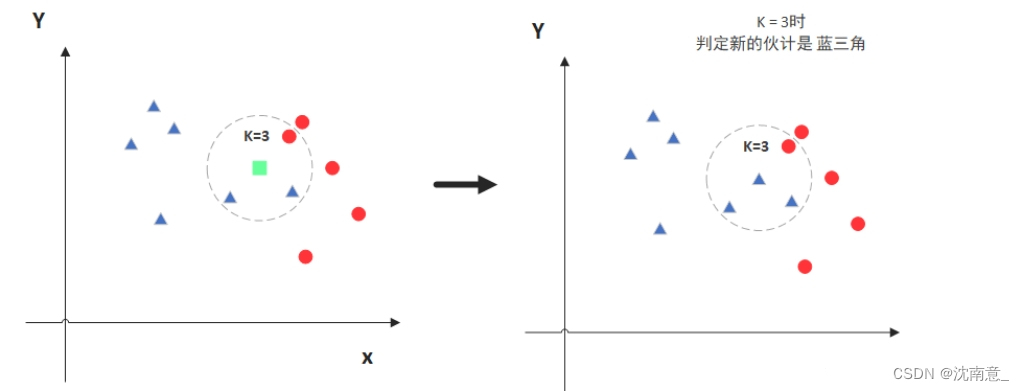
# KNN算法

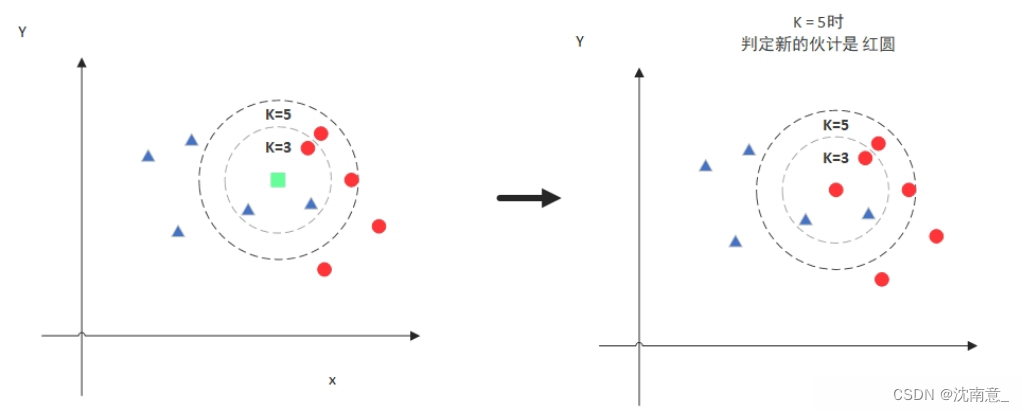
K近邻算法（K-Nearest Neighbors，KNN）是一种简单而直观的非参数监督学习算法，常用于分类和回归任务。KNN在分类问题中尤其常用，它的基本思想是基于"相似样本的类别相似"的假设。

### KNN算法的思想与原理

由KNN算法的名字我们就可以窥探到该算法的核心思想——k个最近的邻居。KNN的机制极为简单：给定一个训练集进行训练，针对一个新的测试集，分别计算与每一个训练集数据的距离，得到k个最近的数据进行预测。一般情况下，都会使用“投票法”来进行预测，例如一个二分类问题，在k范围内，类别A有个，而类别B有个，那么对这个新的数据就认为其属于类别B.



在上图的例子中，绿色方块即为需要预测的新数据，当k=3时，判断新数据为蓝色三角。但是当k=5时，又会将新的数据认作红色的圆：



由此可知，在KNN算法中，k起着决定性的作用，在调整参数时对k要进行慎重的选择。

### 二、KNN算法的计算核心

在KNN算法中，距离度量是关键步骤。以下是三种最常见的距离度量方式的数学推导：**欧几里得距离**、**曼哈顿距离**、和**余弦相似度**。

#### 1. 欧几里得距离

欧几里得距离是两个点在空间中直线距离的度量方式，计算公式为：

其中，x 和 z是两个样本向量，n是特征的维数。对于高维数据，如图像，可以将每个像素展平成一个特征向量，再使用欧几里得距离计算样本间的距离。

#### 2. 曼哈顿距离示例

曼哈顿距离的计算过程如下：

曼哈顿距离强调每个特征的绝对差距，适用于某些特征对整体距离影响较大的情况。

#### 3. 余弦相似度

余弦相似度是用向量夹角余弦值来衡量样本间的相似性，适合归一化数据。其计算公式为：

余弦相似度越接近1，表示两个向量的方向越相似。通常将余弦相似度转为距离形式：1-.

### 以甲骨文数据集处理后为例进行推导计算

#### 1.数据集的准备

现在我们构建一个只包含六张图片的训练集以及一张图片作为测试，六张图片大小分别为36\*93、69\*143、44\*80、56\*152、59\*118、65\*105。为了方便我们实例推算，我们将图片平均分成九个同样大小的九宫格，然后对每一个格子中图片的特征值求平均值后归一化。可以使用一下代码完成这个任务：

def preprocess\_image(image\_path):

# 打开图像并转换为灰度模式 ('L' 模式)，即单通道

img = Image.open(image\_path).convert('L')

# 将图像转换为NumPy数组，以便进行矩阵操作

img\_array = np.array(img)

# 确定每个网格单元的高度和宽度

# 将图像的高度和宽度分别整除3，这样可以将图像分成3x3的九宫格

grid\_height, grid\_width = img\_array.shape[0] // 3, img\_array.shape[1] // 3

# 用于存储每个网格的特征值的列表

feature\_vector = []

# 使用嵌套循环遍历3x3的网格

for i in range(3):

for j in range(3):

# 确定当前网格单元的起始和结束像素坐标

# 行坐标：从 i \* grid\_height 开始，到 (i + 1) \* grid\_height 结束

# 列坐标：从 j \* grid\_width 开始，到 (j + 1) \* grid\_width 结束

start\_i, end\_i = i \* grid\_height, (i + 1) \* grid\_height

start\_j, end\_j = j \* grid\_width, (j + 1) \* grid\_width

# 提取当前网格单元的像素数据，形成一个小矩阵

patch = img\_array[start\_i:end\_i, start\_j:end\_j]

# 计算该网格的平均像素值

# 将平均值归一化为0到1之间的浮点数，并保留两位小数

avg\_value = round(np.mean(patch) / 255.0, 2)

# 将计算的特征值添加到特征向量列表中

feature\_vector.append(avg\_value)

# 返回该图像的九个特征值（每个网格单元的归一化平均像素值）

return feature\_vector

以下是作为训练集的6张图片，共有三个类别，分别为“勿”、“牛”、“羊”三种标签，每张种类共计两张图片，对这些图片调整大小以后进行核处理，我们可以得到以下数据：

Filename,Feature\_1,Feature\_2,Feature\_3,Feature\_4,Feature\_5,Feature\_6,Feature\_7,Feature\_8,Feature\_9,Label

\_日1.png,0.72,0.45,0.79,0.66,0.75,0.84,0.84,0.60,0.80 ,日

\_日2.png,0.68,0.65,0.66,0.68,0.69,0.79,0.63,0.64,0.61,日

\_牛1.png,0.78,0.98,0.76,0.69,0.41,0.88,0.93,0.58,1.0,牛

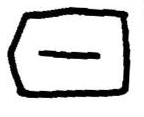
\_牛2.png,0.74,0.77,0.75,0.85,0.41,0.92,0.88,0.63,0.93,牛

\_羊1.png,0.82,0.66,0.73,0.91,0.55,0.95,1.0,0.78,1.0,羊

\_羊2.png,0.74,0.69,0.74,0.88,0.5,0.89,0.82,0.8,0.92,羊

从左到右边依次为文件名称、九个提取出的特征以及标签。

对应的六张图片分别如下：

p8w7ujqanz_h01884_129_237_71_44 _牛1 _牛2  _羊1

经过KNN训练以后，我们所保存的模型文件中就会将这些值保存下来，接着我们输入需要预测的图像进行同样的图像处理操作，可以得到以下数据：

aoaipuy20h_h01216_180_347_49_63

需要预测的图片，实际标签为“羊”

Feature\_1,Feature\_2,Feature\_3,Feature\_4,Feature\_5,Feature\_6,Feature\_7,Feature\_8,Feature\_9

0.72,0.58,0.63,0.93,0.36,0.91,1.0,0.69,1.0

在得到了每张图片经过提取之后的特征值以后，我们就可以利用KNN算法来进行计算了（假设k = 3）。

#### 2.使用欧几里得距离进行计算

我们知晓欧几里得距离的计算公式如下：

在本例中n = 9，针对所有训练集都与预测的图片都进行一次计算：

与\_日1.png进行计算：

=

综上，我们可得式：

与\_日1.png计算相似，我们利用这个方式依次对剩余的训练集进行计算得:

由于我们的k=3，因此我们选择得到的距离最小的三个数字0.4782、0.3397、0.3896，其对应的标签（label）分别为“羊”、“牛”、“羊”，根据投票原则我们可以得到该预测的图像对应的标签应该为“羊”。

#### 3.利用曼哈顿距离进行计算

已知曼哈顿距离计算公式如下：

将公式带入数据可得：

与\_日1.png进行计算：

由同样的计算方式可得：

由于我们的k=3，因此我们选择得到的距离最小的三个数字0.88、0.81、0.93，其对应的标签（label）分别为“羊”、“牛”、“羊”，根据投票原则我们可以得到该预测的图像对应的标签应该为“羊”。

我们经过两次计算，已经可以深刻了解到KNN算法在进行预测时的数学原理了，但是KNN算法中对于距离的计算方式有很多，又应该如何进行选择呢？

#### 针对不同算法的选择

在 KNN（K-Nearest Neighbors）算法中，距离度量决定了样本之间的相似性，从而影响分类或回归的准确性。不同的距离计算方式适用于不同的数据特征和应用场景，选择合适的距离度量对 KNN 的性能至关重要。以下将从应用场景、模型大小（数据维度）、数据特征等方面，详细讨论欧几里得距离、曼哈顿距离和余弦相似度三种常见距离度量的适用性及选择原则。

**4.1. 欧几里得距离**

适用场景：

欧几里得距离是一种常用的距离度量方法，适合连续型数据的情况，尤其是在特征空间上各特征值的数量级一致、数据相对稠密且没有过多噪声的场景。欧几里得距离最适合点状、密集的数值数据，通常应用于图像处理、传感器数据、地理定位等需要精确的几何距离的任务中。例如，在图像识别或视觉处理任务中，图像的像素特征非常适合用欧几里得距离来计算相似度。

模型大小（数据维度）：

欧几里得距离容易受到高维空间的影响，即“维度灾难”。在高维空间中，样本点间的欧几里得距离差异会逐渐变小，导致相似度的有效性下降。此现象在维度较高的情况下更为明显。因此，若数据集维度较高，建议在降维（如 PCA）后使用欧几里得距离。

数据特征：

欧几里得距离假设数据空间中的每个特征是独立且等重要的，但对特征的尺度较为敏感。因此，在使用欧几里得距离之前，需要对数据进行归一化或标准化处理，以避免因不同特征的量纲差异而影响距离计算。比如，高分辨率图像数据会产生较大幅度的像素值变化，如果不标准化，距离计算可能会被某些幅度大的特征值主导。

总结：

优点：计算简单，适合密集型、尺度统一的数值型数据。

缺点：对高维数据表现较差，易受到噪声和尺度影响。

适用场景：图像分类、地理位置计算、连续性变量的相似度计算。

**4.2. 曼哈顿距离**

适用场景：

曼哈顿距离适用于特征稀疏、数据结构不规则的情况。在空间布局分布或网格状结构的任务中，曼哈顿距离表现更为合理，例如在城市街区路径计算中，街道通常是网格化的，行走距离遵循曼哈顿距离。它也适合在数据特征间有较大差异的场景，比如具有异质特征（特征间关系不同）的数据，例如房价预测、城市人口流动模型等，曼哈顿距离的鲁棒性更强。

模型大小（数据维度）：

曼哈顿距离对高维数据的适应性稍好于欧几里得距离，因为它不受维度之间的相互关系的影响。在高维空间中，曼哈顿距离比欧几里得距离更加鲁棒。但随着维度的增加，它也会遭遇“维度灾难”，需要结合降维技术提升模型效果。

数据特征：

曼哈顿距离适合稀疏数据，尤其是二值型、离散型特征较多的数据。这种距离对特征之间的尺度差异更具抗干扰性，即使没有对特征进行归一化处理，曼哈顿距离也能有效衡量样本间的相似性。这使得它适用于特征重要性差异大或特征间相互独立的场景。

总结：

优点：鲁棒性强，适合稀疏型、异质特征较多的数据。

缺点：与数据的网格结构关联性较强，易受高维影响，但不如欧几里得距离敏感。

适用场景：城市路径规划、带有离散特征或异质特征的数据集。

**4.3. 余弦相似度**

适用场景：

余弦相似度适用于文本数据或高维稀疏数据（如文本向量化后数据），更关注样本之间的“方向”而非“距离”。它特别适用于方向性强、数值大小不敏感的数据，适合数据在高维空间中的应用，如自然语言处理（NLP）、推荐系统、社交网络分析等场景。在文本分类中，词频向量是稀疏的，余弦相似度能够有效捕捉文档之间的相似性。

模型大小（数据维度）：

余弦相似度在高维数据中表现良好，因为它不依赖特征值的绝对大小，而是关注向量之间的夹角，即便在高维空间中也能稳定表现。这种特性使其在高维、稀疏数据的任务中具有优势。

数据特征：

余弦相似度对于特征值的尺度不敏感，不需要进行归一化或标准化。它更适合用于度量方向相似性，而非具体数值的距离。数据特征最好是非负值，因为余弦相似度对负值较敏感，通常会导致方向误差。在使用中，应注意数据是否符合余弦相似度的特性，如文本词频、嵌入向量等。

总结：

优点：适合高维稀疏数据，尤其适合在方向性相似度的场景下使用。

缺点：不适用于数值差异较大的数据；只适用于方向性相关的相似度判断。

适用场景：文本分类、推荐系统、社交网络节点相似度分析。

### 实现KNN算法

KNN算法代码实现主要包括数据处理、KNN分类器以及KNN训练三个模块，我们将按照每个模块来讲解KNN算法的代码实现：

#### 数据处理模块

在我们的示例中，我们对数据的命名规则为：下划线+图片标签+标号，因此在处理数据的时候主要包括一下内容：图像调整大小为9\*9、利用3\*3大小步长为3的核提取出图片的九个特征并展开为一维

def preprocess\_image(image\_path):

img = Image.open(image\_path).convert('L') # Convert to grayscale

img\_array = np.array(img)

# Determine the size of each grid cell

grid\_height, grid\_width = img\_array.shape[0] // 3, img\_array.shape[1] // 3

feature\_vector = []

for i in range(3):

for j in range(3):

# Define the boundaries of the grid cell

start\_i, end\_i = i \* grid\_height, (i + 1) \* grid\_height

start\_j, end\_j = j \* grid\_width, (j + 1) \* grid\_width

patch = img\_array[start\_i:end\_i, start\_j:end\_j]

avg\_value = round(np.mean(patch) / 255.0, 2) # Normalize and round to 2 decimal places

feature\_vector.append(avg\_value)

return feature\_vector

#### KNN分类器模块

针对于KNN的算法思路来逐步编写我们的代码，KNN算法的思路可以整理为：遍历训练集计算距离 -> 排序并选择k个最近距离 -> 投票法判断新数据标签 -> 记录训练集的数据以便后续使用。根据这个思路来完成我们的代码：

def knn\_classifier(train\_data, train\_labels, test\_data, k=3):

distances = []

# 遍历训练集计算距离

for i in range(len(train\_data)):

# 计算测试数据和训练数据样本之间的欧几里得距离

dist = np.sqrt(np.sum((test\_data - train\_data[i]) \*\* 2))

# 将距离和对应的标签存入 distances 列表

distances.append((dist, train\_labels[i]))

# 排序并选择 k 个最近距离

# 根据距离排序，获取距离最近的 k 个邻居

distances.sort(key=lambda x: x[0])

nearest\_neighbors = distances[:k]

# 投票法判断新数据标签

# 提取最近 k 个邻居的标签

labels = [label for \_, label in nearest\_neighbors]

# 获取所有训练集中的可能标签

all\_labels = list(set(train\_labels))

# 通过计数找到出现次数最多的标签作为预测结果

prediction = max(set(labels), key=labels.count)

# 输出所有可能的标签和预测的标签

print(f"可能的标签：{all\_labels}")

print(f"预测的标签：{prediction}")

return prediction

# 将训练数据保存到 CSV 文件以供将来使用

def save\_training\_data\_to\_csv(train\_data, train\_labels, filename="training\_data.csv"):

# 将训练数据和标签转换为 DataFrame 并保存为 CSV

df = pd.DataFrame(train\_data, columns=[f"Feature\_{i + 1}" for i in range(train\_data.shape[1])])

df['Label'] = train\_labels

df.to\_csv(filename, index=False)

print(f"训练数据已保存到 {filename}")

#### KNN训练模型

KNN模型在训练时，我们遵循一下流程：定义路径并预处理训练集 -> 将新保存的训练数据保存到csv文件 -> 从csv文件加载训练数据并进行预测。

def main():

# 1. 定义路径并准备训练数据（根据需要修改用于新数据集或重新训练）

dataset\_folder = r"Y:\py-torch\Machine\_learing\_pytorch\KNN\TEST\INPUT" # 替换为实际训练图像文件夹路径

train\_data = []

train\_labels = []

# 2. 加载并预处理训练数据集中的每张图像

for filename in os.listdir(dataset\_folder):

if filename.endswith(".png") and '\_' in filename:

# 从文件名中提取标签并处理图像

label = filename.split('\_')[1][0] # 假设标签是'\_'后面的第一个字符

image\_path = os.path.join(dataset\_folder, filename)

feature\_vector = preprocess\_image(image\_path)

train\_data.append(feature\_vector)

train\_labels.append(label)

# 将 train\_data 转换为 numpy 数组

train\_data = np.array(train\_data)

# 3. 将新处理的训练数据保存到 CSV

save\_training\_data\_to\_csv(train\_data, train\_labels)

# 4. 从保存的 CSV 文件加载训练数据，以便进行预测

train\_df = pd.read\_csv("training\_data.csv") # 加载刚保存的 CSV 文件

train\_data = train\_df.iloc[:, :-1].values # 获取特征向量，排除标签列

train\_labels = train\_df['Label'].values # 仅加载标签

# 5. 测试图像的处理以进行预测

test\_image\_path = r"Y:\py-torch\Machine\_learing\_pytorch\KNN\TEST\test\_image\aoaipuy20h\_h01216\_180\_347\_49\_63.png" # 根据需要修改

test\_feature = preprocess\_image(test\_image\_path)

# 6. 使用 KNN 进行预测并打印结果

predicted\_label = knn\_classifier(train\_data, train\_labels, np.array(test\_feature), k=3)

以上，便是我们利用六张图片作为简单的训练集以及一张图片作为预测数据进行的KNN算法推导，以上的KNN算法包括数据代码等均是为了方便理解而进行的简化操作，在一般使用KNN算法进行训练时，我们或许会面对更加庞大的数据集，届时我们可以使用sklearn库或者torch库来进行训练，只需要调试参数而不需要重复的造轮子了。

### 使用sklearn库实现KNN算法预测

#### 1.使用sklearn实现KNN算法

接下来我们将利用sklearn库简单演示一下KNN算法的相关实现。针对数据集的命名规则为：xxxxx\_labelxxxx，即为train或test文件夹下每一个子文件夹为一个类别，其标签为子文件夹名称后第一个字符，共有60个甲骨文类别，代码如下：

import os # 导入os模块，用于处理文件和目录路径

import time # 导入time模块，用于测量时间

from sklearn.neighbors import KNeighborsClassifier # 从sklearn库导入KNeighborsClassifier类，用于KNN分类

from sklearn.metrics import accuracy\_score # 从sklearn库导入accuracy\_score函数，用于计算准确率

from torchvision import datasets, transforms # 从torchvision库导入datasets和transforms模块，用于加载和预处理数据集

from torch.utils.data import DataLoader # 从torch.utils.data导入DataLoader类，用于加载数据

import numpy as np # 导入numpy库，并简写为np，用于处理数组

# 定义数据预处理函数

def get\_transform():

transform = transforms.Compose([

transforms.Resize((112, 112)), # 将图片大小调整为112x112

transforms.Grayscale(num\_output\_channels=1), # 将图片转换为灰度图

transforms.ToTensor(), # 将图片转换为Tensor格式

])

return transform

# 定义加载数据集函数

def load\_dataset(root\_dir, transform=None):

if transform is None:

transform = get\_transform() # 如果未提供transform，则使用默认的预处理步骤

data = datasets.ImageFolder(root=root\_dir, transform=transform) # 使用ImageFolder加载数据集

return data

def train\_knn(train\_loader):# 训练KNN模型

features = [] # 存储特征

labels = [] # 存储标签

for images, labels\_batch in train\_loader: # 遍历训练数据

features.extend(images.view(images.size(0), -1).numpy()) # 提取特征并转换为numpy数组

labels.extend(labels\_batch.numpy()) # 提取标签并转换为numpy数组

features = np.array(features) # 将特征列表转换为numpy数组

labels = np.array(labels) # 将标签列表转换为numpy数组

knn = KNeighborsClassifier(n\_neighbors=3, algorithm="auto", p=2) # 创建KNN模型

start\_time = time.time() # 记录训练开始时间

knn.fit(features, labels) # 训练模型

end\_time = time.time() # 记录训练结束时间

print(f"Training time: {end\_time - start\_time:.2f} seconds") # 打印训练时间

return knn # 返回训练后的模型

# 测试KNN模型

def test\_knn(model, test\_loader):

features = [] # 存储特征

for images, \_ in test\_loader: # 遍历测试数据

features.extend(images.view(images.size(0), -1).numpy()) # 提取特征并转换为numpy数组

features = np.array(features) # 将特征列表转换为numpy数组

start\_time = time.time() # 记录测试开始时间

predictions = model.predict(features) # 使用模型进行预测

end\_time = time.time() # 记录测试结束时间

print(f"Testing time: {end\_time - start\_time:.2f} seconds") # 打印测试时间

labels = np.array([label for \_, label in test\_loader.dataset]) # 提取测试数据的标签

accuracy = accuracy\_score(labels, predictions) # 计算准确率

print(f"Accuracy: {accuracy:.2f}") # 打印准确率

# 主函数

def main():

"""

主函数，执行训练和测试流程。

"""

train\_dir = r"Y:\py-torch\甲骨文切割\train\_拓片" # 训练数据集目录

test\_dir = r"Y:\py-torch\甲骨文切割\test\_拓片" # 测试数据集目录

train\_data = load\_dataset(train\_dir) # 加载训练数据集

test\_data = load\_dataset(test\_dir) # 加载测试数据集

train\_loader = DataLoader(train\_data, batch\_size=32, shuffle=True) # 创建训练数据加载器

test\_loader = DataLoader(test\_data, batch\_size=32, shuffle=False) # 创建测试数据加载器

model = train\_knn(train\_loader) # 训练模型

test\_knn(model, test\_loader) # 测试模型

if \_\_name\_\_ == "\_\_main\_\_":

main() # 执行主函数

#### **2.**针对sklearn库中KNN算法的超参数讲解

在KNN算法中，超参数的选择对算法性能有着至关重要的影响。以下是KNN算法中几个关键超参数的详细讲解：

**n\_neighbors（K值）：**

n\_neighbors是KNN算法中最主要的超参数，它表示选择最近邻居的数目，即K值。

K值的选择对算法结果有很大影响。K值太小，算法容易受到噪声的影响，导致过拟合；K值太大，则可能使邻居点包含其他类别的点，导致欠拟合。

通常，K值会取一个较小的整数值，如3、5、7等。在实际应用中，可以通过交叉验证等方法来确定最佳的K值。

**algorithm：**

algorithm参数用于指定计算最近邻居的算法。常用的算法有'brute'（暴力搜索）、'kd\_tree'（KD树）、'ball\_tree'（球树）和'auto'（自动选择）。

暴力搜索是通过计算查询点与所有训练点之间的距离来找到最近邻居，适用于小规模数据集。

KD树和球树是两种空间划分数据结构，它们可以用于加速大规模数据集上的近邻搜索。

'auto'选项会根据数据的特性自动选择最合适的算法。

**p：**

p参数用于指定闵可夫斯基距离中的p值。

闵可夫斯基距离定义式如下：

闵可夫斯基距离是一种广义的距离度量方法，经过简单推导可得，当p=2时，它退化为欧氏距离；当p=1时，它退化为曼哈顿距离。

通过调整p值，可以改变距离度量的性质，从而影响KNN算法的性能。

在实际应用中，需要根据数据的具体情况选择合适的p值。例如，当数据特征具有不同的量纲或分布时，可能需要通过调整p值来获得更好的性能。

# ID3-决策树算法

### 算法介绍

ID3算法的核心思想是通过计算信息增益（Information Gain）来选择最优的特征作为决策树的节点，从而实现数据的划分。信息增益是衡量特征在划分数据集后能够减少的不确定性（即熵）的指标。具体来说，信息增益越大，表示该特征对数据集的划分效果越好，能够使得子集的纯度更高，不确定性更小。因此，ID3算法在构建决策树时，优先选择信息增益最大的特征进行划分，以达到最佳的分类效果。

在ID3决策树算法中，有以下几个概念：

#### 熵

熵在这里通俗的来说是平均信息量，是对被传送信息度量采用的平均值。与化学和物理中的概念类似，熵反应的是系统的混乱程度，熵越大，系统的混乱程度越高，信息越不纯。对于一个有序系统，它的熵为0.

#### 消息量

一个消息的信息传递量为，其中p为一个消息的概率。因为概率是一个小于1的数，而信息传递量又是一个大于等于0的数，所以在传递量前有一个符号，此时消息传递量为正。对于n个消息，概率分布为，其概率分布传递的信息量为，此时的信息量也成为该概率分布的信息熵。

#### 信息增益

信息增益用于衡量某一特征变量对于分类任务所能提供的信息量，即该特征变量在区分不同类别时的有效性。信息增益的基本思想是通过比较在使用某个特征进行划分前后的信息熵（或称为不确定性）的减少量来评估该特征的重要性。公式如下：

其中，A是当前考虑的特征，Values(A)是特征A的所有取值，是特征值A取值为v时的子集，||是自己的样本数量，H()是子集的信息熵。

### 算法基本流程

**1.计算当前数据集的熵（Entropy）：**熵是衡量数据集纯度的指标，熵越小，数据集越纯。首先，计算整个数据集的熵，作为划分前的数据不确定性。

**2.计算各特征的信息增益（Information Gain）：**对于每一个特征，按照该特征的取值划分数据集，并计算划分后的子集的熵。然后，根据信息增益的公式计算该特征的信息增益。信息增益表示在使用该特征进行数据划分后，信息的不确定性减少的程度。

**3.选择信息增益最大的特征：**在所有特征中，选择信息增益最大的特征作为当前节点的划分属性。这个特征能够最大化地减少数据的不确定性，提供最有效的划分。

**4.递归地构建子树：**对每一个由选定特征划分出的子集，递归地重复上述步骤。在每次递归时，忽略已经用作划分的特征。直到满足停止条件：子集中的所有实例属于同一类别，或者没有更多的特征可供选择。

**5.构建决策树：**通过递归选择最优特征，并将其作为决策树的节点，最终构建完整的决策树。

### ID3决策树算法数学推导

有了对算法的基本认识以及对流程的了解，我们就可以使用其数学公式来推导一遍整个ID3决策树算法帮助我们更深入地理解算法的流程。我们使用的数据依旧是之前经过处理的几张图像。

训练集数据：

Filename,Feature\_1,Feature\_2,Feature\_3,Feature\_4,Feature\_5,Feature\_6,Feature\_7,Feature\_8,Feature\_9,Label

\_日1.png,0.72,0.45,0.79,0.66,0.75,0.84,0.84,0.60,0.80 ,日

\_日2.png,0.68,0.65,0.66,0.68,0.69,0.79,0.63,0.64,0.61,日

\_牛1.png,0.78,0.98,0.76,0.69,0.41,0.88,0.93,0.58,1.0,牛

\_牛2.png,0.74,0.77,0.75,0.85,0.41,0.92,0.88,0.63,0.93,牛

\_羊1.png,0.82,0.66,0.73,0.91,0.55,0.95,1.0,0.78,1.0,羊

\_羊2.png,0.74,0.69,0.74,0.88,0.5,0.89,0.82,0.8,0.92,羊

测试集数据：

Feature\_1,Feature\_2,Feature\_3,Feature\_4,Feature\_5,Feature\_6,Feature\_7,Feature\_8,Feature\_9

0.72,0.58,0.63,0.93,0.36,0.91,1.0,0.69,1.0

#### **1.**计算信息熵

首先需要计算信息的总信息熵。针对分类问题，信息熵用于衡量信息纯度。信息熵公式如下：

在我们的数据中，共有6张3个类别，其类别分布入下：

因此，计算总信息熵H(D)为：

#### **2.**计算信息熵增益

为了选择最佳特征进行分裂，我们需要计算每个特征的**信息增益**，信息增益反映了一个特征在分裂数据集时能够带来的纯度提升，在之前我们就已经介绍过信息增益的相关计算公式，接下来我们进行特征分裂的计算。

**我们首先根据特征Feature\_1进行分裂，可以得到两个子集：**

(特征 > 0.75) ：包含牛1.png，牛2.png，羊1.png，羊2.png，标签为 "牛" 和 "羊"。

(特征 <= 0.75) ：包含 日1.png，日2.png，标签为 "日"。

**子集1的信息熵H()**

由子集1的标签分布可得：

则子集1的信息熵为：

**子集2的信息熵H()**

由子集2的标签分布可得：

子集2的信息熵为：

由以上分裂情况可得由Featur\_1的信息增益为：

类比以上过程，我们分别计算出其余特征的信息增益：

我们选择信息增益最大的特征值进行分裂，在这里我们选择作为根节点进行分裂预测。

**步骤 1: 根据分裂数据集**

子集 1 (> 0.75)：样本为 牛1.png, 牛2.png, 羊1.png, 羊2.png

标签分布：牛 2个, 羊 2个

对此子集继续计算信息增益，选择最优特征分裂。

子集 2 ( <= 0.75)：样本为 勿1.png, 勿2.png

标签分布：勿 2个

这个子集已经纯净，标签为 "勿"，无需继续分裂。

**步骤 2: 继续分裂子集 1**

我们对 > 0.75 的子集进行进一步的分裂。接下来，我们计算 的信息增益。

子集 1 ( > 0.75)：样本为 牛1.png, 牛2.png

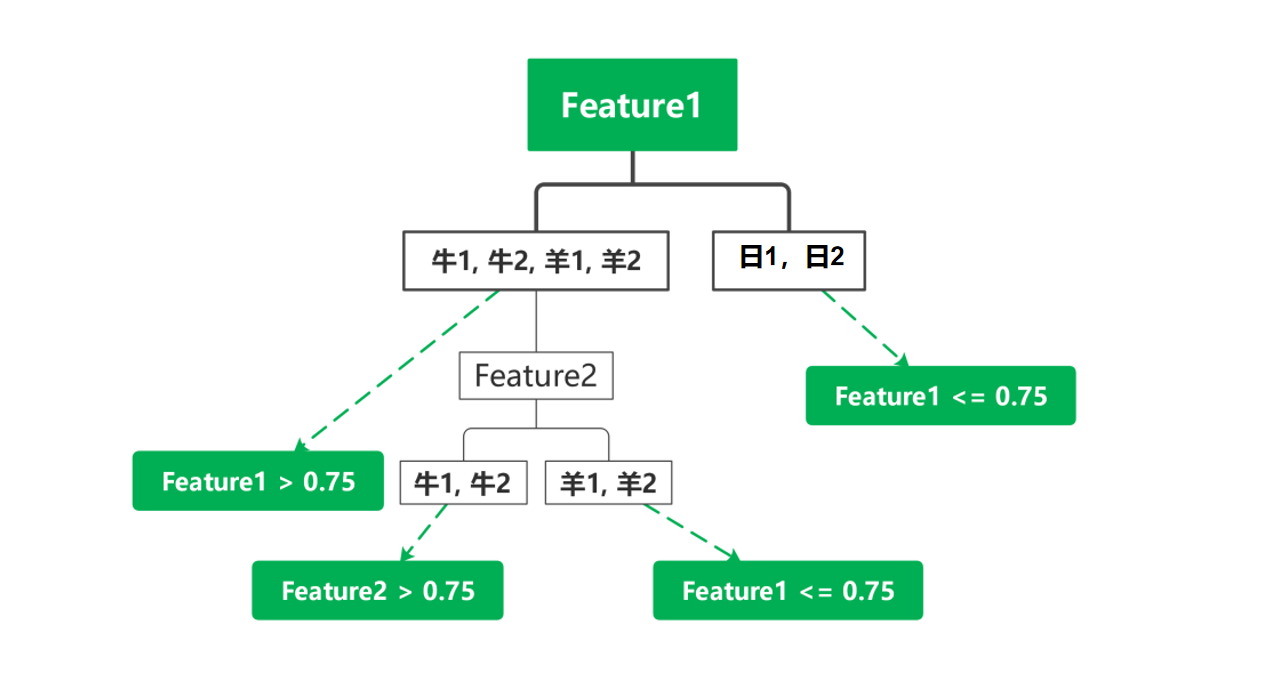
标签分布：牛 2个

这个子集纯净，标签为 "牛"，无需继续分裂。

子集 2 ( <= 0.75)：样本为 羊1.png, 羊2.png

标签分布：羊 2个

这个子集纯净，标签为 "羊"，无需继续分裂。

由此，我们便得到了我们的决策树，决策树如下：

在测试数据中， = 0.72 < 0.75，因此，对于测试数据的预测标签为“日”。

我们可以发现，在本次数学推导的过程中，预测结果是错误的（预测值的标签为“羊”），这是因为ID3决策树算法大部分是并不适用于图像识别任务中，相比之下，ID3决策树算法在面对回归问题等离散型数据问题时可能会有更好的表现。

#### 实现ID3-决策树算法

接下来我们将尝试使用sklearn库来实现ID3-决策树算法，在之后会简单讲解一下算法中不同超参数如何调整。

import os

import time

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

def load\_data\_and\_labels(directory):

texts = []

labels = []

for label\_id, subdir in enumerate(os.listdir(directory)):

if os.path.isdir(os.path.join(directory, subdir)):

for filename in os.listdir(os.path.join(directory, subdir)):

with open(os.path.join(directory, subdir, filename), 'r', encoding='utf-8', errors='ignore') as f:

texts.append(f.read())

labels.append(label\_id)

return texts, labels

# 加载数据

train\_texts, train\_labels = load\_data\_and\_labels(r"Y:\py-torch\甲骨文切割\train\_拓片")

test\_texts, test\_labels = load\_data\_and\_labels(r"Y:\py-torch\甲骨文切割\test\_拓片")

# 向量化

vectorizer = TfidfVectorizer()

X\_train = vectorizer.fit\_transform(train\_texts)

X\_test = vectorizer.transform(test\_texts)

# 决策树分类器

clf = DecisionTreeClassifier()

# 记录训练时间

start\_time = time.time()

clf.fit(X\_train, train\_labels) # 直接使用稀疏矩阵进行训练

train\_time = time.time() - start\_time

# 记录测试时间

start\_time = time.time()

predicted = clf.predict(X\_test) # 直接使用稀疏矩阵进行预测

test\_time = time.time() - start\_time

# 准确率

accuracy = accuracy\_score(test\_labels, predicted)

print(f"Training time: {train\_time:.2f} seconds")

print(f"Testing time: {test\_time:.2f} seconds")

print(f"Accuracy: {accuracy:.4f}")

超参数讲解：

1. criterion: {'gini', 'entropy'}

'gini'表示使用Gini不纯度作为分裂标准（CART算法默认）。

'entropy'表示使用信息增益作为分裂标准（即ID3算法）。

调试建议：对于大多数情况，两者都可以尝试，但如果你明确要使用ID3，则选择'entropy'。

2. splitter: {'best', 'random'}

'best'表示选择最佳分裂。

'random'表示选择最佳随机分裂（适用于随机森林等集成方法）。

调试建议：一般使用'best'。

3. max\_depth: int or None

限制树的最大深度。如果为None，则扩展节点直到所有叶子都是纯净的或者直到所有叶子包含少于min\_samples\_split个样本。

调试建议：可以从较大的值开始，然后逐步减小以避免过拟合。

4. min\_samples\_split: int or float

分裂内部节点所需的最小样本数。如果是浮点数，则表示为总样本数的比例。

调试建议：增加这个值可以防止模型创建过多的节点，有助于避免过拟合。

5. min\_samples\_leaf: int or float

在叶节点处需要的最小样本数。这个值限制了树的生长，如果一个分裂导致任何一个叶节点包含的样本数少于这个值，则分裂不会进行。

- 调试建议：增加这个值可以使模型更加保守，有助于避免过拟合。

6. max\_features: int, float or {'auto', 'sqrt', 'log2'}

寻找最佳分裂时要考虑的特征数量。

调试建议：对于高维数据，可以尝试减少这个值来加速训练。

7. random\_state: int, RandomState instance or None

控制构造树时样本的随机性。

调试建议：为了确保结果的可重复性，设置一个固定的种子值。

8. max\_leaf\_nodes: int or None

最大叶子节点数。通过`max\_leaf\_nodes`来生长树，而不是`max\_depth`。

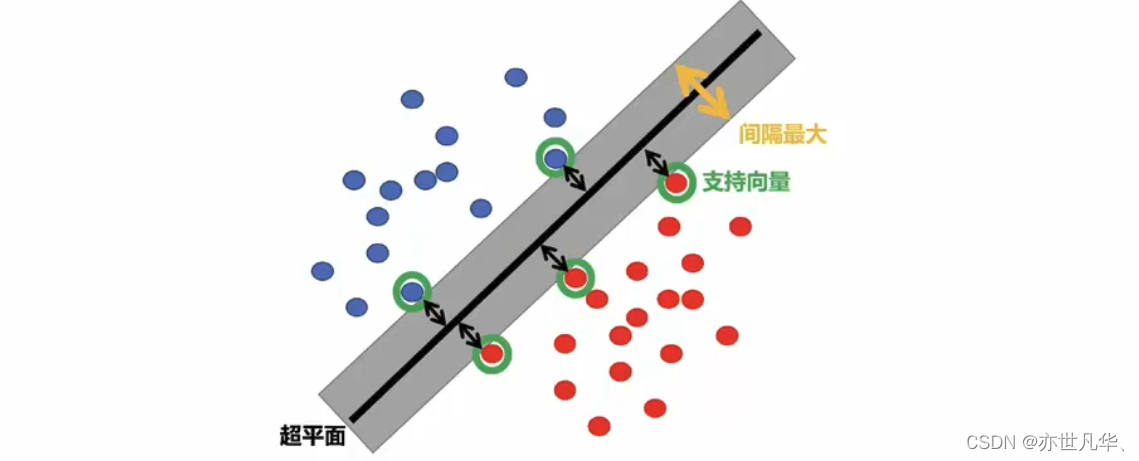
调试建议：可以尝试使用这个参数来限制树的复杂度，特别是在你对树的深度没有直接控制时。

# SVM支持向量机算法

在机器学习的早期，Fisher判别式被认为是解决分类问题的经典方法，它通过寻找一个线性投影，使得不同类别的数据能够在低维空间中分开，且类间方差最大化、类内方差最小化。然而，Fisher判别式在实际应用中也暴露出了一些局限性，尤其是在面对高维数据时，效果往往不尽如人意。首先，Fisher判别式假设数据服从高斯分布，并且要求各类别之间是线性可分的，这在很多实际问题中并不成立。此外，Fisher判别式在处理复杂、非线性的数据分布时也显得力不从心，无法找到合适的分类边界。为了克服这些问题，支持向量机应运而生。SVM通过寻找一个超平面来最大化类别之间的间隔，不仅能够有效处理线性可分问题，还可以通过引入核函数处理非线性分类问题，从而在各种复杂情况下都能提供稳定且高效的分类效果。SVM的强大之处在于其能够在高维空间中工作，并通过最大化间隔来提高模型的泛化能力，使其成为在许多实际应用中不可或缺的工具。 支持向量机（SVM）是一种广泛应用的监督学习算法，尤其在分类任务中表现出色。其基本思想是寻找一个“最优分隔超平面”（optimal separating hyperplane），使得不同类别的样本能够得到良好的划分，且边界最大化。

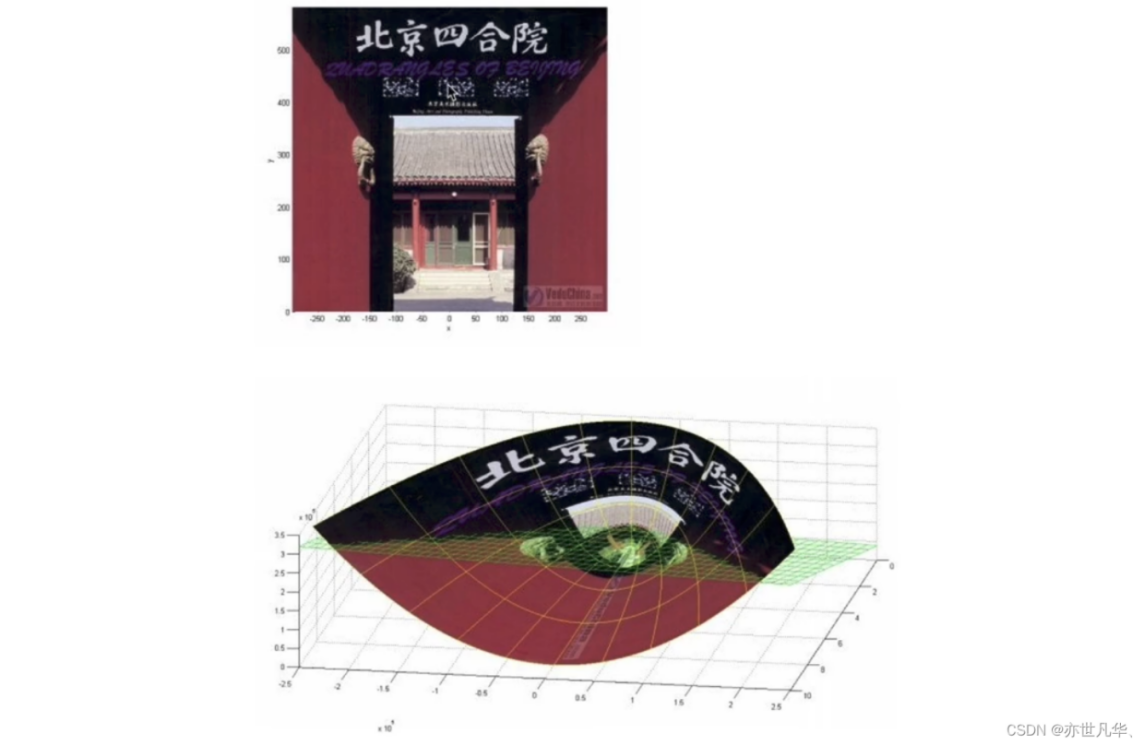
### 1.SVM算法基本思想

SVM算法是一种基于二分类问题的算法，对于不同的数据，通过寻找到一个“超平面”来尝试将数据分为两类。一般情况下“超平面”维度为数据维度-1，例如对于一个二维数据，那么“超平面”就是一条直线；对于三维数据，“超平面”则是一个二维平面。



二维数据的超平面分类

在面对一个线性不可分的数据时，可以考虑将数据映射到更高维度，然后使用超平面进行划分，再映射到原维度。



如上图所示，如果想要把门红色的部分与黑色阴影部分分开，直接在二维平面难以找到一个一维的超平面直接完成，因此我们可以将该图像映射到三维空间内，这时候就可以找到一个二维的超平面完成我们的任务。而将一个低维数据升到高维的操作，我们称为**核技巧**。

### 2.SVM算法的数学原理

在了解SVM算法的数学原理时，我们首先了解一下SVM算法在进行预测时候的基本流程：

定义优化目标->带入拉格朗日函数->对w和b求偏导->求解对偶问题得到拉格朗日乘子->计算支持向量->求解w和b并利用决策函数进行预测

根据以上流程，我们将逐步了解SVM算法的数学原理。

#### 2.1 定义优化目标

SVM算法目标是通过选择一个超平面来最大化分类间隔，同时最小化分类误差。分类最大化间隔相当于要满足最小可得公式：

这是一个带有不等式约束的凸优化问题，其中，代表松弛变量，允许分类错误；C代表正则化参数，权衡分类间隔与误分类代价；w代表超平面的法向量，b代表偏置，代表样本，代表标签（由于SVM算法是基于二分类问题的算法，其标签一般表示为1和-1。在面对多分类问题时，我们可以通过其他策略如**一对多策略**（**OvA**）将其推广到多分类问题上）。

那么，在这个公式中，为什么要如此定义间隔呢？

首先，我们假设已经对每一个特征x都已经赋予了一个对应的标签y，由于SVM算法是一种基于二分类问题算法，所以y一般只有两个值：1或-1.那么对于N个特征

我们认为，当存在一个向量w和一个标量b满足下列不等式，则称其为线性可分：

将上述式子结合成为一个式子如下：

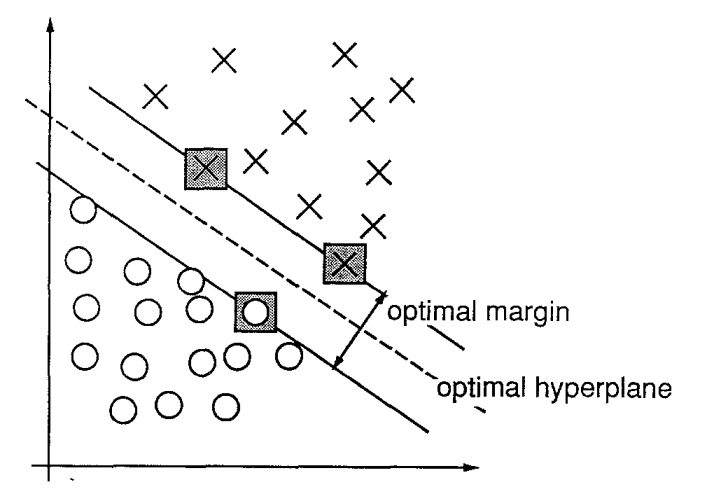
以上就是优化目标的约束条件，总的来说，我们要约束优化目标为一个线性可分的式子。为什么一定要要求线性可分呢？

首先SVM的基本思路是寻找一个能够分隔两类样本的超平面（在二维空间中是直线）。如果数据集是线性可分的，就可以通过一个超平面（或直线）将两类样本完全分开，且保证两类样本的边界距离最大化，从而达到最佳的分类效果。

其次，SVM算法中的一个重要特点是“最大间隔分类”，即希望分割两类样本的超平面离两类样本的最近点尽可能远。通过最大化间隔，能够提高模型对新样本的分类准确性和泛化能力。

最后，在线性可分的情况下，SVM的目标函数是凸优化问题，具有唯一的解，且解可以通过数学方法（如拉格朗日乘子法）有效求解。在线性可分的情况下，求解过程更为直接且稳定。

我们知道，在SVM算法中，我们优化目标是为了最大化间隔，即最大化下图中的optimal margin。



那么，如何表示这个间隔呢？

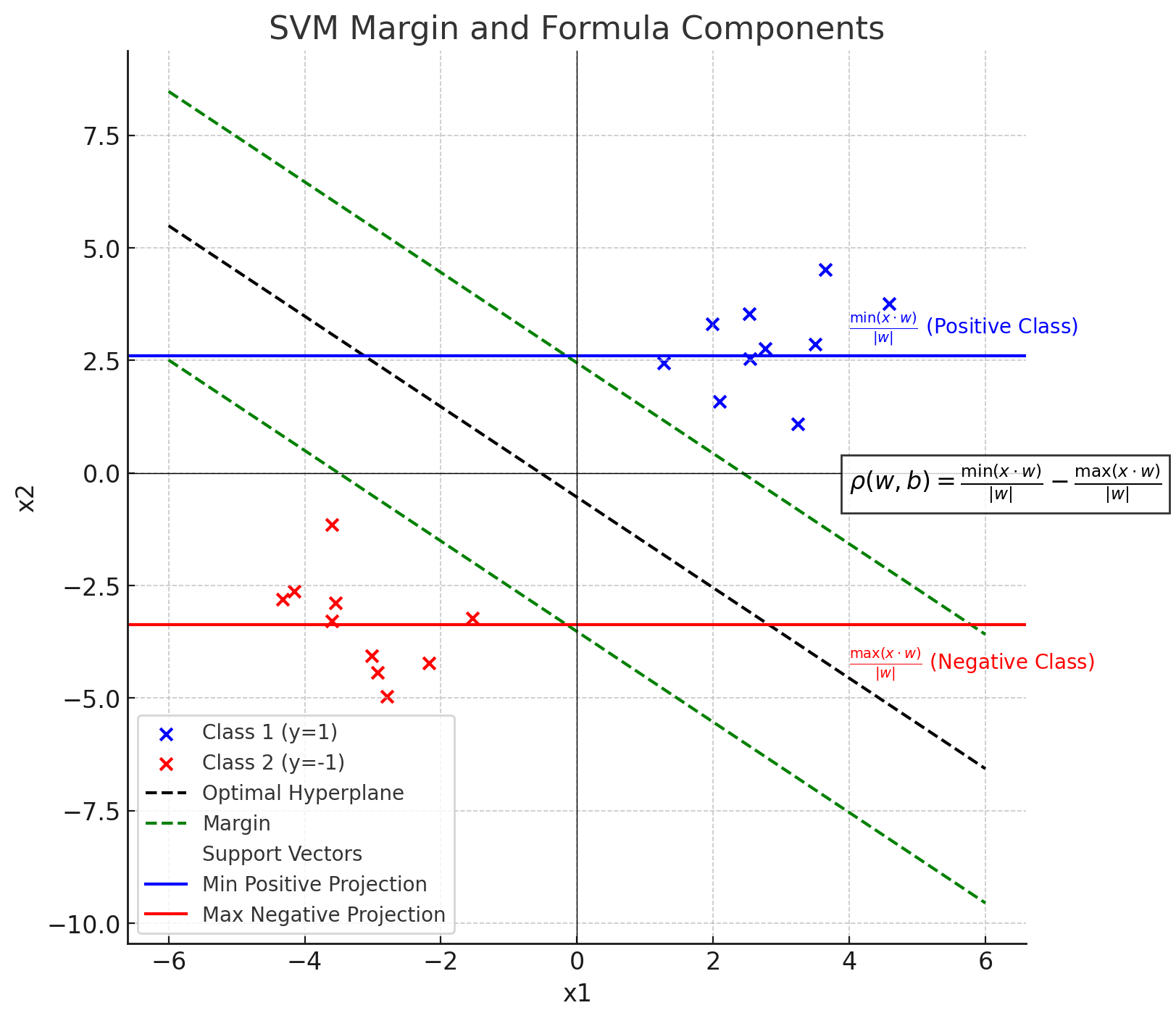
分类间隔的定义：

分类间隔是正类点和负类点中距离超平面最近的两点之间的投影距离。

正类点的最近投影：

负类点的最近投影：

分类间隔即为两类点最近投影差：



那么正负类点的投影距离又是怎么来的呢？在SVM算法中，.接下来我们将讲解一下如何得到这个公式。

已知有一个超平面,我们将点到超平面的距离定义为从到超平面上最近点的距离

其中，是在超平面上的正交投影点。

正交投影点的条件：

对于超平面上任意点都有

如果是的正交投影点，那么从到的向量必须与法向量w平行，可得式

其中λ为标量。

确定λ的值：

根据我们刚刚得到的两个公式，我们将带入可得

解得

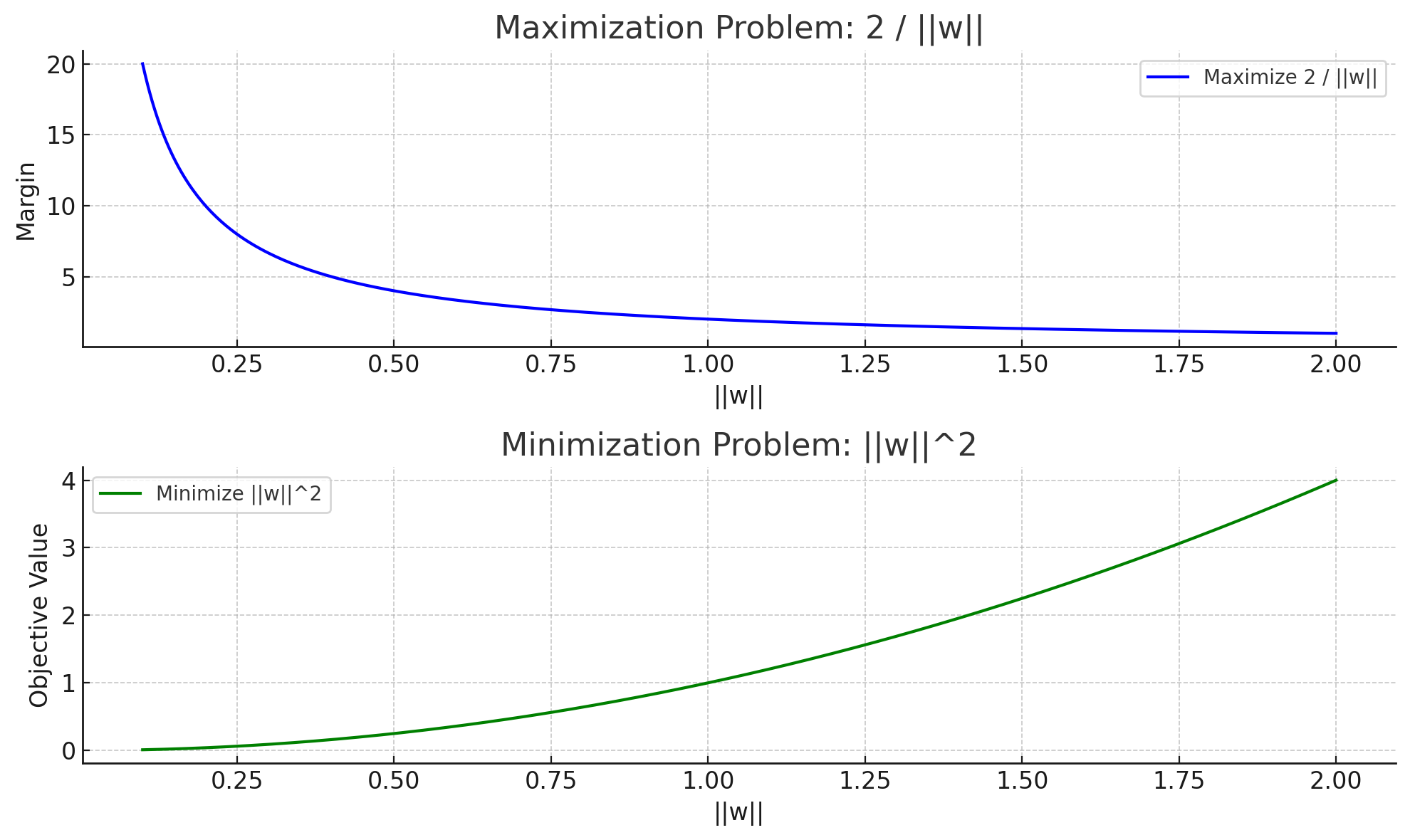
得到距离公式：

我们已经了解了的定义式以及由来接下来根据归一化条件

带入可得

其中，是正类最近点，其满足，是负类最近点，其满足。因为两点在超平面上的投影距离总是 2（从+1到-1），间隔可以简化为：

由于我们要最大化，等价于最小化（在优化问题中，这两个式子的结果相等并且的导数为，更易计算梯度）。为什么要把一个最大化的问题想尽办法等价为一个最小化问题呢？从本质上来讲是为了简化计算。如果直接使用原式子**最大化目标求二次规划问题**的话，是一个非线性且非凸的目标函数，直接最大化会在数学上增加复杂性。如果将目标函数转化为最小化，那么问题就会转化为一个凸优化问题，有更加成熟的数学工具来进行计算且更加便捷。



上半部分为最大化，下半部分为最小化

刚才我们提到二次规划问题，为什么求解SVM的优化目标是一个二次规划问题呢?

求解 SVM 的优化目标是一个二次规划问题，这是因为：

1.其目标函数是二次函数，用于间接实现最大化分类间隔的目标；

2.其约束条件是线性的；

3.二次目标函数与线性约束的结合符合二次规划问题的定义。

通过这样的定义，SVM 的优化可以借助成熟的二次规划求解算法，如拉格朗日对偶法，确保问题得到高效且稳定的求解。

在之前的描述中我们提到：“我们认为，当存在一个向量w和一个标量b满足下列不等式，则称其为线性可分。”如果这时候不存在这样的w和b，我们应该如何处理呢？

这时候就要引入硬间隔与软间隔的概念。

如果我们直接找到了w和b，不引入其他因素，那么就称为硬间隔，其定义式如下：

当不存在这样的w和b的时候，我们就要引入松弛变量以及正则化系数。

这是一个带有不等式约束的凸优化问题，其中，代表松弛变量，允许分类错误；C代表正则化参数，权衡分类间隔与误分类代价。在软间隔的定义式中，我们将允许一定的分类错误存在，然而具体对于错误的容忍程度，就是由我们定义的决定的。

#### 2.2 引入拉格朗日函数

我们通过定义优化目标的推导已经得知：SVM算法最大化间隔的优化目标是一个凸二次规划问题。那么如何求解凸二次规划问题呢？在这里我们选择使用**拉格朗日乘子法**，这是因为拉格朗日乘子法自然满足二次规划问题的**KKT条件**，我们不需要额外的去考虑KKT条件中的约束。

拉格朗日函数的定义一般可以定义为

对于一个约束优化问题

通过引入拉格朗日乘子和，我们可以定义**拉格朗日函数**如下;

KKT条件主要包括以下四项：

1.可行条件;

2.拉格朗日条件（驻点条件）：

其中，表示对变量x的梯度运算，即对的所有分量求偏导数，结果是一个向量：

它描述了函数在x方向上的变化率，优化问题的梯度平衡是关键的最优条件；是目标函数在最优点处的梯度（向量）；拉格朗日乘子，表示非等式约束在最优处的“权重”，如果（即约束不活跃），，

如果（即约束活跃），则会对优化方向产生影响；表示对x的梯度，代表非等式约束的变化率方向；由以上可知，组合项代表约束在最优点处对优化方向的影响，如果约束不活跃，则其对梯度平衡无贡献；等式约束则始终有贡献。

3.互补松弛性:

互补松弛性描述了拉格朗日乘子与不等式约束的关系。

4.**拉格朗日乘子非负性**：

接下来我们逐步分析拉格朗日函数是如何自然满足KKT条件的：

1.可行条件：

在拉格朗日函数中，包含了约束条件，在优化过程中，解是在约束条件被激活时找到的（否则无法定义优化目标），因此自然满足可行性条件。

2.拉格朗日条件（驻点条件）：

拉格朗日函数的优化本质上是在目标函数 和约束的“权衡”之间寻找平衡点。当时，拉格朗日函数对x的偏导为零，表示找到了最优的。

3.松弛互补条件：

拉格朗日函数中，是一项，如果，为了优化拉格朗日函数，必须满足，否则会对目标函数产生负影响。当时，约束被激活，此时，用于调节目标函数。

4.非负性条件：

如果 ，意味着约束会对目标函数产生类似“奖励”的作用，这与优化方向不符。

我们已经了解了二次规划问题以及用于求解二次规划问题的KKT条件和求法拉格朗日乘子法，我们已知SVM算法中求解最大间隔本质上是一个二次规划问题，我们便可以构造对应的拉格朗日函数来对该问题进行正对性的求解。

我们将拉格朗日函数（软间隔）定义如下：

硬间隔：

其中，N代表样本数量，α、代表拉格朗日乘子。

#### 2.3 对w和b求偏导并令其等于零

我们对w、b以及其他变量求偏导的目的是为了求解优化问题的**必要条件**。偏导数等于 0 的条件保证了拉格朗日函数在最优解点处满足一阶必要条件（KKT 条件）。为了求解这个优化问题，我们需要满足**一阶必要条件**。这些条件包括：

1. 对的偏导数为0（可行域内平衡条件）；
2. 约束条件；
3. 互补松弛条件。

对w、b求导是为了消去这些变量，得到仅关于拉格朗日乘子的对偶优化问题。

**以上这个式子表示，最优超平面解可以写成训练向量的线性组合。**

#### 2.4 求解对偶问题求解拉格朗日乘子

由于原拉格朗日函数中包含过多变量，在求解的时候就显得十分困难，因此我们可以将权重、偏置等转化为只包含拉格朗日乘子的式子然后带入拉格朗日函数，这样我们就将式子简化为支取要求解拉格朗日乘子的形式了。

将对w求偏导带入拉格朗日函数，对b求偏导作为约束，可得对偶问题公式如下：

#### 2.5 求解支持向量

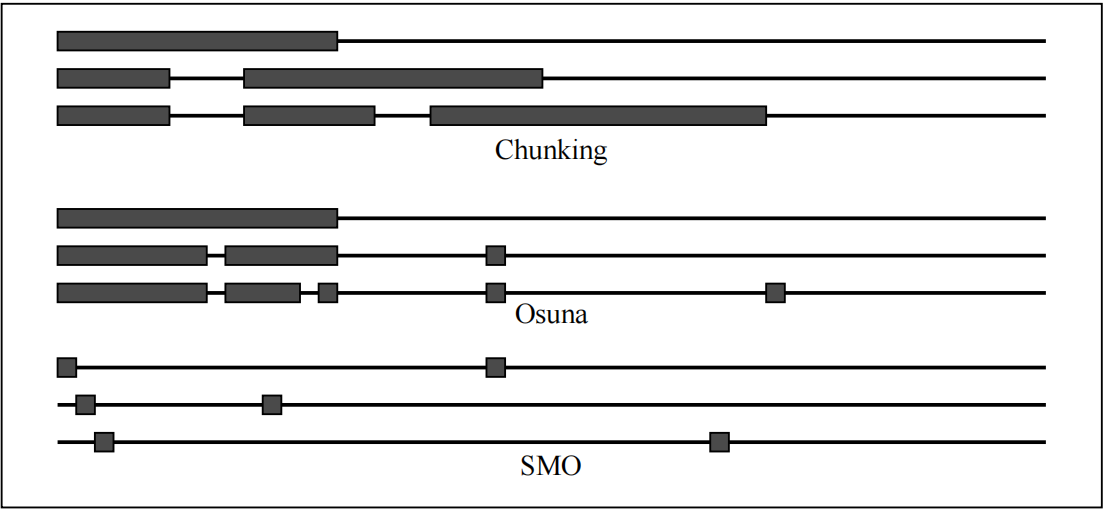
从某些角度来讲，我们在之前所求的拉格朗日乘子，即为支持向量，那么如何求解拉格朗日乘子呢？在这里我们使用SMO算法结合KKT条件对拉格朗日乘子进行求解。

###### **SMO（最小化序列）算法原理**

由于拉格朗日转化为对偶问题以后，二次形式涉及到一个矩阵，该矩阵的元素的数量等于训练示例的数量的平方。如果有超过4000个训练示例，这个矩阵就不能容纳128兆字节。

Vapnik在其提出的Support Vector Network理论中，定义了SVM算法的二次规划问题。之后，Osuna等学者提出了“分块法”（Chunking）作为解决这一问题的一种有效方法。分块法的基本思想是，通过移除与零拉格朗日乘数相关的矩阵行列，这样二次目标函数的值保持不变。因此，大规模的二次规划问题可以被分解成一系列更小的子问题，目标是识别所有非零拉格朗日乘数并丢弃零拉格朗日乘数。由于在SVM的训练过程中，只有少数拉格朗日乘子会变为非零，非零的拉格朗日乘子对应着支持向量。尽管分块法能够有效减少内存占用，但它仍然无法完全解决内存不足的问题，尤其是在面对非常大的数据集时。

SMO算法在1998年由John C. Platt提出，SMO 算法将高维优化问题分解为多个简单的子问题。与之前的方法不同，SMO在每一步都选择解决可能最小的优化问题。对于标准的SVM QP问题，最小可能的优化问题涉及两个拉格朗日乘子，因为拉格朗日乘子必须服从一个线性等式约束。在每一步，SMO选择两个拉格朗日乘子进行联合优化，找到这些乘子的最优值，并更新SVM以反映新的最优值。



训练支持向量机的三种替代方法：分块、Osuna算法和SMO。对于每种方法，都说明了三个步骤。每一步的水平细线代表训练集，而粗盒子代表在这一步被优化的拉格朗日乘子。对于分块，每一步都添加固定数量的示例，而每一步都丢弃零拉格朗日乘子。因此，每一步训练的示例数量趋于增长。对于Osuna的算法，每一步都优化一个固定数量的示例：每一步都在问题中添加和丢弃相同数量的示例。对于SMO，每个步骤只有两个例子进行解析优化，因此每个步骤都非常快。

SMO算法每次只选择两个拉格朗日乘子进行优化，其余保持不变。这种做法通过逐步优化两个变量，将高维优化问题转化为多个一维的优化问题，从而大大降低了计算复杂度。由于拉格朗日乘子的约束条件，优化时可以简化为：

这种方式将高维优化问题转化为多个二元优化问题，便于求解。

**优化流程**：

1.选择两个变量（）：

通过启发式方法选择两个不满足KKT条件的拉格朗日乘子。KKT条件包括原始约束、梯度条件和互补松弛条件，若某个乘子未满足KKT条件，则该乘子存在可优化的空间。

2.计算约束范围：

在优化过程中，拉格朗日乘子 和 需要满足约束条件。为此，计算更新后的乘子必须位于区间[L,H]中，其中

由于约束，优化时需要确保 和 满足一下条件：

其中:

3.更新变量：

使用对偶问题的二次优化公式，更新选中的两个拉格朗日乘子 和 ，使得目标函数的值更接近最优解。

4.更新偏置b：

通过支持向量的条件 更新b的值。

5.重复迭代：

反复选择不满足KKT条件的乘子并更新，直到所有乘子都满足KKT条件，算法收敛，找到最优解。

SMO算法流程手算示例:

假设我们已经选择了两个不满足KKT条件的乘子：和，这两个乘子对应于训练样本和我们将根据SMO算法的步骤来优化这两个乘子。

**计算约束范围：**

根据拉格朗日乘子的约束，我们需要计算出的更新范围，对于本例，假设：

所以的更新范围为。

**计算误差**：

为了进行SMO更新，我们还需要计算误差和。这些误差是目标函数对两个乘子梯度的表示。

假设我们已经计算出了误差和（这些值需要通过模型计算）。

**计算步长：**

为了更新乘子，我们还需要计算步长，这个步长是由两个样本的内积（或核函数）决定的。对于线性核，内积为：

假设内积计算结果为：

那么步长为：

**更新乘子：**

根据SMO的优化公式，我们可以计算新的乘子和：

假设：

接下来，更新：

假设：

**更新偏置b：**

更新后的偏置b可以通过以下公式计算：

其中w是通过拉格朗日乘子计算的权重向量。为了简化计算，假设w和b可以通过支持向量的条件来更新。我们需要更新乘子和计算新的偏置。

#### 2.6 求解w和b得到预测函数并进行预测

在得到拉格朗日乘子以后，我们就可以将拉格朗日乘子带入之前的式子求得w与b，公式如下：

如果有多个支持向量，则需要计算所有偏置项并求得平均值，公示如下：

其中，S代表所有支持向量的集合。

最后，利用求解得到的 w 和 b，构建分类的决策函数：

### 3.SVM算法推导的简单举例

我们依旧使用提取出的甲骨文样例数据进行手算，其中训练集与测试集分别如下：

Filename,Feature\_1,Feature\_2,Feature\_3,Feature\_4,Feature\_5,Feature\_6,Feature\_7,Feature\_8,Feature\_9,Label

\_日1.png,0.72,0.45,0.79,0.66,0.75,0.84,0.84,0.60,0.80 ,日

\_日2.png,0.68,0.65,0.66,0.68,0.69,0.79,0.63,0.64,0.61,日

\_牛1.png,0.78,0.98,0.76,0.69,0.41,0.88,0.93,0.58,1.0,牛

\_牛2.png,0.74,0.77,0.75,0.85,0.41,0.92,0.88,0.63,0.93,牛

\_羊1.png,0.82,0.66,0.73,0.91,0.55,0.95,1.0,0.78,1.0,羊

\_羊2.png,0.74,0.69,0.74,0.88,0.5,0.89,0.82,0.8,0.92,羊

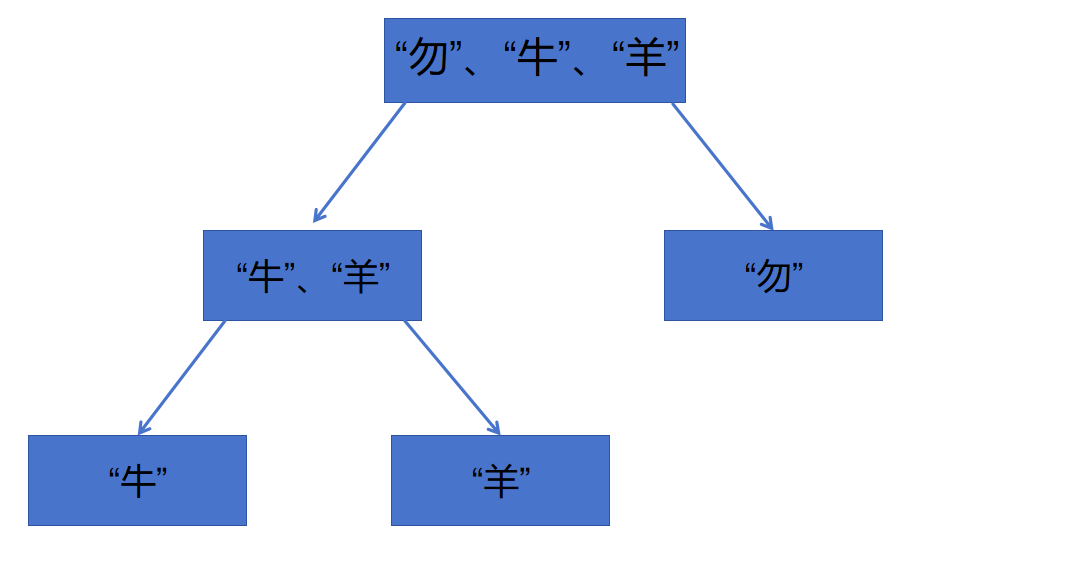
Feature\_1,Feature\_2,Feature\_3,Feature\_4,Feature\_5,Feature\_6,Feature\_7,Feature\_8,Feature\_9

0.72,0.58,0.63,0.93,0.36,0.91,1.0,0.69,1.0

在简单举例推导中，我们使用线性核，即

对于每一类label，我们分别计算支持向量和超平面，采用OvA（一对多）策略。

OvA策略的应用与ID3中树形结构类似，每一次单独判断一个label与其他所有label的超平面，逐步迭代直至最后一个label，在本例中的应用如下图所示：



我们首先假设超平面公式为：

以“日”类别为正类，其余类别为负类进行划分，通过拉格朗日函数对偶问题求解拉格朗日乘子，由于在这个过程中使用SMO算法优化计算较为复杂，我们在此处省略这个过程。

假设我们通过计算得到：

我们使用类似方法继续计算“牛”“羊”的分类边界，得到如下数据：

得到了三个类别的超平面以后，我们可以将预测特征值带入每个超平面公式可得如下：

1.对“日”类：

2.对“牛”类：

3.对“羊”类：

通过对比我们可以发现，其中最大，那么我们就可以认为该预测类别为“羊”。

在这个例子中，我们也可以使用sklearn库简单的进行一个SVM算法的实现，代码如下：

import numpy as np

from sklearn.svm import SVC

# 训练集数据

X\_train = np.array([

[0.72, 0.63, 0.67, 0.65, 0.59, 0.41, 0.99, 0.69, 0.82], # 勿

[0.66, 0.82, 0.98, 0.69, 0.84, 0.86, 0.81, 0.79, 0.91], # 勿

[0.78, 0.98, 0.76, 0.69, 0.41, 0.88, 0.93, 0.58, 1.0], # 牛

[0.74, 0.77, 0.75, 0.85, 0.41, 0.92, 0.88, 0.63, 0.93], # 牛

[0.82, 0.66, 0.73, 0.91, 0.55, 0.95, 1.0, 0.78, 1.0], # 羊

[0.74, 0.69, 0.74, 0.88, 0.5, 0.89, 0.82, 0.8, 0.92] # 羊

])

y\_train = np.array(["勿", "勿", "牛", "牛", "羊", "羊"])

# 等待预测的数据

X\_predict = np.array([[0.72, 0.58, 0.63, 0.93, 0.36, 0.91, 1.0, 0.69, 1.0]])

# OvA策略进行多分类

model = SVC(kernel='linear', decision\_function\_shape='ovr', C=1.0)

//其中‘linear’代表使用线性核，‘ovr’代表使用OvA策略，C代表惩罚系数

model.fit(X\_train, y\_train)

# 预测

prediction = model.predict(X\_predict)

print("预测结果:", prediction)

### 4.SVM算法的代码实现

虽然SVM算法的数学原理略显复杂，但是利用现成的sklearn库实现利用SVM算法解决多分类识别任务并没有那么复杂，接下来我们将利用SVM算法来进行甲骨文识别任务的示例。需要注意的是，SVM仅是一个分类器，不能够直接从图像中提取特征，所以我们要用到其他预训练模型来提取特征，在实例中我们将使用ResNet来提取特征。

import os

import numpy as np

from sklearn.svm import SVC

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

from tensorflow.keras.applications.vgg16 import VGG16, preprocess\_input

from tensorflow.keras.preprocessing.image import load\_img, img\_to\_array

# 数据路径

data\_path = r"Y:\py-torch\Machine\_learing\_pytorch\need" # 主数据文件夹路径

# 1. 数据预处理与特征提取

def load\_data\_with\_vgg16(data\_path):

features = [] # 存储提取的特征

labels = [] # 存储图像对应的标签

# 加载预训练的VGG16模型，移除顶部分类层，保留全局平均池化层

model = VGG16(weights='imagenet', include\_top=False, pooling='avg')

for subfolder in os.listdir(data\_path):

subfolder\_path = os.path.join(data\_path, subfolder)

if os.path.isdir(subfolder\_path):

# 提取子文件夹名称中的标签：下划线后的第一个字符

label = subfolder.split('\_')[1][0]

for file in os.listdir(subfolder\_path):

file\_path = os.path.join(subfolder\_path, file)

if file.lower().endswith(('.png', '.jpg', '.jpeg')): # 确保是图片文件

# 加载图像并预处理

img = load\_img(file\_path, target\_size=(224, 224)) # VGG16需要224x224大小的图片

img\_array = img\_to\_array(img) # 转为numpy数组

img\_array = np.expand\_dims(img\_array, axis=0) # 增加批次维度

img\_array = preprocess\_input(img\_array) # 预处理图片（符合VGG16输入格式）

# 提取特征

feature = model.predict(img\_array) # 输出特征向量

features.append(feature.flatten()) # 将特征向量展平

labels.append(label) # 存储对应的标签

return np.array(features), np.array(labels)

# 2. 加载数据并提取特征

print("使用VGG16提取图像特征...")

X, y = load\_data\_with\_vgg16(data\_path)

print("特征提取完成！")

# 3. 划分训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 4. 使用线性核的SVM进行分类

print("训练SVM模型...")

svm = SVC(kernel='linear') # 定义线性核SVM

svm.fit(X\_train, y\_train) # 训练模型

print("SVM模型训练完成！")

# 5. 测试模型并评估性能

y\_pred = svm.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"模型在测试集上的准确率: {accuracy \* 100:.2f}%")

# 朴素贝叶斯算法

朴素贝叶斯算法的出现，主要是为了解决高维数据分类中的概率估计困难，尤其是在特征之间存在复杂依赖关系时，传统的贝叶斯分类器往往需要大量的样本来准确估计特征的联合分布。

根据条件概率的定义，联合概率可以表示为：

如果我们没有额外的假设，这个联合概率就非常复杂，因为它包含了所有特征之间的条件依赖关系。在实际应用中，这意味着我们需要估计所有特征组合的条件概率，即:

对于每个特征。对于n个特征，计算这种联合概率需要非常庞大的样本量和计算资源。

结合贝叶斯算法我们可以轻易得到贝叶斯算法的时间复杂度为，空间复杂度为，其中，m代表训练集样本数，n代表特征数量，k代表类别数目。朴素贝叶斯算法的时间复杂度为，空间复杂度为（在后文中我们会证明这些复杂度），通常情况下k远小于n，因此，朴素贝叶斯算法

在图像分类任务中，每张图像通常由成千上万的像素值组成，这些像素值通常是相互依赖的。为了简化这一问题，朴素贝叶斯算法假设特征（像素值）在给定图像类别的条件下是独立的，这使得联合概率的计算变得简单，可以将其分解为单个像素在该类别下的条件概率的乘积。

举例来说，在人脸识别任务中，每个像素的灰度值可以单独计算其在不同类别（例如“人脸”与“非人脸”）下的出现概率，而不必考虑像素之间的相关性。尽管这个假设并不完全符合实际情况，因为图像中的像素通常是相关的，但在许多情况下，朴素贝叶斯仍能提供一个有效且高效的分类模型。朴素贝叶斯还特别适合小样本学习，因为它对数据量的需求较少，且计算高效。

尽管特征独立性假设在图像数据中不成立，但其容错能力强，适合处理特征相关性较弱的任务。典型的应用包括物体识别、图像分类等问题，其中朴素贝叶斯能够在较小的样本和复杂的图像数据条件下，快速且有效地进行分类。

### 一.朴素贝叶斯算法的原理与核心思想

贝叶斯公式又被称为贝叶斯规则，是概率统计中的应用所观察到的现象对有关概率分布的主观判断（先验概率）进行修正的标准方法。如果你看到一个人总是做一些好事，则那个人多半会是一个好人。这就是说，当你不能准确知悉一个事物的本质时，你可以依靠与事物特定本质相关的事件出现的多少去判断其本质属性的概率。用数学语言表达就是：支持某项属性的事件发生得愈多，则该属性成立的可能性就愈大。贝叶斯公式中涉及到先验概率、后验概率、条件概率等，具体解释如下。

先验概率：即基于统计的概率，是基于以往历史经验和分析得到的结果，不需要依赖当前发生的条件。

后验概率：则是从条件概率而来，由因推果，是基于当下发生了事件之后计算的概率，依赖于当前发生的条件。

条件概率：记事件A发生的概率为P(A)，事件B发生的概率为P(B)，则在B事件发生的前提下，A事件发生的概率即为条件概率，记为P(A|B)，读作“在B条件下A的概率”。

联合概率：表示两个事件共同发生的概率。A与B的联合概率表示为P(AB),或者P(A,B),或者P（A∩B）。

### 二．朴素贝叶斯的数学原理

**贝叶斯定理**

贝叶斯定理是朴素贝叶斯算法的理论基础，公示如下：

其中，是后验概率，即在已知特征X的情况下，属于类别C的概率;是似然概率，即在类别C已知的情况下，观测到特征X的概率;是先验概率，即类别C的概率;是特征X的概率.

在分类任务中，是一个常数，只需要比较的大小即可。

**朴素假设**

朴素贝叶斯假设特征之间是**条件独立**的，即给定类别C后，各特征相互独立：

根据这一假设，贝叶斯定理可以写为：

其中，是先验概率，是每个特征值在类别C条件下的概率。

由以上公式我们可以知道，计算每个特征值在类别 C 条件下的概率需要遍历训练数据集，时间复杂度为，m代表训练样本数量；由于有 n 个特征，计算需要对每个特征进行上述操作，因此总的时间复杂度为；在分类决策时，需要计算k个类别的后验概率并取最大值，时间复杂度为。综上，朴素贝叶斯算法的时间复杂度仅为。每个特征值在类别 C 条件下的概率表需要存储，每个特征需要的储存空间，所以空间复杂度为。

由空间复杂度与时间复杂度可以看出朴素贝叶斯算法在贝叶斯算法上的巨大改进。

**分类决策**

为了分类，我们比较所有类别的后验概率,选择最大值对应的类别。

**参数估计**

在实际应用中，需要估计和：

先验概率P(C)：

条件概率：

这里引入了拉普拉斯平滑参数，用于避免概率为零的情况。通常。

### 三．朴素贝叶斯推导实例

预测样本的类别y，样本特征为：

训练集中有三个类别：勿、牛、羊。

根据朴素贝叶斯分类器：

由我们的训练集可知，样本总数N = 6，则

假设特征满足正态分布（高斯分布），概率密度函数为：

其中，是类别C下特征的均值；是类别C下特征的方差;代表拉普拉斯平滑值，一般设置为。

**类别“勿”的统计量计算：**

训练数据（日）中的记录：

(0.72,0.63,0.67,0.65,0.59,0.41,0.99,0.69,0.82)

(0.66,0.82,0.98,0.69,0.84,0.86,0.81,0.79,0.91)

计算均值：

特征1（）的均值：

特征1（）的方差：

以此类推，计算所有特征的均值和方差：

|  |  |  |
| --- | --- | --- |
| 特征 | 均值 | 方差 |
| 特征1 | 0.70 | 0.0004 |
| 特征2 | 0.55 | 0.01 |
| 特征3 | 0.725 | 0.004225 |
| 特征4 | 0.67 | 0.0001 |
| 特征5 | 0.72 | 0.0009 |
| 特征6 | 0.815 | 0.000625 |
| 特征7 | 0.735 | 0.011025 |
| 特征8 | 0.62 | 0.0004 |
| 特征9 | 0.705 | 0.009025 |

同理计算“牛”、“羊”两个类别的均值与方差：

|  |  |  |  |
| --- | --- | --- | --- |
| 类别 | 特征 | 均值 | 方差 |
| 牛 | 特征1 | 0.76 | 0.0016 |
|  | 特征2 | 0.875 | 0.11025 |
|  | 特征3 | 0.755 | 0.00025 |
|  | 特征4 | 0.77 | 0.0648 |
|  | 特征5 | 0.41 | 0 |
|  | 特征6 | 0.9 | 0.0016 |
|  | 特征7 | 0.905 | 0.001225 |
|  | 特征8 | 0.605 | 0.0025 |
|  | 特征9 | 0.965 | 0.000625 |
| 羊 | 特征1 | 0.78 | 0.016 |
|  | 特征2 | 0.675 | 0.000225 |
|  | 特征3 | 0.735 | 0.000225 |
|  | 特征4 | 0.895 | 0.0009 |
|  | 特征5 | 0.525 | 0.00125 |
|  | 特征6 | 0.92 | 0.009025 |
|  | 特征7 | 0.91 | 0.0324 |
|  | 特征8 | 0.79 | 0.0001 |
|  | 特征9 | 0.96 | 0.0016 |

单个特征的条件概率计算：

以此类推，计算出所有特征值的条件概率：

将所有条件特征值累乘后乘以先验概率：

重复以上过程，可以计算得到其余特征：

最后经过比较，我们可以得到预测的结果为“勿”。我们可以发现这一次的计算过程无误，但是最终预测结果出现了问题，这是因为第一数据集太小了，可能会导致欠拟合；其次，贝叶斯的独立性假设可能在现实中并不成立；当然也有可能数据分布问题，由于特征方差的影响，“羊”类别训练集中第2、3个特征值得到的特征方差极小，所以对最终结果产生了巨大影响。

在实际训练的过程中，如果遇到这些问题，可以逐个排查并使用例如图像增强等方法来提高准确率。

### 四．朴素贝叶斯算法的代码实现

接下来，我们将实现了一个基于朴素贝叶斯算法的图像分类器，主要步骤包括从指定路径中加载图像数据，将每张图片缩放到统一尺寸并展平为一维向量作为特征，提取文件夹名称中的标签并进行标签编码，然后将数据划分为训练集和测试集。接着使用 sklearn 提供的 GaussianNB 创建朴素贝叶斯分类器，并设置拉普拉斯平滑参数 var\_smoothing=1e-6，对训练数据进行训练，最后对测试数据进行预测并输出分类结果的详细评估报告（包括精度、召回率和 F1 分数）。

import os

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import classification\_report

from skimage.io import imread

from skimage.transform import resize

# 设置图片路径和目标尺寸

base\_path = r"Y:\py-torch\Machine\_learing\_pytorch\need" # 主文件夹路径

target\_size = (64, 64) # 图片统一缩放尺寸

# 初始化特征和标签列表

features = []

labels = []

# 遍历每个子文件夹

for folder in os.listdir(base\_path):

folder\_path = os.path.join(base\_path, folder)

if os.path.isdir(folder\_path):

# 提取标签：子文件夹名称的下划线后第一个字符

label = folder.split('\_')[1][0]

# 遍历子文件夹内的图片

for file in os.listdir(folder\_path):

file\_path = os.path.join(folder\_path, file)

if file.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.tiff')):

# 读取图片并调整到目标尺寸

image = imread(file\_path, as\_gray=True) # 读取为灰度图

image\_resized = resize(image, target\_size, anti\_aliasing=True)

# 将图片展平为一维向量并存储

features.append(image\_resized.flatten())

labels.append(label)

# 转换为NumPy数组

features = np.array(features)

labels = np.array(labels)

# 编码标签为数字格式

label\_encoder = LabelEncoder()

encoded\_labels = label\_encoder.fit\_transform(labels)

# 划分训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(features, encoded\_labels, test\_size=0.2, random\_state=42)

# 创建朴素贝叶斯分类器（带拉普拉斯平滑）

nb\_model = GaussianNB(var\_smoothing=1e-6)

# 训练模型

nb\_model.fit(X\_train, y\_train)

# 预测测试集

y\_pred = nb\_model.predict(X\_test)

# 输出分类报告

print("分类结果：")

print(classification\_report(y\_test, y\_pred, target\_names=label\_encoder.classes\_))

# Random Forest算法（随机森林）

在二十世纪八十至90年代，由于统计学以及计算机科学的进步，机器学习算法得到巨大发展，彼时面对的最大问题是：如何对于线性不可分的数据进行分类？在这一过程中，研究者们提出了大量的算法解决这个问题，包括SVM、决策树等算法在此时崭露头角。其中决策树算法是一个很有趣的算法，但是依旧存在许多问题，最大的问题是十分容易过拟合。为了解决这个问题，在2001年，LEO BREIMAN提出了一个名为Random Forest[[1]](#footnote-0)（Random Forest）的集成学习[[2]](#footnote-1)算法来解决这个问题。

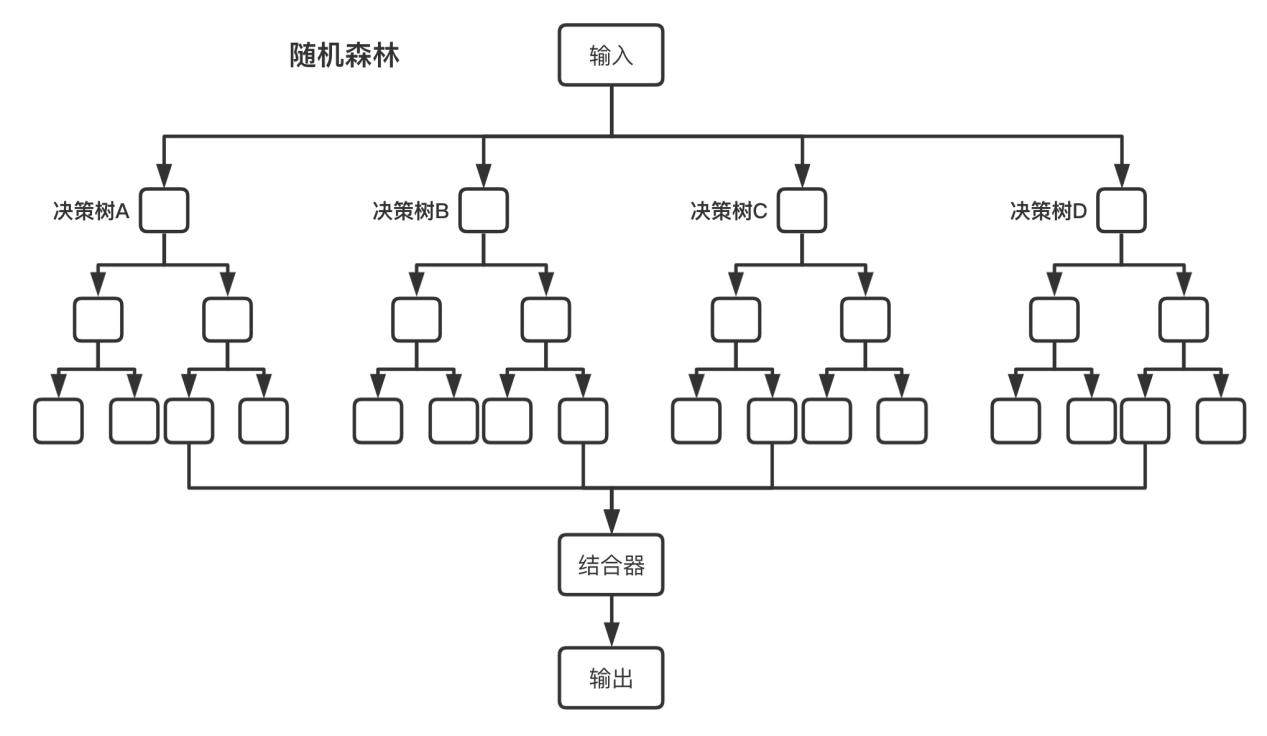


图 1 Random Forest结构

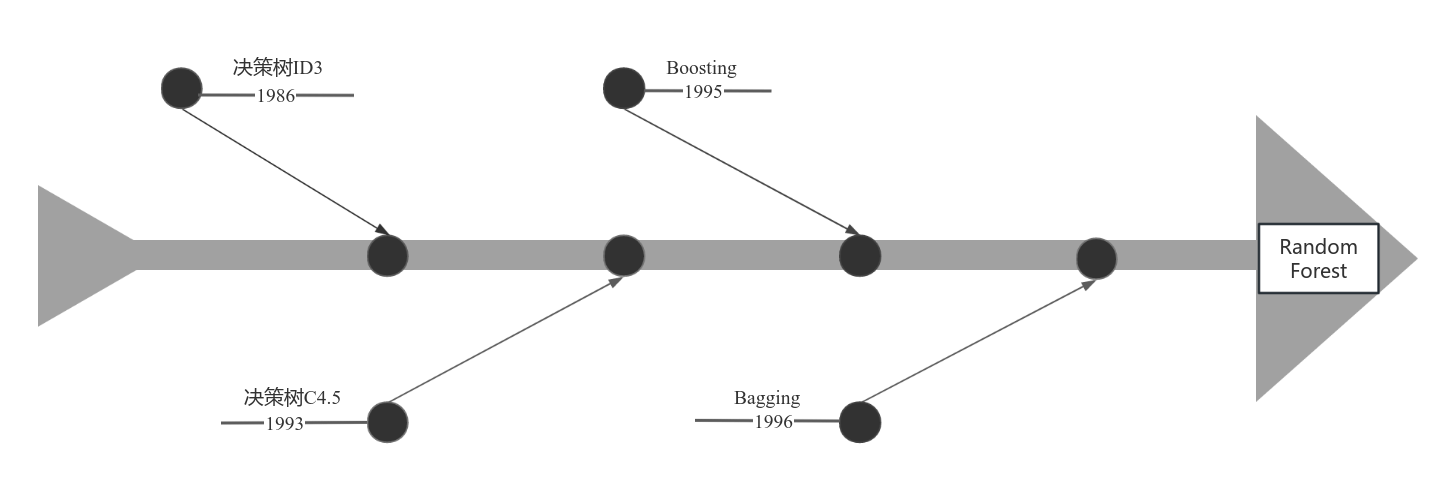


图 2 Random Forest发展历程

## 0.Random Forest前置理论

经过数十年的发展，从最开始的ID3算法，到之后的集成学习Boosting与Bagging算法，这些算法都是拥有巨大的优势，但是同样也存在一些缺陷，我们将逐步简介这些算法并引入Random Forest算法。

### 0.1 决策树C4.5

C4.5算法是ID3的改进版本，解决了ID3的一些缺点。C4.5使用**信息增益比**（Gain Ratio）代替了ID3中的信息增益。信息增益比的公式为：

衡量某个特征A对数据集D进行划分后，划分的“信息量”。其计算公式为：

其中，是数据集中某个属于取值v的比例，是特征A所有可能取值的集合。

衡量在选择特征 A之后，数据集D的信息量减少的程度。信息增益越大，表示选择该特征划分数据集后，结果越纯净。公式为：

其中，是数据集D的熵（不确定性度量），是划分后的子集的熵表示在特征A取值为v时，数据集的纯度；计算公式为：

以上公式中D代表数据集；A代表特征，用于划分数据集D；是数据集中样本总数，数据集D中属于特征A取值v的样本数。

通过引入信息增益比，C4.5克服了ID3偏向选择取值多的特征的问题，并能够更好地处理连续特征。C4.5的主要缺点是：对数据中的噪声较为敏感；对缺失值的处理能力不足。

### 0.2 Bagging算法

Bagging（Bootstrap Aggregating）是一种集成学习方法，它通过对训练集进行有放回的抽样生成多个子集（即“Bootstrap抽样”），并训练多个模型（通常是决策树），最终通过投票或平均的方式汇总模型的结果。Bagging的目标是减少模型的方差，从而提高模型的稳定性。

在Bagging中，模型的训练过程是独立的，每棵树都基于不同的子集进行训练，而预测结果则是这些树的平均值或投票结果。Bagging的公式推导通常依赖于多个模型输出的加权平均或投票决策。

在数学上，假设有M个基学习器每个基学习器由不同的子数据集训练得到），每个基学习器的预测结果是，其中。假设每个基学习器的预测概率是。如果是多数投票，则最终预测的类别为：

其中，是指数函数，当基学习器m预测类别为y时，值为1，否则为0。

假设每个基学习器的误差由偏差和方差组成。根据期望值的线性性质，Bagging通过平均多个基学习器的输出，可以减少方差，但偏差保持不变。具体来说，若单个基学习器的误差为：

这里的**偏差（Bias）**表示基学习器在预测过程中产生的系统性误差，指的是模型预测的期望值与真实值之间的差距。**方差（Variance）**则表示基学习器对不同训练数据集的敏感度，具体来说，就是当训练集发生变化时，基学习器预测结果的波动程度。

### 0.3 Boosting算法

Boosting也是一种集成学习方法，但与Bagging不同的是，Boosting是通过加权组合多个弱学习器来构建一个强学习器。Boosting的核心思想是逐步训练弱学习器，每个新的学习器都会更加关注前一个学习器分类错误的样本。

一个经典的Boosting算法是AdaBoost，它通过以下公式来更新每个训练样本的权重：

其中，是第t轮训练中样本i的权重；代表权重；y代表真实标签；是第t轮学习器对样本i的预测。

Boosting的优势在于能够通过组合弱学习器来提升模型的预测能力，特别是在解决偏差问题上具有显著效果。然而，Boosting也有缺点：它对噪声和异常值较为敏感；容易过拟合。

## Random Forest算法的构建

### 决策树的过拟合证明

我们以ID3决策树为例，来证明决策树易过拟合的问题。

ID3决策树的信息增益[[3]](#footnote-2)定义为：

其中，A是当前考虑的特征，Values(A)是特征A的所有取值，是特征值A取值为v时的子集，||是自己的样本数量，H()是子集的信息熵。

ID3算法倾向于选择取值多的特征（例如类别数多的特征）进行分裂，因为这种特征通常会带来较大的信息增益。但这种选择并不一定是最优的，它可能会导致模型过于复杂，从而产生过拟合。例如，当特征取值过多时，模型可能会记住训练集中的具体样本而不是捕捉数据的普遍规律。

这个现象可以通过偏差-方差权衡[[4]](#footnote-3)来解释：如果我们选择复杂的特征进行分裂，树的深度会增加，模型会对训练数据过度拟合，导致方差增大而偏差减小。

假设我们有训练集与测试集，训练误差与测试误差分别为：

其中代表指示函数，当成立时为1，否则为0.

ID3算法在训练过程中不断选择信息增益最大的特征，因此随着树的深度增加，训练误差趋于零。实际上，当树的深度足够大时，模型能够完全拟合训练集中的每个样本，因此：

然而，随着树的深度增加，模型变得越来越复杂，它开始过度拟合训练数据中的噪声和偶然性。这种过拟合会导致在新数据（测试集）上的表现变差，表现为**测试误差的增大**：

因此，训练误差和测试误差之间的差异逐渐增大，表现为**训练误差小，测试误差大**，这就是过拟合的典型特征。

除此以外，决策树还有高方差（对数据变化敏感）、低偏差（泛化能力差）等问题。

### 1.2 Random Forest的构建

简单来讲，Random Forest算法是一种集成学习方法，通过构建多个决策树并结合它们的预测结果来提高模型的准确性和稳定性。每棵树在训练时使用数据的随机子集和特征的随机子集，这种随机性减少了过拟合并增强了泛化能力。最终，Random Forest通过投票（分类任务）或平均（回归任务）来给出整体预测，且能够有效评估特征的重要性。

#### 1.2.1 集成学习模型

**集成学习**的核心思想是通过组合多个弱学习器（如决策树）来得到一个强学习器，以提高模型的准确性和稳定性。假设我们有一个训练集 ，x与y分别代表输入的特征与标签。

在集成学习中，通过构建多个模型，并将它们的输出结合起来，最终的预测输出为：

或

*[[5]](#footnote-4)*

#### 1.2.2 Random Forest算法

Random Forest是一种集成学习算法，通过生成多个决策树来提高预测准确性和稳定性。每棵树在构建时都使用了两种类型的随机性：**数据随机性**和**特征随机性**。

##### 1.2.2.1 数据子集（Bootstrap采样）

每棵树在训练时使用的数据集不是整个训练集，而是通过 **Bootstrap采样** 随机选取的子集，即从训练集中 **有放回地** 采样m个样本，形成每棵树的训练集:

其中，m是每棵树的样本数量，通常m与原始数据集大小相同。

##### 1.2.2.2 特征随机性（随机选择特征）

在每棵树的构建过程中，**每个节点的分裂**并不是通过使用所有特征来选择最优的分裂特征，而是从所有特征中随机选择一个小子集（其中，p是特征总数），然后在这个子集上选择最优的分裂特征。

因此，对于节点t，每次分裂时，算法在特征子集上计算每个特征的信息增益或基尼指数，并选择带来最大增益的特征进行分裂。

##### 1.2.2.3 集成预测

当训练完成后，对于新样本x，Random Forest的预测是由每棵树的预测结果的**投票**或**平均**结果决定的。

最终预测公式即为：

或

*[[6]](#footnote-5)*

## Random Forest的误差分析及准确性描述[[7]](#footnote-6)

### 2.1 Random Forest的收敛

#### 2.1.1 边际函数的定义

Random Forest由多个决策树组成，每棵树都基于数据的随机子集进行训练。在构建过程中，每个节点的特征选择也是随机的，而不是使用所有特征。最终，Random Forest的预测结果通过对所有决策树结果的投票或平均来得出。通过这种方式，Random Forest可以减少过拟合，提高模型的泛化能力。

给定一个分类器的集合h1(x)，h2(x)，……，hK (x)，并使用从随机向量Y，X的分布中随机抽取的训练集，将边际函数定义为

其中，X代表真实样本；Y代表真实标签；代表第k棵决策树对X的预测值；表示对所有树的预测结果进行平均，计算正确预测的比例，换句话讲，它表示 **Random Forest中正确分类为Y的树的平均数量；**表示对每个错误类别，计算预测为该类别的树的平均比例，然后选择其中最大的比例。

总的来讲，边际函数即通过计算 **正确类别的预测比例** 和 **最大错误类别的预测比例** 之间的差异，得到了**边际函数**。**如果边际函数的值较大**，说明大部分树都预测了正确的标签Y，且错误标签的预测比例较低，模型对样本的分类信心较强；**如果边际函数的值较小**，说明模型对样本的分类较为模糊，可能有一些树预测为错误类别，模型的不确定性较高。

那么这个边际函数是如何推导得到的呢？接下来我们将逐步推导得到这个边际函数。

#### 2.1.2 边际函数的推导

2.1.2.1 正确类别的平均预测比例：

首先，考虑到**正确类别**的预测比例，假设在v棵树中，第k棵树预测为Y时，指示函数)的值为1，否则为0。因此，对于所有v棵树，正确类别Y的预测比例可以表示为

这部分是所有树中预测正确类别Y的比例。也可以理解为Random Forest对样本 X属于类别Y的信心度。

##### 2.1.2.2 错误类别的预测比例：

接下来，我们考虑对于**错误类别**  的预测比例。假设对于某个错误类别j，第k棵树预测为j时，指数函数的值为1，否则为0。因此，对于类别j，预测错误的平均比例可以表示为：

##### 2.1.2.3 取最大错误类别的预测比例：

我们希望找到**最不确定的错误类别**，即最大误分类的比例。所以，我们对于所有的错误类别，计算其预测比例并取最大：

##### 2.1.2.4 计算边际函数：

现在我们可以通过计算两个部分的差异来定义边际函数：

**第一部分**：表示模型在所有树中，预测为正确类别Y的平均比例，即模型对该样本正确分类的信心。

**第二部分**：表示模型在所有树中，预测为任何错误类别j的平均比例，取其中最大的那个，衡量模型对错误分类的信心。

因此，边际函数可以定义为：

#### 2.1.3 泛化误差

我们有了模型之后，需要评估模型的性能，模型的泛化能力[[8]](#footnote-8)是一个极其重要的指标。而我们用来评定泛化能力的值，成为泛化误差。

泛化误差一般定义为：

其中，是模型对输入样本X的预测结果；Y是X的真实标签；表示期望，即在所有可能的输入样本和真实标签上进行平均。

**定义泛化误差**是为了避免仅通过训练集的表现来过度评估模型，而忽略模型在真实场景中的表现。泛化误差的高低直接决定了模型是否能有效地推广到新的、未见过的数据。

在Random Forest算法中，我们将泛化误差定义如下：

我们可以理解这个公式为：计算**边际函数（Margin Function）**小于零的概率。边际函数量化了模型对某个样本X属于真实类别Y的信心。边际函数小于零，意味着模型对样本X分类的信心较低，或者在错误类别上的信心较强。

在Random Forest中

这个公式表明第k棵决策树的预测是基于输入的X以及参数得到的，其中是每一棵树的参数集合，包括但不限于该树的**训练数据子集**、**树的结构**（分裂规则、树的深度等），以及**特征选择策略**（每次分裂时选取的特征子集）。

在Random Forest中，Theorem 1.2描述为：随着树的数量增加，模型的预测会趋于稳定，并且最终的泛化误差将会收敛到某个常数值。我们将利用大数定律[[9]](#footnote-9)以及树的结构两方面来证明这一定律。

引入指标函数：

首先，我们用**指标函数**来描述每棵树的预测。假设第n棵树的预测归于类别j时，使用指标函数表示为;

这个函数值为1当且仅当第n棵树预测x属于类别j，否则为0.

定义收敛目标：

我们希望证明以下表达式的平均值收敛到真实概率：

这个式子表示，随着树的数量N增加，所有树对样本x的投票平均值会收敛于真实的类别概率。

类别划分;

对于固定的训练集和固定的树结构Θ，我们定义类别j下的所有样本集合为：

这些样本集合可以被看作是**超矩形**[[10]](#footnote-10)**(**hyper-rectangles)的并集。这个划分有一个关键的性质：对每个预测，这些集合数量是有限的，最多只有K个超矩形（即可能的类别划分），我们把这些集合表示为.

定义：

我们定义一个函数，它是一个类别标号，表示哪些x属于第k个超矩形。具体可表示为：

这个定义让我们能够描述每棵树预测结果所对应的区域。

引入：

接下来，我们定义为前N次实验中，的次数。即统计每个超矩形中有多少棵树的预测属于类别j。

这意味着是对每个超矩形的树数量的计数。

使用大数定律：

根据大数定律，对于每一个k，在时会收敛到它的期望：

即随着树的数量增多，每个类别k的预测概率会收敛到真实的类别概率。

定义收敛的平均值：

通过之前的定义，我们可得到：

这个式子是所有树对类别j投票的平均值。它表示，随着树的数量增多，每个类别的投票会按其真实概率加权求和。

得到收敛结论：

根据大数法则和加权平均的收敛性，最终我们得到：

这个结论即为，即随着树的数量增多，Random Forest的预测会收敛于真实的类别概率。

结合以上的证明步骤，我们已经证明了，随着树的数量增加，Random Forest的平均预测会几乎确定地收敛到期望值，即收敛到真实的类别概率 。

### 2.2 树的强度与相关性

对于Random Forest，可以根据两个参数推导出泛化误差的上界，这两个参数是对单个分类器的准确性和它们之间的依赖性的度量。这两者之间的相互作用为理解Random Forest的工作原理奠定了基础。这一部分是基于Amit and Geman（1997）[[11]](#footnote-11)分析得到的。

由上文可知，边际函数可以表示为：

*[[12]](#footnote-12)*

对于分类器，我们定义其分类强度为：

[[13]](#footnote-13)

其中，表示**边际函数的期望值**，是对所有可能样本X和标签Y上边际函数值的平均。换句话说，集合分类器的强度s表示了在所有样本中，分类器做出正确预测的平均信心。

根据**切比雪夫不等式[[14]](#footnote-14)**，我们可以得到：

什么是切比雪夫不等式？为什么这里要使用切比雪夫不等式呢？使用切比雪夫不等式的意义或目的又是什么呢？

**切比雪夫不等式**是一种非常有用的概率不等式，它提供了关于随机变量偏离其期望值的上界。具体来说，对于任何随机变量X，如果X的期望是 和方差，那么对于任何,都有：

换句话说，切比雪夫不等式告诉我们，随机变量X偏离其期望值 的概率上界是由其方差和偏离的程度k来决定的。这个不等式在概率分布不完全知道的情况下尤其有用，因为它不依赖于分布的具体形式，只需要方差和期望值。

**那么为什么可以使用切比雪夫不等式呢？**在我们讨论**Random Forest**中的**边际函数**时，它是一个随机变量，表示分类器对样本X正确分类的信心。边际函数的方差描述了**分类器的预测信心**在多个样本上的波动情况。因此，边际函数的方差反映了分类器预测的稳定性。

为了分析泛化误差，我们想知道模型在面对新样本时的误分类概率。理论上，泛化误差应该与 模型的稳定性 和 边际函数的变化 之间有一定关系。因为如果边际函数的方差较大，表示模型的预测结果波动较大，从而导致不稳定的预测，进而增加泛化误差。

切比雪夫不等式为我们提供了这样一个工具，通过它可以将边际函数的方差与泛化误差的关系绑定起来，进而用边际函数的方差来界定误差的上界。

**使用切比雪夫不等式的意义或目的又是什么呢？**使用切比雪夫不等式可以将泛化误差与边际函数方差来联系在一起，即为：

其中，是泛化误差，表示分类器在未知数据上的误分类概率；是边际函数的方差，衡量了分类器对不同样本的预测信心的波动性；s是集合分类器的强度，表示模型对正确分类的信心。

得到泛化误差与边际函数方差之间的关系之后，我们便可以通过最可能的错误分类来表达边际函数。

LEO BREIMAN通过定义了一个最可能的错误分类来重新表达边际函数。假设给定输入样本X和标签Y，最可能的错误分类是指预测错误的类别j中概率最大的那个，即：

在此基础上，边际函数可以表示为：

总的来讲，Definition 2.1定义了Random Forest分类器的**边际函数**，即正确分类的概率与最大错误分类概率之间的差异，并通过边际函数的期望值定义了**集成分类器的强度**。该定义为后续的误差分析提供了基础，证明了泛化误差与边际函数的**方差**之间的关系，且**强度**和方差决定了分类器的整体性能。

根据Definition 2.2，我们定义原始边际函数为：

*[[15]](#footnote-15)*

因此是关于的的期望值，对于任何函数f，有恒等式：

当，独立且具有相同分布时成立，这意味着

这一公式的含义是边际函数的平方（即期望的平方）等于对两个不同的树参数，下的原始边际函数的**协方差期望**。

根据协方差的定义我们可得：

因此，边际函数的协方差可以表示为：

假设在给定参数，的情况下，边际函数之间存在一定的**相关性**。这个相关性[[16]](#footnote-16)是通过协方差来衡量的，该相关性可以通过标准化的协方差[[17]](#footnote-17)来表示：

由之前推导的，我们可以得到：

我们假设的均值为，即：

这意味着，**相关性**的期望值是一个常数，可以用它来简化公式。通过引入这个期望，可以得到式子：

对于随机变量，我们可知标准差的值等同于方差开根，因此，标准差的乘积可以表示为：

由于对树的期望是已知的，因此我们可以将期望操作应用到标准差的乘积中。通过期望值的分配性质，我们可以写出：

在假设**样本独立同分布**（i.i.d.）的情况下，标准偏差的期望值是相同的，因此我们有：

将上式结果带入我们可得：

接下来我们需要讨论三个关系：树方差与边际函数期望的关系；边际函数期望的平方与树方差的关系；与之间的关系。通过这三个关系结合泛化误差与方差，我们将会得到一个关于边际函数方差的**上界**，并通过树的方差来限制模型的**总体不确定性**，这为理解Random Forest的**稳定性**和**泛化能力**提供了理论依据。

**树方差与边际函数期望的关系：**

对于之前关于的不等式，其中表示树的方差的期望值。这个期望值代表的是不同树的预测不确定性的平均水平。在Random Forest中，每棵树的输出通常是由不同的训练集和参数配置决定的，因此它们的预测结果会有所不同。假设我们在Random Forest中有很多棵树，它们的方差是，那么整个森林的方差可以通过这些单棵树的方差的期望来表示。

**边际函数期望的平方与树方差的关系：**

针对边际函数期望的平方，它表示**正确分类的概率与最可能错误分类的概率之差的平方**。这个期望值描述了模型对样本X和标签 Y分类结果的**信心**，即在大多数树上的正确分类的概率与最可能的错误分类概率之间的差异。

这里，我们需要注意的是，这个期望值的平方有助于描述Random Forest分类器的 **强度（confidence）**，即模型正确分类的能力。因此，边际函数期望的平方反映了树分类结果的稳定性。

**与之间的关系[[18]](#footnote-18)：**

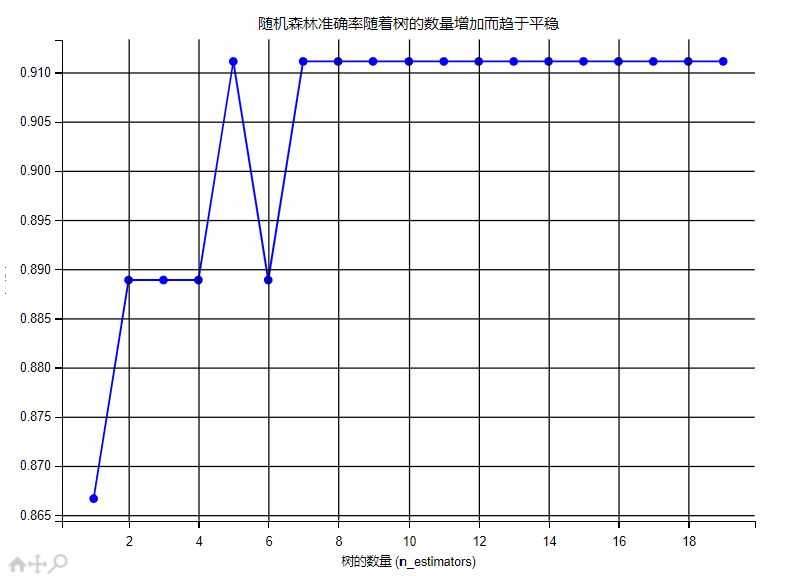
公式中是集合分类器的强度平方的常数，表示了分类器**稳定性**的下界。具体地，s越大，分类器正确分类的概率越大。对于一个给定的分类器而言，描述了**错误分类的概率**和**误差度量**。通过边际函数期望值的约束，我们可以得出 **泛化误差**的上界，进一步推导出**模型的性能约束**。

综上，我们可以得到式子：

通过对该公式中的方差约束进行优化，我们可以通过调整**树的数量**、**深度**或**特征选择策略**来减少方差，从而提高模型的泛化能力。同时，该公式为Random Forest模型的**误差度量**提供了数学上的界限，使得研究人员可以在理论上评估模型的性能，进一步为参数调整和模型优化提供依据。

结合以下三个公式;

我们可以得到**Theorem 2.3——泛化误差的上界被定义为：**



### 2.3 总结

根据2.1与2.2小节，我们得到了泛化误差的上界与树间相关性、树的强度之间的关系。在集成学习中，偏差-方差权衡是一个极其重要的步骤，因为集成方法的目标之一是通过有效地平衡偏差和方差来提高模型的**泛化能力**。

在随机森林这一算法中，偏差-方差权衡这一步骤被我们等价为了树的强度（偏差）与树间关系（方差）的权衡。

## OOB估计（袋外估计）

在随机森林算法中，使用的是Bootstrap自主采样法，从原始数据集D中又放回的抽取。Bootstrap的关键特点是从训练集D中随机抽取样本，但每个样本可以被多次抽取，也可以完全不被抽取。

假设我们原始样本总数为n，对于某一样本，其每一次被选中的概率即为，由于抽样是有放回的，每个样本被“排除”不被选择的机会是独立的。所以对于每个样本在一轮抽样中不被选择的概率是：

假设我们从训练集中抽取n个样本，并且每次抽样是独立的。每个样本不被选中的概率是 ，那么该样本在所有n次抽样中都没有被选中的概率是：

当时，设未知数L可得式：

取对数可得：

使用自然对数的泰勒展开可得：

代入前式可得：

因此我们可知，在随机抽取时，大约有36.8%的数据在每棵树的训练集中未被选中。这些样本被称为该树的OOB样本（袋外样本）。

这些数据并不是没作用的，这些OOB样本都被用作OOB估计了。OOB估计通过利用每棵决策树在训练过程中未被使用的样本，提供了一种无需额外验证集即可评估模型泛化能力的有效方法。它能够无偏地估计随机森林的误差，帮助监控模型性能，并为模型优化提供指导，同时避免了数据的浪费。

OOB误差率的公式定义为：

其中，n是训练集的样本数；是样本的OBB预测值；是的真实标签。对于每个样本，其OOB预测由以下步骤计算：

1.收集所有将样本作为袋外样本的决策树的预测结果。

2.对这些树的预测结果进行多数投票（分类问题）或平均（回归问题），得到最终的 OOB 预测。

## 4.基尼指数

基尼指数[[19]](#footnote-19)（Gini Index），又称基尼不纯度（Gini Impurity），是衡量数据集纯度的一种指标。它由意大利统计学家科拉多·基尼（Corrado Gini）在1912年提出，最初用于衡量收入分配的不平等程度。在机器学习中，基尼指数被广泛应用于分类问题中，特别是在决策树和随机森林算法中，用于评估节点的纯度和选择最佳划分特征。

### 4.1基尼指数推导

基尼指数衡量的是在一个节点内，两个随机选取的样本属于不同类别的概率。换句话说，它表示的是节点的不纯度或混乱程度。基尼指数越低，节点的纯度越高；反之，基尼指数越高，节点的混乱程度越大。

假设在某个节点内有N个样本，分布在C个不同的类别中。每个类别i的样本数为，则：

选择两个不同的样本，且这两个样本都属于类别i的概率为：

为了简化计算，我们假设N极大，那么，因此可得：

两个样本属于不同类别的概率 则为：

基尼指数定义为两个随机选取的样本属于不同类别的概率，因此：

### 4.2基尼指数直观理解

纯节点：如果一个节点内所有样本都属于同一类别，即且，则基尼指数为：

这表示节点完全纯净，没有任何混乱。

**完全不纯的节点**：在二分类问题中，如果两个类别的样本数量相等，即 ，则基尼指数为：

这表示节点具有最大的混乱程度。

## 5.实例推导Random Forest算法

我们的训练集与预测集分别如下：

Filename,Feature\_1,Feature\_2,Feature\_3,Feature\_4,Feature\_5,Feature\_6,Feature\_7,Feature\_8,Feature\_9,Label

\_日1.png 0.72,0.45,0.79,0.66,0.75,0.84,0.84,0.60,0.80,日——S1（编号）

\_日2.png 0.68,0.65,0.66,0.68,0.69,0.79,0.63,0.64,0.61,日——S2

\_牛1.png,0.78,0.98,0.76,0.69,0.41,0.88,0.93,0.58,1.0,牛——S3

\_牛2.png,0.74,0.77,0.75,0.85,0.41,0.92,0.88,0.63,0.93,牛——S4

\_羊1.png,0.82,0.66,0.73,0.91,0.55,0.95,1.0,0.78,1.0,羊——S5

\_羊2.png,0.74,0.69,0.74,0.88,0.5,0.89,0.82,0.8,0.92,羊——S6

Feature\_1,Feature\_2,Feature\_3,Feature\_4,Feature\_5,Feature\_6,Feature\_7,Feature\_8,Feature\_9

0.72,0.58,0.63,0.93,0.36,0.91,1.0,0.69,1.0

根据以上数据，我们接下来的推导过程中我们将构建一个只包含3棵树的随机森林，每个节点划分时选取3个作为候选，使用Bootstrap有放回抽样构建每棵树的训练集，对未出现在某棵树Bootstrap样本中的数据作为OOB数据计算OOB误差，尝试尽可能纯化节点。

### 5.1 树的构建与预测

#### 5.1.1 构建第一棵树

Bootstrap样本抽取（从S1~S6中有放回抽6次）

举例（模拟随机）：

抽取结果：S2, S4, S1, S5, S4, S3

总计6个样本（加权计数），S6未出现。

OOB样本（对Tree1而言）：S6

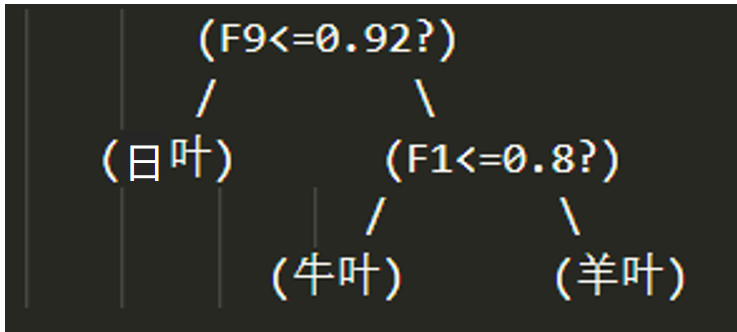
类分布（含重复计数）：日=2 (S1,S2)，牛=3 (S3,S4×2)，羊=1 (S5)

初始基尼（Gini）指数：

**根节点划分**：从9个特征中随机选3个特征，假设抽中(F2,F7,F9)。

对(F2,F7,F9)分别尝试不同划分,最终找到最优划分在F9上，阈值F9=0.92可最大减少基尼不纯度（减少约0.3611）。

最终模型即为：



OOB估计(Tree1)：OOB样本：S6(羊)

预测S6：

F9=0.92正好等于阈值0.92（设定≤则左） -> 左边是勿叶子

预测：勿

真实：羊

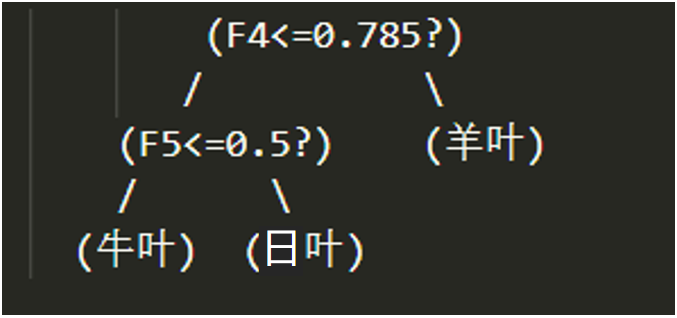
Tree1对OOB样本S6预测错误

#### 5.1.2 构建第二棵树

**Bootstrap样本**：S5, S1, S6, S3, S5, S2 类计数：勿=2(S1,S2),牛=1(S3),羊=3(S5×2+S6) OOB: 未出现S4，所以S4是Tree2的OOB样本。

随机选取3个特征（假设F2,F4,F8）找最优划分。计算后发现最佳划分在F4=0.785处，可以大幅降低Gini（约0.3889的降低）。

最终模型即为：



**OOB(Tree2)**：  
 OOB样本：S4(牛)  
 预测S4：  
 F4=0.85>0.785走右边 -> 羊叶

预测：羊  
 真实：牛

Tree2对OOB样本S4预测错误。

#### 5.1.3 构建第三棵树

Bootstrap样本： 例：S6, S4, S2, S6, S1, S3

类计数：勿=2(S1,S2),牛=2(S3,S4),羊=2(S6×2)，各类均衡。

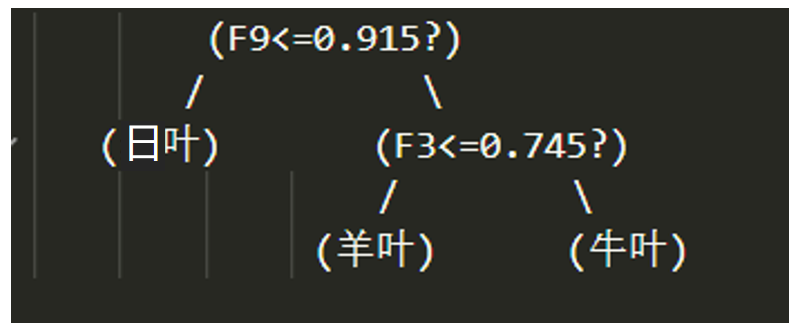
OOB：未出现S5，所以S5是Tree3的OOB样本。

初始Gini=0.6111

随机选3特征(假设F1,F2,F9)，寻找最优划分。

找到最佳在F9=0.915附近可显著降低不纯度（约0.2778降低）。

最终模型为：



OOB(Tree3)：

OOB样本：S5(羊)

预测S5：

F9=1.0>0.915去右

F3=0.73≤0.745去左 -> 羊叶

预测：羊

真实：羊

Tree3对OOB样本S5预测正确。

#### 5.1.4 OOB错误率估计

Tree1 OOB: S6预测错 (1错)

Tree2 OOB: S4预测错 (1错)

Tree3 OOB: S5预测对 (0错)

共有3个OOB预测(各树OOB各1样本)，共2错：

#### 5.1.5 对样本的预测

测试样本特征：

**使用Tree1:**

检查右子树

右子树用左侧为牛叶

Tree1预测：牛

**使用Tree2:**

右侧为羊叶

Tree2预测：羊

**使用Tree3:**

右侧

右侧左叶为羊

Tree3预测：羊

经过多数投票，最终的预测结果即为：羊。OOB误差约为66.67%。OOB误差过高，意味着该模型泛化能力差，可能存在过拟合的风险，这时候就可以选择增加树或者使用其他方法来避免过拟合。

### 5.2 树间关系与树的强度的计算

以下是我根据Breiman在随机森林论文中关于树的强度（strength）和相关度（correlation）的定义，对我们上一轮构建的三棵树进行的一个大致的强度和相关度计算。因为我们的训练数据集非常小，只有六条数据，而且其中一棵树在训练集上的准确率达到了100%，这就可能导致一些统计指标，比如方差，变为零，进而在计算相关度时产生问题。实际上，这么小的数据量并不适合我们精确地估算这两个参数，但在这里，我还是会根据定义试着做一个示范性的计算。

#### 5.2.1 树的强度计算

由之前的推导可得，树的强度s即为边际函数的期望，定义式为：

在上一轮计算中，我们得到了三棵树对于训练集的预测结果：

**训练集样本及真实标签：**

(勿),(勿), (牛), (牛), (羊), (羊)

**各树对训练集的预测正确性（1为正确，0为错误）：**

Tree1：

（对预测错误）

Tree2：

（对预测错误）

Tree3：

（全对）

**汇总正确率：**

T1正确5/6

T2正确5/6

T3正确6/6

**计算Margin：**

对每个样本，随机森林最终投票结果为3棵树投票。如果该样本的真实类别为Y，对应样本的margin计算如下：

(标签=勿)：三棵树预测全部正确为勿

(勿)：同

(牛)：三棵树全部预测牛

(牛)：T1预测牛正确，T2预测羊错误，T3预测牛正确

(羊)：三棵树全预测羊

(羊)：T1预测勿错，T2预测羊对，T3预测羊对

**计算强度s：**

#### 5.2.2 计算树间相关度

**编码类别标签为数值：**

勿 = 0

牛 = 1

羊 = 2

**构建每棵树的预测向量：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **样本** | **真实标签** | **Tree1** | **Tree2** | **Tree3** |
|  | 日 | 0 | 0 | 0 |
|  | 日 | 0 | 0 | 0 |
|  | 牛 | 1 | 1 | 1 |
|  | 牛 | 1 | 1 | 1 |
|  | 羊 | 2 | 2 | 2 |
|  | 羊 | 1 | 2 | 2 |

**计算每对树之间的皮尔逊系数：**

**Tree1 vs. Tree2**

Tree1预测向量：0,0,1,1,2,1

Tree2预测向量：0,0,1,1,2,2

计算步骤：

**均值计算：**

**协方差计算：**

每个样本的乘积和：

总和：

可得：

**方差计算：**

**相关系数计算：**

计算步骤示例如上，根据这个步骤，我们可以计算出Tree1与Tree3的相关度为0.563，Tree2与Tree3的相关度为1，那么森林的树间相关性即为：

### 5.2.3 总结

根据之前的计算，我们可以知道，这个由三棵树构成的简单森林中，树的强度与树间相关性分别为0.7778与0.708，根据这两个数值与之前得到的OOB估计我们可以分析出，这棵树可能出现了过拟合，在这种情况下，我们可以选择增强数据或者继续增加森林中树的数量来提高泛化能力。

## 6.基于sklearn的Random Forest实验

Random Forest算法如今已经发展的十分成熟了，我们可以直接从sklearn库中调用这个算法来完成我们的任务，接下来我们将给出示例代码。

import os

import glob

from PIL import Image

import numpy as np

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

# 数据目录

data\_dir = r"Y:\py-torch\Machine\_learing\_pytorch\need\_tapian"

# 为了统一输入特征，我们先定义将图片resize到固定尺寸，比如64x64

img\_width, img\_height = 64, 64

# 准备存储数据和标签的列表

X = []

y = []

# 遍历数据集文件夹下的每个子文件夹

# 每个子文件夹代表一个类别

for folder\_name in os.listdir(data\_dir):

sub\_dir = os.path.join(data\_dir, folder\_name)

# 判断是否为文件夹

if os.path.isdir(sub\_dir):

# 子文件夹的类别标签解析逻辑：

# 子文件夹名称中下划线后第一个汉字为类的标签

# 假设文件夹名格式类似 "someprefix\_牛abc"

# 则用split('\_')取下划线后部分，再取第一个字符作为标签

if '\_' in folder\_name:

label\_part = folder\_name.split('\_', 1)[1] # 取下划线后面的所有部分

if label\_part: # 确保存在此部分

label = label\_part[0] # 第一个汉字即为标签

else:

# 如果没有下划线后的内容，这里可以根据需求进行处理

# 为了安全起见，给个默认标签或跳过

continue

else:

# 如果没有下划线，跳过或自定义标签规则

continue

# 查找子文件夹中所有png图片

image\_files = glob.glob(os.path.join(sub\_dir, '\*.png'))

# 读取并处理每张图片

for img\_path in image\_files:

# 打开图片

with Image.open(img\_path) as img:

# 将图片转换为RGB格式（以防灰度等情况）

img = img.convert('RGB')

# 将图片缩放到固定尺寸

img = img.resize((img\_width, img\_height))

# 转换为numpy数组

img\_array = np.array(img)

# 将图像展平为一维向量 (64x64x3) -> (12288,)

# 这里的特征只是RGB像素值，不进行额外特征提取

img\_flat = img\_array.flatten()

X.append(img\_flat)

y.append(label)

# 转换为numpy数组

X = np.array(X)

y = np.array(y)

# 数据分割为训练集和测试集，假设8:2比例

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 初始化随机森林分类器

# 设置oob\_score=True以启用袋外估计

rfc = RandomForestClassifier(n\_estimators=200, oob\_score=True, random\_state=42)

# 训练模型

rfc.fit(X\_train, y\_train)

# 使用模型对测试集进行预测

y\_pred = rfc.predict(X\_test)

# 计算测试集准确度

accuracy = accuracy\_score(y\_test, y\_pred)

# OOB分数（袋外估计）

oob\_score = rfc.oob\_score\_

# 打印结果

print("测试集准确率：", accuracy)

print("OOB估计：", oob\_score)

**超参数讲解：**

n\_estimators：随机森林中树的数量，增加树的数量可以提高模型准确率，但会增加计算成本。

max\_depth：每棵决策树的最大深度，控制树的生长，防止过拟合。

min\_samples\_split：内部节点分裂所需的最小样本数。

min\_samples\_leaf：叶节点所需的最小样本数。

max\_features：每次分裂时要考虑的最大特征数。

bootstrap：是否使用有放回的随机采样训练数据来构建每棵树。

1. Breiman, L. (2001). Random forests. \**Machine Learning*\*, 45(1), 5-32. [https://doi.org/10.1023/A:1010933404324](mailto:https:/doi.org/10.1023/A:1010933404324) [↑](#footnote-ref-0)
2. 集成学习：它通过组合多个基学习器（通常是同一类型的模型，如决策树、支持向量机等）来提高整体模型的性能和稳定性。集成学习的核心思想是：多个“弱”学习器的组合，能够比单个“强”学习器更好地进行预测，从而降低偏差和方差，提高准确度和泛化能力。 [↑](#footnote-ref-1)
3. 信息增益：用于衡量某一特征变量对于分类任务所能提供的信息量，即该特征变量在区分不同类别时的有效性。信息增益的基本思想是通过比较在使用某个特征进行划分前后的信息熵（或称为不确定性）的减少量来评估该特征的重要性。 [↑](#footnote-ref-2)
4. 偏差-方差权衡指的是在尝试降低模型的偏差（提高模型复杂度，提升拟合能力）时，模型的方差往往会增加（模型变得对训练数据敏感，易于过拟合）；相反，在尝试降低模型的方差（使模型变得简单，泛化能力提升）时，偏差可能会增加（模型不能充分拟合训练数据，容易欠拟合）。 [↑](#footnote-ref-3)
5. 式中mode为统计学术语，即为投票最多的值。 [↑](#footnote-ref-4)
6. 式中mode为统计学术语，即为投票最多的值。 [↑](#footnote-ref-5)
7. 这一部分内容基于论文而来。Breiman, L. (2001). Random forests. \**Machine Learning*\*, 45(1), 5-32. [https://doi.org/10.1023/A:1010933404324](mailto:https:/doi.org/10.1023/A:1010933404324) [↑](#footnote-ref-6)
8. 泛化能力：是一个模型在训练数据之外的 未知数据 上表现的能力，也就是说，模型对新数据的预测能力。泛化能力与鲁棒性有一定差别，鲁棒性是指模型对 扰动、噪声或不确定性 的抵抗能力，也就是模型在输入数据中存在噪声、错误或变化时，依然能够保持较好的性能。两者有较大区别，避免混淆 [↑](#footnote-ref-8)
9. 大数定律（SLLN）：在概率论中表明，对于一系列独立同分布（i.i.d.）随机变量，随着样本数量的增加，样本均值几乎肯定会收敛到它们的期望值。 [↑](#footnote-ref-9)
10. 超矩形：超矩形 是一个高维空间的区域，其维度与数据的特征数相同。每个超矩形由特征的范围来定义（例如，特征 1 的值在某个区间内，特征 2 的值在另一个区间内）。在二维空间中，超矩形就是一个矩形区域；在三维空间中，超矩形就是一个长方体区域；在高维空间中，它是一个超长方体。 [↑](#footnote-ref-10)
11. Amit, Y. & Geman, D. (1997). Shape quantization and recognition with randomized trees. Neural Computation,

    9, 1545–1588. [↑](#footnote-ref-11)
12. 这里的边际函数之所以与之前定义的不同，这里的边际函数为期望边际，是边际函数的期望值，表示所有树（集成模型）在多次采样下的期望分类信心。 [↑](#footnote-ref-12)
13. 集合分类器的强度s是Random Forest分类器中一个非常重要的指标。主要表现在：衡量分类器的信心；决定分类性能的稳定性；关联误差和方差（通过切比雪夫不等式推导）；强度与模型的稳定性。 [↑](#footnote-ref-13)
14. 统计学知识。 [↑](#footnote-ref-14)
15. 这一边际函数与之前两个又有不同，此为原始边际函数，用于衡量单棵树的分类结果，表示模型是否正确预测并且是否错误预测为最有可能的错误类别。 [↑](#footnote-ref-15)
16. 该相关性称之为皮尔逊相关系数( Pearson correlation coefficient），是用于度量两个变量X和Y之间的相关（线性相关），其值介于-1与1之间。定义为两个变量之间的协方差与标准差之商。 [↑](#footnote-ref-16)
17. 下式中，cov代表协方差，sd代表标准差。 [↑](#footnote-ref-17)
18. s为分类器强度，定义式为： [↑](#footnote-ref-18)
19. Gini, C. (1912). Variabilità e mutabilità. Rome: Tipografia Nazionale. [↑](#footnote-ref-19)