3. 具体内容

3.3 决策树部分

3.3.1模型结构

决策树是一种基于树结构的机器学习模型，其基本原理是通过对特征空间进行划分，构建一棵树状结构来进行决策和预测。决策树会根据训练数据学习特征之间的关系和规律，将这些规律表示为一系列的决策路径，以实现对未知样本的分类或回归预测。

决策树是一种直观且易于解释的模型，通过对特征空间进行划分和规则的构建来进行分类。对于Mnist数据集中的图像分类，决策树能够自动选择重要的特征，并捕捉特征之间的非线性关系。此外，决策树具有良好的可扩展性，能够处理Mnist数据集的规模。然而，决策树也存在过拟合和对噪声敏感等问题，可以通过后剪枝等技术进行优化。

下面我们介绍一些决策树模型中的设计要点。

* **节点表示**：决策树的结构由节点和边组成。节点可以分为内部节点和叶子节点。内部节点表示对特征空间的划分，每个内部节点包含一个特征和一个分裂阈值，用于将样本划分到不同的子节点。叶子节点表示最终的预测结果或决策类别。我们实现了下列数据结构来构建决策树。

class Node:

    def \_\_init\_\_(self, feature\_idx=None, threshold=None, left=None, right=None, value=None, is\_leaf=False):

       self.feature\_idx = feature\_idx

       self.threshold = threshold

       self.left = left

       self.right = right

       self.value = value

       self.is\_leaf = is\_leaf

class Node:

    def \_\_init\_\_(self, feature\_idx=None, threshold=None, left=None, right=None, value=None, is\_leaf=False):

        self.feature\_idx = feature\_idx

        self.threshold = threshold

        self.left = left

        self.right = right

        self.value = value

        self.is\_leaf = is\_leaf

* **特征选择**：决策树需要选择最佳的划分特征和阈值来构建树。在我们的实现中，使用了随机选择特征的方法，从特征集合中随机选择一部分特征作为候选特征。这样可以增加模型的多样性，降低过拟合的风险。
* **基尼系数和信息增益**：决策树算法通常使用基尼系数或信息增益作为划分标准，用于选择最佳的划分特征和阈值。基尼系数衡量了样本集合的纯度，信息增益衡量了划分后的纯度提升。在我们的实现中，使用了基尼系数作为划分标准，这样降低了计算对数的复杂度。
* **决策树参数配置**：模型中的决策树具有最大深度、最小划分样本数和划分标准等参数。经过不同参数的选择，最后选择了, 。通过将最大深度设置为20，决策树能够学习更复杂的特征和规则，从而具备更强的表达能力。较大的最大深度可以让模型更深入地划分特征空间，更准确地捕捉训练数据中的特征关系，因此能够更好地适应复杂的数据集。最小划分样本数设置为2意味着在决策树构建过程中，只要有足够的样本数就可以继续划分节点。这有助于决策树捕捉到更细致和复杂的特征模式。
* **后剪枝**：模型实现了后剪枝的方法，通过验证集来剪枝决策树，以减小模型的复杂度和提高泛化能力。后剪枝的过程中，通过比较剪枝前后的分类误差来决定是否剪枝。
* **叶子节点的生成和类别预测**：当决策树递归地划分样本到达一定条件时，如达到分类系数的阈值或达到最好划分样本数，将生成叶子节点并给出最终的预测结果。通常采用多数表决的方式确定叶子节点的类别，即选择该节点中样本数最多的类别作为预测结果。

3.3.2模型实现

下面给出模型实现的代码，并进行解释。

* \_\_init\_\_(self, max\_depth=5, min\_samples\_split=5,criterion='gini') : 构造函数，初始化决策树的最大深度、最小划分样本数和划分标准（基尼系数或信息增益）。

class DecisionTree:

    def \_\_init\_\_(self, max\_depth=5, min\_samples\_split=5,criterion='gini'):

        self.max\_depth = max\_depth #决策树的最大深度

        self.min\_samples\_split = min\_samples\_split #

        #表示在对决策树节点进行分裂时，至少要有多少个样本才能进行分裂，如果小于该值，则不再进行分裂,防止过拟合

        self.criterion = criterion

        self.tree = None

* fit(self, X, y): 训练决策树，根据给定的训练数据和标签构建决策树。

def fit(self, X, y):

        self.n\_classes = len(np.unique(y)) #标签中不重复的元素，即分类的个数

        self.n\_features = X.reshape((X.shape[0], -1)).shape[1] #将X训练集中每一张图片28\*28展开成一维的向量

        self.tree = self.\_grow\_tree(X, y) #构建决策树

        print("训练完成")

* predict(self, X): 对输入的数据集进行预测，返回预测结果。

def predict(self, X):

        return [self.\_predict(inputs) for inputs in X]

def \_predict(self, inputs):

        node = self.tree

        while node.left:

            if inputs[node.feature\_idx] <= node.threshold:

                node = node.left

            else:

                node = node.right

        return node.value

* \_grow\_tree(self, X, y, depth=0):递归构建决策树，根据当前节点的样本和标签，选择最佳的划分特征和阈值来构建左右子树。

def \_grow\_tree(self, X, y, depth=0):

        '''

            递归构建决策树

        '''

        n\_samples, n\_features = X.shape # X.shape = (60000,784)

        n\_labels = len(np.unique(y)) # n\_labels = 10

        # 如果满足以下任一条件，则停止生长树

        # 1. 当前深度大于等于最大深度

        # 2. 样本中只有一种类别

        # 3. 样本数量小于最小划分样本数

        if depth >= self.max\_depth or n\_labels == 1 or n\_samples < self.min\_samples\_split:

            leaf\_value = self.\_leaf\_value(y) # 返回一个叶子节点

            return Node(value=leaf\_value, is\_leaf=True)

        best\_threshold = None

        while best\_threshold is None:

            # 随机选择 sqrt(n\_features) 个特征,即从784列中随机选择28个不重复的列作为特征

            feature\_idxs = np.random.choice(n\_features, int(np.sqrt(n\_features)), replace=False)

            # 根据最佳划分特征和最佳划分阈值划分样本

            best\_feature\_idx, best\_threshold = self.\_best\_criteria(X, y, feature\_idxs)

        left\_idxs, right\_idxs = self.\_split(X[:, best\_feature\_idx], best\_threshold)

        left = self.\_grow\_tree(X[left\_idxs, :], y[left\_idxs], depth+1)

        right = self.\_grow\_tree(X[right\_idxs, :], y[right\_idxs], depth+1)

        return Node(best\_feature\_idx, best\_threshold, left, right)

* \_best\_criteria(self, X, y, feature\_idxs, gain=0): 根据给定的特征集合，找到最佳的分裂特征和分裂阈值。

def \_best\_criteria(self, X, y, feature\_idxs, gain=0):

        '''

        给定特征集合(feature\_idxs)中，找到最佳的分裂特征和分裂阈值

        '''

        best\_gain = -np.inf                                #初始化信息增益为负无穷

        split\_idx, split\_threshold = None, None            #初始化分裂的特征索引和分裂的阈值为None

        for feature\_idx in feature\_idxs:                   #在特征集合中遍历每个特征

            thresholds = np.unique(X[:, feature\_idx])      #获取当前特征列上的所有唯一值，作为可能的分裂阈值

            thresholds = thresholds[np.array([e is not None for e in thresholds])]

            for threshold in thresholds:                   #在当前特征的所有可能分裂阈值中遍历

                if threshold is not None:

                    left\_idxs, right\_idxs = self.\_split(X[:, feature\_idx], threshold)

                    #通过阈值对数据进行分裂，获取左子树和右子树的样本索引

                    if len(left\_idxs) == 0 or len(right\_idxs) == 0:

                    #如果左子树或右子树为空，说明当前阈值无法实现分裂，直接跳过当前阈值

                        continue

                    gain = self.\_gini\_gain(y, left\_idxs, right\_idxs) #计算当前分割的基尼增益

                    if gain > best\_gain:                       #如果当前增益大于最佳增益，则将当前增益和分割特征、阈值更新为最佳值。

                        best\_gain = gain

                        split\_idx = feature\_idx

                        split\_threshold = threshold

        return split\_idx, split\_threshold

* \_split(self, X\_column, threshold=np.inf): 根据阈值将特征列X\_column分割成左子集和右子集，并返回这些子集的索引。

def \_split(self, X\_column, threshold=np.inf):

        left\_idxs = np.argwhere(X\_column <= threshold).flatten()

        right\_idxs = np.argwhere(X\_column > threshold).flatten()

        return left\_idxs, right\_idxs

* \_gini\_gain(self, y, left\_idxs, right\_idxs): 计算基尼系数增益，用于选择最佳的分裂特征和阈值。

def \_gini\_gain(self, y, left\_idxs, right\_idxs):

        p = len(left\_idxs) / (len(left\_idxs) + len(right\_idxs))  # 计算左子节点在样本中的占比

        left\_gini = self.gini(y[left\_idxs])

# 计算左子节点的基尼系数

        right\_gini = self.gini(y[right\_idxs])

# 计算右子节点的基尼系数

        gain = self.gini(y) - p \* left\_gini - (1 - p) \* right\_gini  # 计算基尼系数增益值

        return gain

* \_leaf\_value(self, y): 计算叶子节点的值，即出现频率最高的标签。

def \_leaf\_value(self, y):

      counts = np.bincount(y)

      return np.argmax(counts)

* post\_prune(self, X\_val, y\_val)：后剪枝方法，通过验证集来剪枝决策树，递归地对决策树进行后剪枝处理。

def post\_prune(self, X\_val, y\_val):

        """Post-prune the decision tree."""

        self.num\_pruned = 0

        self.\_post\_prune(X\_val, y\_val, self.tree)

        print(f"Pruned {self.num\_pruned} nodes.")

def \_post\_prune(self, X\_val, y\_val, node):

        if node.is\_leaf:

            return

        # Recursively prune children

        if not node.left.is\_leaf:

            self.\_post\_prune(X\_val, y\_val, node.left)

        if not node.right.is\_leaf:

            self.\_post\_prune(X\_val, y\_val, node.right)

        # Calculate the error with and without the node's children

        y\_val\_pred = self.predict(X\_val)

        error\_without\_prune = np.sum(y\_val\_pred != y\_val) / len(y\_val)

        node\_left = node.left

        node\_right = node.right

        node.left = None

        node.right = None

        y\_val\_pred = self.predict(X\_val)

        error\_with\_prune = np.sum(y\_val\_pred != y\_val) / len(y\_val)

        # Decide whether to prune the node or not

        if error\_with\_prune <= error\_without\_prune:

            node.is\_leaf = True

            node.left = None

            node.right = None

        else:

            node.left = node\_left

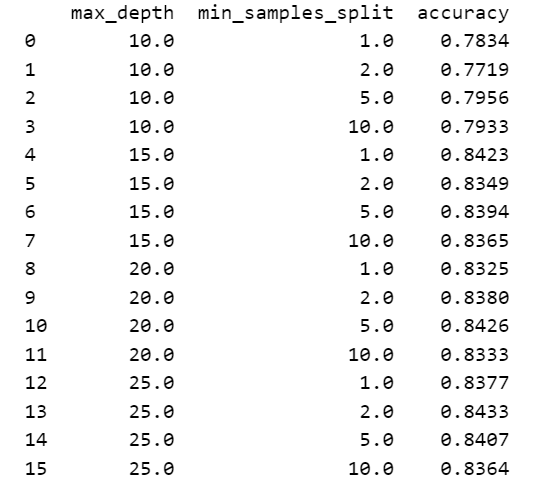
            node.right = node\_right

3.3.3结果分析

3.3.3 参数分析

在**train&test.ipynb**文件中，我们按前文所述的方法进行了数据的导入和解析、数据集的制作、模型创建和训练、测试准确率。

我们尝试了不同参数配置下的模型，结果如下图：



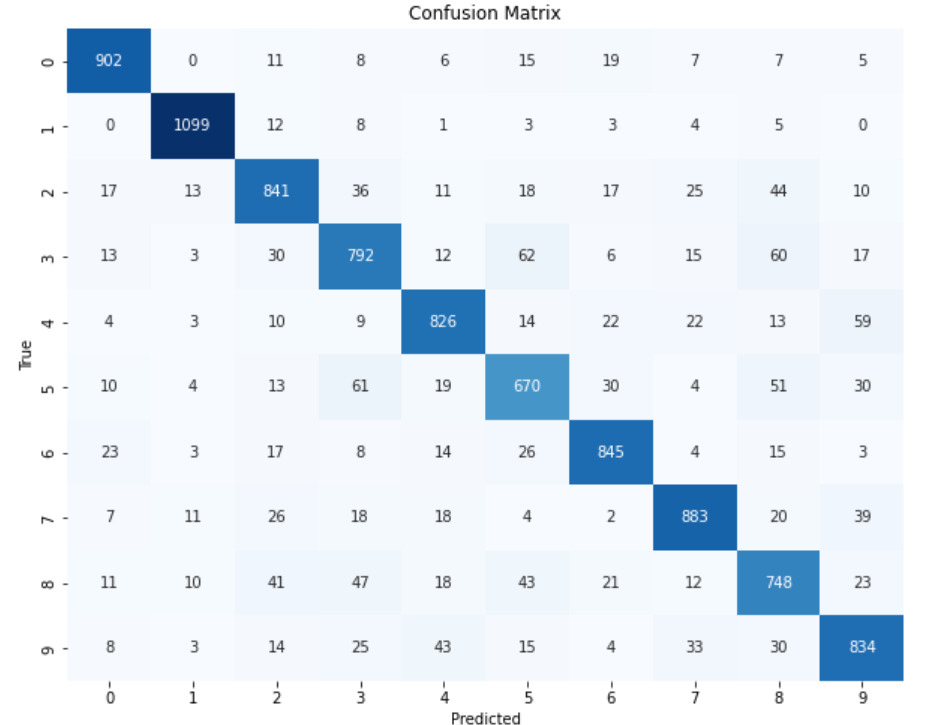
根据给定的结果，可以进行以下分析：

* max\_depth和min\_samples\_split对准确率的影响：从结果可以看出，随着max\_depth的增加，准确率有所提高。这是因为增加最大深度可以增加决策树的复杂度和表达能力，使其能够更好地拟合训练数据，但过大的max\_depth可能导致过拟合。对于min\_samples\_split，较小的值会导致决策树过度拟合训练数据，而较大的值可以减少过拟合风险。从结果来看，min\_samples\_split为5时获得了相对较高的准确率。
* 最佳参数组合：根据本次实验结果，可以看出max\_depth为25，min\_samples\_split为2时，获得了最高的准确率，为0.8433。这个组合在给定的参数范围内表现最好。
* 结果的波动性：在不同参数组合下，准确率有所波动。这是因为不同的参数组合会影响决策树的复杂度、拟合能力和泛化能力。因此，结果的波动性表明模型对参数的敏感性，选择合适的参数可以对准确率产生显著影响。

经过我们对参数的调整测试以及不断的尝试，最后，在某次参数为max\_depth=20,min\_samples\_split=2, criterion='gini'。

时，模型的准确率最高，在测试集上达到了84.8%。

3.3.3.2 模型分析



根据混淆矩阵图的观察结果，分析如下：

* 错误分类较多的类别：从混淆矩阵图中可以看出，模型在3和8的分类、4和9的分类以及3和5的分类上存在较多的错误。这意味着模型在识别这些数字时相对较为困难，可能存在一定的混淆和误判。
* 1和0的分类准确性较高：根据混淆矩阵图，可以观察到模型在1和0的分类上几乎没有出现错误。这说明模型在识别这两个数字时表现出较高的准确性和可靠性，能够很好地将它们与其他数字区分开来。
* 错误分类的原因：错误分类可能是由于数字之间的相似性导致的。例如，3和8在形状上有一定的相似性，容易造成模型的混淆。同样，4和9以及3和5在一些书写方式上也存在一定的相似之处，这可能导致模型在这些类别上出现错误。

综合来看，模型在特定数字之间存在一定的混淆和误判，可能是由于它们在形状和书写方式上的相似性所导致。进一步改进模型的性能可以考虑增加更多的训练数据、调整模型参数或尝试其他分类算法来提高对这些数字的识别准确性。

3.3.3投毒攻击

3.3.3.1 决策树的投毒攻击

决策树模型面临的投毒攻击问题是指恶意攻击者试图通过修改训练数据中的样本标签或特征，来干扰决策树模型的预测结果。这种攻击可以导致模型产生错误的决策或分类结果，从而对系统的安全性和可靠性造成威胁。

投毒攻击的具体方式包括：

* 修改样本标签：攻击者可能会修改训练数据中的样本标签，将正确的标签改为错误的标签。这样一来，当模型进行预测时，它可能会将本应属于某个类别的样本错误地分类到其他类别，导致错误的决策结果。
* 修改样本特征：攻击者还可以修改训练数据中的样本特征，通过添加噪音、扰动或操纵特征值，以改变模型对样本的判断。这样做可能会导致模型对特征的解释和判断出现偏差，从而影响模型的预测准确性。

投毒攻击对决策树模型的影响包括：

* 错误分类：攻击者修改样本标签后，模型可能将本应属于某个类别的样本错误地分类到其他类别。这可能导致对系统进行错误的决策或产生误导性的结果。
* 降低模型性能：投毒攻击可以破坏训练数据的准确性和代表性，从而降低模型的性能。模型在受到攻击的数据上的预测准确率和泛化能力都会受到影响。
* 安全隐患：在一些敏感领域，如金融、医疗等，投毒攻击可能会对系统的安全性和隐私产生严重威胁。攻击者可能通过有目的地修改数据来欺骗模型，获得不当的利益或侵犯用户的隐私。

3.3.3.2投毒攻击实现

投毒攻击的实现分为两个部分：

* **修改样本标签：**

修改样本标签的思想是在60000个训练集标签中随机挑选一定比例的标签进行随机修改。在决策树的修改样本标签攻击中，投毒数量从5000到50000，增量为5000。

修改样本标签的代码如下所示：

def poison(y\_train,poison\_num):

    feature\_idxs = np.random.choice(60000, poison\_num, replace=False)

    for idx in feature\_idxs:

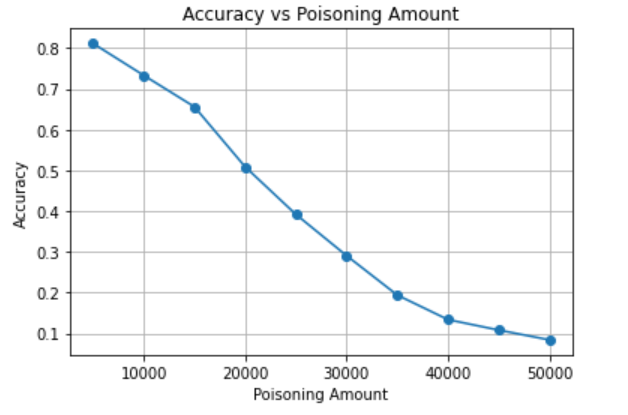
        random\_number = random.randint(0, 9)

        y\_train[idx] = random\_number

    return y\_train

我们实现了poison 函数，该函数接收两个参数y\_train, poison\_num，分别代表数据处理后得到的训练集标签和投毒数量。函数首先使用np.random.choice函数从训练数据的标签集合中选择poison\_num个样本的索引，这些索引将被用于修改对应样本的标签。replace=False参数确保选择的索引不会重复。然后，使用循环遍历选定的样本索引，并在每个索引位置上生成一个随机数random\_number，该随机数范围在0到9之间。这个随机数将被作为新的错误标签来替换原始样本的标签。最后，修改后的标签集合y\_train被返回。

修改样本标签对准确率的影响：



针对修改标签比例对准确率的影响进行分析，我们可以观察到以下情况：

* 随着修改标签数量的增加，准确率呈现下降的趋势。这是因为修改了一定比例的样本标签后，训练数据中存在了错误的标签信息，导致模型的学习过程受到干扰，从而降低了模型的准确率。
* 在修改标签数量较少的情况下，准确率的下降较为缓慢。例如，当修改标签数量为5000时，准确率为81.16%；当修改标签数量为15000时，准确率为65.60%。这是因为修改标签的数量相对较少，训练数据中仍然存在大量正确标签的样本，使得模型仍能够学习到一定程度的正确规律。
* 随着修改标签数量的增加，准确率下降的速率加快。当修改标签数量较大时，例如修改标签数量为20000时，准确率降至50.87%；修改标签数量为40000时，准确率降至13.36%。这是因为修改标签的数量逐渐占据了训练数据的主导，使得模型更难以学习到正确的决策规则，导致准确率急剧下降。

通过这些结果，我们可以得出结论：修改样本标签会对决策树模型的准确率产生负面影响。随着修改标签数量的增加，模型的准确率会逐渐降低。这进一步证明了决策树模型对于训练数据的标签信息敏感，并且容易受到投毒攻击的影响。

* **修改样本特征：**

修改样本特征的思想是在60000个训练集图片中随机挑选一定比例的像素值进行随机修改。在决策树的修改样本标签攻击中，投毒比例从5%到75%，增量为5%。

修改样本特征的代码如下所示：

def poison\_images(images, poison\_percentage):

    num\_images, num\_pixels = images.shape

    num\_pixels\_to\_poison = int(num\_pixels \* poison\_percentage)

    poisoned\_images = np.copy(images)

    for i in range(num\_images):

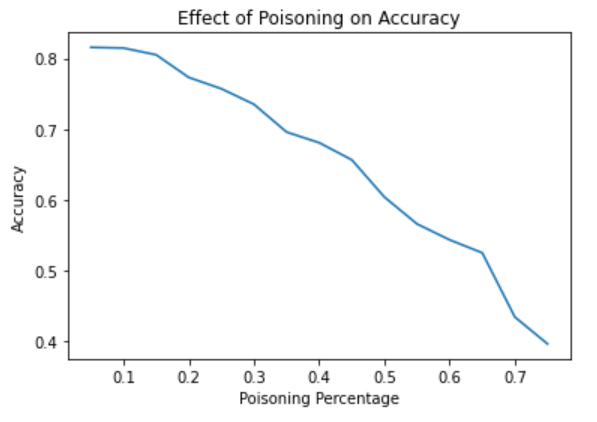
        pixels\_to\_poison = np.random.choice(num\_pixels, num\_pixels\_to\_poison, replace=False)

        poisoned\_images[i, pixels\_to\_poison] = np.random.randint(0, 256, num\_pixels\_to\_poison)

return poisoned\_images

我们实现了poison\_images函数，接受两个参数：images是原始的图像数据集，poison\_percentage表示要修改的像素比例。函数首先获取原始图像数据集的维度信息，其中num\_images表示图像数量，num\_pixels表示每个图像的像素数。然后，根据指定的poison\_percentage计算要修改的像素数量num\_pixels\_to\_poison，这是将原始图像中一定比例的像素进行修改的数量。接下来，创建一个副本poisoned\_images，用于存储修改后的图像数据。使用np.copy函数对原始图像数据进行复制，以免影响原始数据。然后，通过循环遍历每个图像，对每个图像中的像素进行修改。使用np.random.choice函数从0到num\_pixels范围内选择num\_pixels\_to\_poison个要修改的像素的索引，确保选择的索引不重复。随后，使用np.random.randint函数生成num\_pixels\_to\_poison个随机整数，范围在0到255之间，作为新的像素值。将这些新的像素值赋值给对应的像素位置，从而实现对图像特征的修改。最后，返回修改后的图像数据集poisoned\_images。

修改样本特征对准确率的影响：



针对修改特征比例对准确率的影响进行分析，我们可以观察到以下情况：

* 随着修改特征比例的增加，准确率呈现下降的趋势。这是因为修改了一定比例的样本特征后，训练数据中的特征信息不再准确，导致模型在学习过程中受到了干扰，从而降低了准确率。
* 在修改特征比例较低的情况下，准确率的下降较为缓慢。例如，当修改特征比例为5%时，准确率为81.55%；当修改特征比例为15%时，准确率为80.50%。这是因为修改特征的比例相对较低，训练数据中仍保留了大部分正确的特征信息，使得模型仍能够学习到一定程度的正确规律。
* 随着修改特征比例的增加，准确率下降的速率加快。当修改特征比例较高时，例如修改特征比例为55%时，准确率降至56.59 %；修改特征比例为75%时，准确率降至39.64%。这是因为修改特征的比例逐渐占据了训练数据的主导，使得模型更难以学习到正确的特征关系，导致准确率急剧下降。
* 修改特征对准确率的影响相对较小，下降速度不如修改标签明显。对于图像数据而言，像素值的微小变化可能对模型的判断产生较小的影响,而修改标签直接改变了样本的真实标签，对模型的判断产生了更大的影响，因此准确率的下降更为明显。另外，决策树模型在处理高维特征时，可能会更加注重特征之间的相对关系，而不是单个特征的具体取值。这导致即使部分特征被修改，模型仍然能够从其他特征中学习到某种程度的分类规律，从而准确率下降相对较小。

通过这些结果，我们可以得出结论：修改样本特征会对决策树模型的准确率产生负面影响。随着修改特征比例的增加，模型的准确率会逐渐降低。这进一步证明了决策树模型对于训练数据的特征信息敏感，并且容易受到投毒攻击的影响。

3.3.3.3投毒攻击防范

为应对投毒攻击，可以采取以下防御措施：

* 数据过滤和清洗：在使用训练数据之前，对数据进行严格的过滤和清洗，排除异常样本和错误标签，以保证训练数据的质量和准确性。
* 多模型集成：使用多个独立的决策树模型进行集成学习，通过多数投票或加权投票的方式来获得最终的预测结果。这样可以降低单个模型受到投毒攻击的影响。
* 检测异常样本：监控模型对输入数据的预测结果，检测是否存在异常样本。通过设定阈值或采用异常检测算法，可以及时发现并排除投毒攻击引入的异常样本。
* 数据加密和隐私保护：对敏感数据进行加密和隐私保护，限制非授权访问，以防止攻击者获取数据并进行投毒攻击。

综上所述，投毒攻击是决策树模型面临的一种重要威胁。为了提高模型的鲁棒性和安全性，我们需要采取有效的防御措施来识别和抵御投毒攻击。