)
 - [空 3] argmax (返回最大值的索引，即票数最多的类别)
- 简答题：
 - (1) **懒惰学习 (Lazy Learning)**。模型在训练阶段不进行计算，只是把数据存储起来。
 - (2) **高**。因为预测时需要计算待预测点与所有训练样本的距离，计算量与训练集大小成正比。

模块二答案

- 代码填空：
 - [空 4] 1 (axis=1, 列拼接)
 - [空 5] n_neighbors (sklearn 参数名)
- 简答题：
 - **K 值过小 ($K=1$)**：模型太复杂，容易捕捉到噪声，导致**过拟合**。
 - **K 值过大**：模型过于简单，忽略了局部特征，决策边界过于平滑，容易导致**欠拟合**。

4 多层感知机 (MLP)

模块一：神经网络底层原理 (NumPy 手写实现)

考察点：链式法则、反向传播梯度计算、激活函数导数。

一、代码填空

1. 全连接层 (Linear Layer) 的反向传播

在 Linear 类的 backward 函数中，我们需要根据上一层传回的梯度 grad 计算对权重 W、偏置 b 以及输入 x 的梯度。

Python

```
def backward(self, grad):
    # grad 的维度为 (batch_size, num_out)

    # [空 1] 计算对权重 W 的梯度
    # 提示：利用链式法则，公式涉及输入 x 的转置与 grad 的矩阵乘法
    self.grad_W = _____ @ grad / grad.shape[0]
    self.x.T

    if self.use_bias:
        # [空 2] 计算对偏置 b 的梯度
        # 提示：偏置在 batch 维度上是共享的，需要进行聚合（求均值或求和）
        self.grad_b = _____(grad, axis=0, keepdims=True)
        np.mean

    # [空 3] 计算传递给前一周的梯度（即对输入 x 的梯度）
    # 提示：需要将 grad 映射回输入维度
    grad = grad @ _____
    self.W.T
    return grad
```

2. Sigmoid 激活函数的导数

Sigmoid 函数 $f(x) = \frac{1}{1+e^{-x}}$ 有一个特殊的导数性质，可以用函数输出值 y 来表示导数。

Python

```
class Sigmoid(Layer):
    def forward(self, x):
        self.y = 1 / (1 + np.exp(-x))
        return self.y

    def backward(self, grad):
        # [空 4] 补全 Sigmoid 的导数计算
        # 提示：f'(x) = f(x) * (1 - f(x))
        return grad * _____
        self.y * (1 - self.y)
```

3. MLP 的反向传播流程

在多层网络中，误差需要从输出层向输入层逐层传递。

Python

```
def backward(self, grad):
    # [空 5] 补全循环逻辑
```

提示：反向传播需要从最后一层往前遍历

```
for layer in _____(self.layers):  
    grad = layer.backward(grad)
```

reversed(self.layers).

二、简答题

1. 关于参数初始化

在 Linear 类的 init 方法中，代码使用了 `np.random.normal` 来初始化权重 W，而不是将其全部初始化为 0。

请问：如果将神经网络的所有权重 W 都初始化为 0，会发生什么后果？这也被称作“对称性难题” (Symmetry Breaking Problem)，请简要解释。

2. 关于激活函数 ReLU

模型会失去学习复杂特征的能力

代码中实现了 ReLU 激活函数，其 backward 方法如下：`return grad * (self.x >= 0)`。

请问：当输入 $x < 0$ 时，梯度会变成多少？这种情况在深层网络训练中可能导致什么问题（提示：Dead ReLU 现象）？

参数不再更新 神经元永久死亡。

模块二：模型训练与 PyTorch 框架

考察点：交叉熵损失、PyTorch 核心训练步骤 (Zero_grad, Backward, Step)。

一、代码填空

1. 交叉熵损失 (Cross-Entropy Loss)

对于二分类问题，我们使用二元交叉熵损失函数。

Python

计算交叉熵损失

y 为真实标签，y_pred 为预测概率，eps 为防止 $\log(0)$ 的微小值

[空 6] 补全损失函数公式

提示： $-y \cdot \log(y_hat) - (1-y) \cdot \log(1-y_hat)$

train_loss = np.sum(-y * np.log(y_pred + eps) _____)

$np.sum(-y \cdot \log$

2. PyTorch 训练循环

使用 PyTorch 训练模型时，优化器和反向传播的标准步骤是固定的。

$(y_pred + ep.$

Python

计算 MLP 的预测

y_pred = mlp(x)

计算损失

```
train_loss = torch.mean(...)
```

```
# [空 7] 清空优化器中累积的梯度
```

opt.zero_grad

```
# [空 8] 自动反向传播，计算梯度
```

train_loss.backward()

```
# [空 9] 优化器更新参数
```

opt.step()

二、简答题

1. 批量训练 (Batch Training) 代码中使用了 `batch_size = 128`。在 `Linear.backward` 中，计算偏置梯度时使用了 `np.mean(grad, axis=0)`。请解释：为什么对偏置 `b` 的梯度需要求均值（或求和）？这与批量数据输入有什么关系？

偏置b是一个向量。

参考答案

模块一：底层原理

- 代码填空
 - [空 1] `self.x.T` (或者转置操作) ——梯度的推导核心。
 - [空 2] `np.mean` (或 `np.sum`，取决于定义，代码中是 `mean`) ——聚合 Batch 维度的梯度。
 - [空 3] `self.W.T` ——误差反向传给上一层。
 - [空 4] `self.y * (1 - self.y)` ——Sigmoid 导数的标准形式。
 - [空 5] `reversed(self.layers)` ——反向传播必须从后往前。
- 简答题
 1. 全 0 初始化后果：如果权重全为 0，同一层的所有神经元对于相同的输入会计算出相同的输出，反向传播时它们的梯度也完全相同。这导致它们在每次更新后依然保持相同，实际上整个层退化为单个神经元，模型失去了学习复杂特征的能力。
 2. Dead ReLU：当 $x < 0$ 时，ReLU 的导数为 0。这意味着该神经元的参数将停止更新。如果学习率过大导致权重剧烈变化，可能导致某个神经元对所有输入都输出负值，该神经元就会“永久死亡”，不再对网络起作用。

模块二：PyTorch 与 训练

- 代码填空
 - [空 6] `-(1 - y) * np.log(1 - y_pred + eps)` ——二元交叉熵的第二部分。
 - [空 7] `opt.zero_grad()` ——必须清空梯度，否则会累加。
 - [空 8] `train_loss.backward()` ——构建计算图并回传。
 - [空 9] `opt.step()` ——

执行梯度下降更新。

- 简答题

1. **偏置梯度聚合**: 在模型中, 偏置 b 是一个向量 (维度为 num_out), 它被广播 (Broadcasting) 加到整个 Batch 的每一个样本上。因此, 在反向传播时, 我们需要将 Batch 中所有样本对该偏置的贡献 (梯度) 累加或求平均, 以得到该偏置最终的梯度更新值。

5 支持向量机 (SVM)

模块一: SMO 算法核心实现

考察点: SMO 算法的更新公式、KKT 条件约束、拉格朗日乘子裁剪。

一、代码填空

1. 参数更新步长 (Eta) 计算

在 SMO 算法中, 我们需要计算目标函数对 α_j 的二阶导数 (或者说是更新的“步长”分母)。

Python

```
# 预先计算所有向量的两两内积 K[i, j]
# [空 1] 计算 eta ( $\eta$ )
# 公式推导:  $\eta = K_{ii} + K_{jj} - 2 * K_{ij}$ 
eta = K[j, j] + K[i, i] - 2 * K[i, j]
```

2. 拉格朗日乘子的裁剪 (Clipping)

由于约束条件 $0 \leq \alpha_i \leq C$, 更新后的 α 必须被限制在这个范围内。

Python

```
# [空 2] 使用 numpy 的函数将 alpha_i 限制在 [lower, upper] 区间内
alpha_i = np.clip(alpha_i, lower, upper)

# 更新完 alpha_i 后, 根据约束条件  $\sum(y * \alpha) = 0$  更新 alpha_j
alpha_j = (zeta - y[i] * alpha_i) / y[j]
```

3. 支持向量的识别

训练完成后, 我们需要找出支持向量 (Support Vectors) 来计算 w 和 b 。

Python

```
# [空 3] 设定一个极小的阈值来判断 alpha 是否大于 0
```

根据 KKT 条件, $\alpha > 0$ 的样本即为支持向量

sup_idx = alpha > 0

计算权重 w (仅针对线性核)

w = np.sum((alpha[sup_idx] * y[sup_idx]).reshape(-1, 1) * x[sup_idx], axis=0)

二、简答题 (20 分)

题目: 在 SMO 函数的迭代循环中, 代码每次随机选择 两个 参数 α_i 和 α_j 进行更新。 请问: 为什么不能像普通梯度下降那样, 每次只更新 一个 参数 α_i ? 这与 SVM 的什么约束条件有关?

会破坏等式约束. $\sum_i \alpha_i \cdot y_i = 0$.

模块二: 核函数 (Kernels) 与非线性分类

考察点: RBF 核公式、核函数的作用、代码中的超参数设置。

一、代码填空

1. 径向基核函数 (RBF Kernel)

RBF 核 (高斯核) 是处理非线性数据最常用的核函数。

Python

```
def rbf_kernel(sigma):  
    def k(x, y):  
        # [空 4] 补全高斯核公式  
        # 公式:  $\exp(-||x-y||^2 / (2 * \sigma^2))$   
        # 提示: np.inner(x-y, x-y) 等价于  $||x-y||^2$   
        return np.exp(-np.inner(x - y, x - y) / (2.0 * sigma ** 2))  
    return k
```

2. Sklearn 库调用

使用 sklearn.svm.SVC 处理螺旋线数据。

Python

```
from sklearn.svm import SVC  
  
# [空 5] 定义模型  
# gamma 参数对应公式中的  $1/(2*\sigma^2)$ , 控制高斯分布的宽度  
model = SVC(kernel='_____', gamma=50, tol=1e-6)  
model.fit(x, y)
```

二、简答题

rbf

1. 关于惩罚系数 C

在处理线性可分数据集（第一部分代码）时，代码设置了 $C = 1e8$ 。请问：设置如此巨大的 C 值意味着我们希望模型是“硬间隔” (Hard Margin) 还是“软间隔” (Soft Margin)? 在这种情况下，模型对误分类点的容忍度是高还是低?

硬间隔 低.

2. 关于 RBF 核的 Sigma

在手动实现的 `rbf_kernel` 中有一个参数 `sigma`，而在 `sklearn` 中对应的参数是 `gamma`。请根据代码中的可视化结果推测：如果我们将 `sigma` 设置得非常小（或者 `gamma` 设置得非常大），模型的决策边界会变得平滑还是变得破碎（过拟合）？为什么？

会导致模型过拟合.

参考答案

模块一：SMO 算法

• 代码填空

- [空 1] 2 —— 公式 $\eta = K_{11} + K_{22} - 2K_{12}$
- [空 2] clip —— `np.clip` 用于数值截断。
- [空 3] $1e-5$ (或 0) —— 浮点数比较通常需要一个微小的阈值 `epsilon`。

• 简答题

原因：SVM 的对偶问题有一个等式约束： $\sum_{i=1}^m \alpha_i y_i = 0$ 。如果只改变一个 α_i ，这个等式约束就会被破坏。因此，SMO 算法每次必须同时更新两个参数，以保持总和约束不变（一个增加，另一个相应减少/变化）。

模块二：核函数与应用

• 代码填空

- [空 4] exp —— 高斯函数基于指数运算。
- [空 5] rbf —— 指定使用径向基核函数。

• 简答题

1. 硬间隔 (Hard Margin): C 极大意味着对误分类的惩罚无限大。模型会尽全力保证所有训练样本都被正确分类，不允许任何样本进入间隔内部（即不允许松弛变量 $\xi > 0$ ）。容忍度极低。
2. 变得破碎（过拟合）：sigma 越小（gamma 越大），高斯分布越“尖”，每个样本的影响范围越窄。模型会倾向于在每个样本周围形成独立的“孤岛”状决策边界，导致严重的过拟合。

6 集成学习算法原理与实战

第一部分：代码填空题

1. 随机森林 (Random Forest) 的构建与采样

在手写 `RandomForest` 类中，我们需要对样本进行有放回的随机采样（`Bootstrap`），并利用未被采样的样本进行 `OOB` 验证。

Python

```
# [空 1] Bootstrap 采样：随机生成索引，允许重复
```

```
idx = np.random.randint(0, n_samples, n_samples)
```

```
# [空 2] 找出 OOB (Out-of-Bag) 样本
```

```
# 提示：统计索引出现次数，为 0 表示未被抽中
```

```
unsampled_mask = np.bincount(idx, minlength=n_samples) == 0
```

```
# 训练当前决策树
```

```
tree.fit(X[idx], y[idx])
```

```
# [空 3] 随机森林与 Bagging 的区别
```

```
# 在初始化 DecisionTreeClassifier 时，Random Forest 通常设置 max_features='sqrt'
```

```
# 而 Bagging 如果退化为普通 Bagging，max_features 应设置为 None
```

```
self.trees = [DTC(max_features=max_features) for _ in range(n_trees)]
```

2. 随机森林的预测逻辑 (Soft Voting)

为了获得更稳健的结果，代码采用了软投票机制。

Python

```
def predict_proba(self, X):
```

```
    # [空 4] 将所有树预测的概率取平均
```

```
    # 提示：axis=0 表示在树的数量维度上求均值
```

```
    ensemble = np.mean([tree.predict_proba(X) for tree in self.trees], axis=0)
```

```
    return ensemble
```

3. Stacking 集成学习

`Stacking` 的核心难点在于如何生成元特征（`Meta-features`）并防止数据泄露，代码中使用了 `K-Fold` 交叉验证。

Python

```
# 遍历基分类器
```

```
for classifier in self.classifiers:
```

```
    # ...
```

```
    # [空 5] 使用 K-Fold 切分数据，防止直接预测导致的过拟合
```

```
    for train_idx, test_idx in self.kf.split(X):
```

```
        # [空 6] 克隆基模型，确保每一折都是一个新的模型实例
```

```
        clf = clone(classifier)
```

```
        clf.fit(X[train_idx], y[train_idx])
```

```
        # 在验证集上进行预测，存入 predict_proba
```

```
        predict_proba[test_idx] = clf.predict_proba(X[test_idx])
```

4. XGBoost 回归

在回归任务中，XGBoost 提供了线性回归没有的正则化参数。

Python

```
xgbr = xgb.XGBRegressor(  
    n_estimators=100,  
    # [空 7] L2 正则化项参数，用于防止过拟合  
    reg_lambda=0.1,  
    objective='reg:squarederror'  
)
```

5. 数据集构建

为了测试特征筛选能力，我们构建了包含噪音的数据集。

Python

```
X, y = make_classification(  
    n_samples=1000,  
    n_features=16,  
    # [空 8] 设置标签翻转比例，模拟标签噪声，增加分类难度  
    flip_y=0.1,  
    random_state=0  
)
```

第二部分：简答题

1. 随机森林 vs Bagging (Random Forest vs Bagging)

实验中对比了 `max_features=None` (Bagging) 和 `max_features='sqrt'` (Random Forest)。

请结合代码注释解释：

引入更多随机性。

(1) 为什么 Random Forest 在划分节点时只考虑特征的一个子集 (sqrt) ?

(2) 从偏差-方差 (Bias-Variance) 的角度看，这样做带来了什么好处？

提升泛化能力。

2. OOB (Out-of-Bag) 验证机制

代码中实现了 `self.oob_score`。

(1) 请解释什么是 OOB 样本？（它是如何产生的？）

未被选中去训练的样本

(2) 使用 OOB 分数作为评估指标有什么优势？（相比于单独划分验证集）

3. Stacking 中的“数据泄露”问题

自带验证集效果。

在 `StackingClassifier` 的 `fit` 方法中，代码注释提到：“为什么不能直接 `classifier.fit(X,y)` 然后 `classifier.predict(X)`？”

请详细解释：如果直接在整个训练集上训练基模型并预测，会对元学习器（`Meta Classifier`）的训练产生什么负面影响？

参考答案

第一部分：代码填空

1. **0**（`np.bincount` 统计次数为 0 即未被采样）
2. **None**（`None` 代表使用所有特征，即标准 `Bagging`）
3. **None**（同上，或者代码中 `max_features` 变量的值）—— 注：根据上下文，此处考察 `Bagging` 的参数设置，应填 `None`。
4. **0**（`axis=0` 对所有树的结果求平均）
5. **clone**（`sklearn` 的克隆函数，用于复制模型配置）
6. **clone**（或者 `split`）—— 注：根据题意空 6 对应代码 `clf = clone(classifier)`
7. **0.1**（代码中设置的具体数值，或者填 `reg_lambda` 对应的变量名）
8. **0.1**（代码中 `flip_y` 的值）

第二部分：简答题

1. RF vs Bagging

- (1) **原因**：为了引入更多的随机性，降低树与树之间的相关性（**De-correlation**）。如果每次都用所有特征，强特征总是会被选为分裂点，导致生成的树过于相似。
- (2) **好处**：虽然单棵树的偏差（**Bias**）可能微弱增加，但树之间的独立性显著降低了整体模型的方差（**Variance**），从而提高了泛化能力。

2. OOB 验证机制

- (1) **产生**：在 `Bootstrap` 有放回采样过程中，约有 36.8% ($1/e$) 的样本未被选中，这些未被当前树用来训练的样本就是该树的 **OOB** 样本。
- (2) **优势**：自带验证集效果。不需要将数据集切分为训练集和验证集，从而可以使用所有数据进行训练，同时又能评估模型性能，特别适合小样本数据。

3. Stacking 的数据泄露

- **解释**：如果直接在全量数据上训练基模型并预测，基模型在训练数据上的表现通常会非常好（甚至过拟合）。元学习器会发现基模型的预测与真实标签几乎完全一致，从而赋予基模型过高的权重。然而，这种“完美预测”在测试集上是无法复现的，导致整体 `Stacking` 模型在面对新数据时性能下降（过拟合）。使用 `K-Fold` 确保了元模型的输入特征是基模型在“未见过”的数据上的预测结果。

7 概率图模型与应用

模块一：朴素贝叶斯文本分类 (Naive Bayes)

考察点：拉普拉斯平滑、对数概率计算、贝叶斯公式实现。

一、代码填空

请补全下列代码，实现基于多项式模型的朴素贝叶斯分类器。

1. 词频统计与平滑

在计算单词的条件概率 $P(w|c)$ 时，为了防止零概率问题，代码引入了 α 进行平滑（Add-one Smoothing）。

Python

```
# alpha = 1.0 (拉普拉斯平滑)
# 统计单词频率，初始化为 alpha
log_voc_freq = np.zeros((20, len(train_data.feature_names))) + alpha
# ... (省略循环统计过程) ...
# [空 1] 计算对数条件概率
# 提示：需将频数除以该类别下的总词数 voc_cnt
log_voc_freq = np.log(log_voc_freq / _____)
```

Voc - cnt

2. 预测函数

在测试阶段，根据贝叶斯公式 $P(c|d) \propto P(c) \prod P(w_i|c)$ 计算后验概率。

Python

```
def test_news(news):
    rows, cols = news.nonzero()
    # [空 2] 初始化对数后验概率为先验概率
    log_post = np.copy(_____)
    for row, voc in zip(rows, cols):
        # [空 3] 加上文档中每个单词的对数条件概率
        # 提示：利用 log_voc_freq 矩阵查询
        log_post += _____[:, voc]
    # 返回概率最大的类别索引
    return np.argmax(log_post)
```

log-cat-freq

log-voc-freq

二、简答题

题目：在计算概率时，代码没有直接使用原始概率值（例如 0.001），而是全程使用了 np.log 取对数（例如 log_cat_freq 和 log_voc_freq ）。

请问：这样做在工程实现上有什么主要好处？如果直接连乘原始概率会发生什么问题？

模块二：马尔可夫随机场图像去噪 (MRF/ICM)

考察点：能量函数定义、邻域交互、ICM 迭代优化。

一、代码填空

请补全下列代码，实现基于能量最小化的图像去噪算法。

1. 局部能量函数计算

能量函数包含两部分：与观测图像的一致性（Data Term）和像素间的平滑性（Smoothness Term）。

Python

```
def compute_energy(X, Y, i, j, alpha, beta):
    # Data Term: 观测约束
    energy = -beta * X[i][j] * Y[i][j]
```

```
# Smoothness Term: 邻域约束
# [空 4] 如果上方有像素，减去与上方像素的相互作用能
if i > 0:
```

```
    energy -= alpha * X[i][j] *  $X[i-1][j]$ 
```

```
# ... (省略其他方向) ...
```

```
return energy
```

2. 迭代条件模式 (ICM) 更新

在每一轮迭代中，对每个像素进行贪心更新。

Python

逐像素优化

```
for i in range(X.shape[0]):
```

```
    for j in range(X.shape[1]):
```

```
        # 分别计算当前像素取 1 (白) 和 -1 (黑) 时的能量
```

```
        X[i, j] = 1
```

```
        pos_energy = compute_energy(X, noisy_img, i, j, alpha, beta)
```

```
        X[i, j] = -1
```

```
        neg_energy = compute_energy(X, noisy_img, i, j, alpha, beta)
```

```
# [空 5] 贪心策略：选择使局部能量更小的值作为当前像素的新值
```

```
X[i, j] = 1 if _____ < _____ else -1
```

pos_energy neg_energy

二、简答题

1. 超参数的作用

在能量函数中定义了两个超参数 α 和 β ：

○ $energy = -\beta * X[i][j] * Y[i][j] - \alpha * \sum X[i][j] * X_neighbors$

请解释：

(1) α 变大时，去噪后的图像会变得更平滑还是更破碎？

(2) β 变大时，去噪后的图像会更接近原噪点图还是更偏离？

2. 算法收敛性

代码中采用的这种“逐像素遍历并取能量最小值”的更新策略，在优化算法中通常被称为什么？（提示：ICM 或 坐标下降）。这种方法能保证找到全局最优解

（Global Optimum）吗？为什么？

不能 可能会陷入局部最优。

参考答案

模块一：朴素贝叶斯

• 代码填空

- [空 1] `voc_cnt` —— 归一化，计算概率。
- [空 2] `log_cat_freq` —— 先验概率项。
- [空 3] `log_voc_freq` —— 似然概率项累加。

• 简答题

- 好处：防止数值下溢 (Underflow)。

- **原因：**由于特征（单词）数量巨大，多个小于 1 的概率直接相乘会导致结果迅速趋近于计算机浮点数能表示的 0，导致精度丢失。取对数后将连乘变为求和，避免了此问题。

模块二：图像去噪

- **代码填空**
 - [空 4] $X[i-1][j]$ —— 上方邻居像素。
 - [空 5] $\text{pos_energy} < \text{neg_energy}$ —— 选择能量较低的状态。
- **简答题**
 1. **超参数：**
 - (1) **更平滑。** α 控制邻域一致性，值越大，相邻像素越倾向于取相同颜色，图像色块越明显。
 - (2) **更接近原噪点图。** β 控制观测一致性，值越大，去噪图被强制要求忠实于输入（带噪）图像，去噪效果变弱。
 2. **收敛性：**
 - 这种方法称为 **ICM (Iterated Conditional Modes)** 或 **坐标下降法 (Coordinate Descent)**。
 - **不能保证找到全局最优解。**它是一种贪心算法，极易陷入**局部最优 (Local Optima)**，结果很大程度上依赖于初始状态。