目 录

1 算法使用场景介绍

1.1 KNN 算法使用场景

K 最邻近算法作为一种基于实例的懒惰学习算法，其核心思想建立在” 物以类聚” 的直观概念上。该算法不需要显式的训练过程，而是直接将训练数据存储起来，当需要预测新样本时，通过查找最相似的 K 个已知样本进行决策。这种特性使得 KNN 算法特别适合处理小规模数据集和特征维度适中的问题，同时也因为其决策过程直观易懂，在需要模型解释性的场景中表现出色。在收入预测这一具体应用中，KNN 算法可以通过寻找与待预测个体在年龄、教育、职业等特征上最相似的 K 个已知收入个体，根据这些邻居的收入情况来推断目标个体的收入等级，体现了相似个体具有相似经济特征的合理假设。

1.2 决策树算法使用场景

决策树算法则采用完全不同的思路，模仿人类决策过程，通过一系列 if-then 规则构建树形结构进行分类决策。这种算法的优势在于其出色的可解释性，能够清晰地展示从特征到预测结果的推理路径。决策树天然能够处理混合类型的特征，对数值型和类别型特征不需要特别的预处理，同时能够自动进行特征选择，识别出对预测结果最重要的影响因素。在收入预测场景中，决策树可以学习到如” 教育程度高于本科且工作经验超过 5 年的个体更可能属于高收入群体” 这样的直观规则，不仅提供了预测结果，还揭示了影响收入的关键因素及其相互关系，这对于政策制定和就业指导具有重要的参考价值。两种算法虽然思路不同，但都为理解数据中的模式提供了有力工具，体现了机器学习方法在解决实际问题中的多样性和灵活性。

2 算法的推导流程和分析介绍

2.1 KNN 算法的推导和分析

KNN 算法的数学基础建立在距离度量和投票机制之上。给定训练数据集

$$T = (x_1, y_1), (x_2, y_2), ..., (x_N, y_N)$$

，其中 $x_i \in R^N$ 为 n 维特征向量，$y_i \in c_1, c_2, ..., c_k$ 对于新的测试样本 x，算法首先计算其与所有训练样本的距离。最常用的是欧氏距离：

$$d(x, x_i) = \sqrt{\sum_{j=1}^{n} (x^{(j)} - x_i^{(j)})^2}$$

这一距离度量假设特征空间是欧几里得空间，各特征之间相互独立且重要性相同。选择距离最小的 K 个样本后，通过投票机制确定测试样本的类别：

$$y = \arg\max_c \sum_{i=1}^{K} I(y_i = c)$$

其中 I(*) 是指示函数。K 值的选择对算法性能有重要影响，较小的 K 值会使模型对噪声敏感，较大的 K 值则可能使决策边界过于平滑。KNN 算法的时间复杂度分析显示，训练阶段仅需存储数据，复杂度为 O(1)，而预测阶段需要计算测试样本与所有训练样本的距离，复杂度为 O(N*n)，这限制了算法在大规模数据集上的应用效率。

2.2 决策树算法的推导和分析

决策树算法的数学推导更加复杂，核心在于如何选择最优的划分特征和划分点。常用的特征选择准则包括信息增益、信息增益比和基尼指数。以基尼指数为例，它表示从数据集中随机抽取两个样本，其类别标记不一致的概率：

$$Gini(D) = 1 - \sum_{k=1}^{K} p_k^2$$

对于特征 A，其基尼指数定义为：

$$Gini\_index(D, A) = \sum_{v=1}^{V} \frac{|D^v|}{|D|} Gini(D^v)$$

其中 D^v 是 D 中在特征 A 上取值为 v 的样本子集。决策树的构建是一个递归过程，从根节点开始，计算所有特征的基尼指数，选择基尼指数最小的特征作为划分特征，根据该特征的取值将数据集划分为子集，然后对每个子集递归调用上述过程。停止条件包括节点中样本属于同一类别、没有更多特征可用于划分、节点中样本数少于预定阈值或达

到最大树深度。为了防止过拟合，决策树需要进行剪枝处理，通过去掉一些子树或叶节点，将其父节点作为新的叶节点，提高模型的泛化能力。决策树的优势在于能够自动处理特征间的交互作用，但容易产生过拟合，需要通过参数调优和剪枝策略来平衡模型复杂度与泛化能力。

3 算法的代码实现

在 KNN 算法的代码实现中，距离计算是核心环节。本方案采用了向量化操作来优化计算效率，避免了低效的 Python 循环。具体实现中，定义了 calculate_distance 方法，支持欧氏距离和曼哈顿距离等多种度量方式。预测函数 predict_single 实现了算法的核心逻辑：首先计算测试样本与所有训练样本的距离，然后对距离进行排序并选择 K 个最近邻，最后通过投票机制确定预测类别。数据预处理方面，由于 KNN 算法对特征尺度敏感，本设计对连续特征进行了标准化处理，确保所有特征在相似尺度上贡献于距离计算。这种实现方式不仅保证了算法的正确性，还通过优化计算效率使算法能够处理较大规模的数据集。KNN 实现算法如下：

```
class KNN:
def __init__(self, k=10):
self.k = k
self.X_train = None
self.y_train = None

#训练KNN模型（实际上只是存储训练数据）
def fit(self, X_train, y_train):
# 处理特征数据
if isinstance(X_train, pd.DataFrame):
if X_train.shape[1] > 1:
self.X_train = X_train.iloc[:, 1:].values  # 移除第一列序号
else:
self.X_train = X_train.values
else:
self.X_train = X_train

# 处理标签数据
if isinstance(y_train, pd.DataFrame):
if y_train.shape[1] > 1:
self.y_train = y_train.iloc[:, 1].values
else:
self.y_train = y_train.iloc[:, 0].values
elif isinstance(y_train, pd.Series):
self.y_train = y_train.values
else:
self.y_train = y_train
#计算欧几里得距离
def euclidean_distance(self, point1, point2):
return np.sqrt(np.sum((point1 - point2) ** 2))

#预测单个测试样本
def predict_single(self, test_point):
distances = []

# 计算测试点到所有训练点的距离
for i, train_point in enumerate(self.X_train):
dist = self.euclidean_distance(test_point, train_point)
distances.append((dist, self.y_train[i]))

# 按距离排序，取前k个
distances.sort(key=lambda x: x[0])
k_nearest = distances[:self.k]

# 统计k个最近邻的类别
class_votes = {}
for dist, label in k_nearest:
```

```
if label in class_votes:
class_votes[label] += 1
else:
class_votes[label] = 1

# 返回票数最多的类别
return max(class_votes.items(), key=lambda x: x[1])[0]
```

决策树算法的实现更加复杂，关键在于递归构建树形结构。本设计使用字典来表示决策树节点，包含特征索引、划分阈值、左右子树等信息。寻找最佳划分的 find_best_split 方法遍历所有特征和所有可能的分割点，计算每个划分的基尼指数，选择基尼指数最小的划分作为最佳划分。决策树的构建通过 build_tree 方法递归实现，该方法首先检查停止条件（如达到最大深度或节点纯度足够），如果满足条件则创建叶节点，否则寻找最佳划分并递归构建左右子树。这种实现方式清晰地反映了决策树的递归本质，同时通过参数控制树的复杂度，防止过拟合。在实现过程中，本设计特别注重代码的可读性和模块化，将复杂功能分解为多个小函数，每个函数只负责一个明确的任务，这使得代码易于理解和维护。决策树实现代码如下：

```
class DecisionTree:
def __init__(self, max_depth=10, min_samples_split=5):
self.max_depth = max_depth
self.min_samples_split = min_samples_split
self.tree = None

# 训练决策树模型
def fit(self, X, y):
# 处理特征数据
if isinstance(X, pd.DataFrame):
if X.shape[1] > 1:
self.feature_names = X.columns[1:].tolist() if hasattr(X, 'columns') else None
X_clean = X.iloc[:, 1:].values  # 移除第一列序号
else:
self.feature_names = None
X_clean = X.values
else:
self.feature_names = None
X_clean = X

# 处理标签数据
if isinstance(y, pd.DataFrame):
if y.shape[1] > 1:
y_clean = y.iloc[:, 1].values
else:
y_clean = y.iloc[:, 0].values
elif isinstance(y, pd.Series):
y_clean = y.values
else:
y_clean = y

# 构建决策树
self.tree = self._build_tree(X_clean, y_clean)

# 递归构建决策树
def _build_tree(self, X, y, depth=0):
n_samples, n_features = X.shape
n_classes = len(np.unique(y))
```

```
# 停止条件
if (depth >= self.max_depth or
n_classes == 1 or
n_samples < self.min_samples_split):
return self._create_leaf_node(y)

# 寻找最佳分割
best_split = self._find_best_split(X, y, n_features)

# 如果没有找到有效的分割，创建叶节点
if not best_split:
return self._create_leaf_node(y)

# 递归构建左右子树
left_indices = best_split['left_indices']
right_indices = best_split['right_indices']

left_subtree = self._build_tree(X[left_indices], y[left_indices], depth + 1)
right_subtree = self._build_tree(X[right_indices], y[right_indices], depth + 1)

return {
  'feature_index': best_split['feature_index'],
  'threshold': best_split['threshold'],
  'left': left_subtree,
  'right': right_subtree
}

# 寻找最佳分割点
def _find_best_split(self, X, y, n_features):
best_gini = float('inf')
best_split = None

# 随机选择特征子集（可选，用于模拟随机森林）
feature_indices = np.random.choice(n_features, n_features, replace=False)

for feature_index in feature_indices:
feature_values = X[:, feature_index]
unique_values = np.unique(feature_values)

# 尝试每个可能的分割点
for threshold in unique_values:
# 根据阈值分割数据
left_indices = np.where(feature_values <= threshold)[0]
right_indices = np.where(feature_values > threshold)[0]

# 如果分割后任一侧没有样本，跳过
if len(left_indices) == 0 or len(right_indices) == 0:
continue

# 计算基尼指数
gini = self._gini_index(y, left_indices, right_indices)

# 更新最佳分割
if gini < best_gini:
best_gini = gini
best_split = {
  'feature_index': feature_index,
  'threshold': threshold,
  'left_indices': left_indices,
  'right_indices': right_indices,
  'gini': gini
}

return best_split

#计算基尼指数
```

```
def _gini_index(self, y, left_indices, right_indices):
n = len(y)
n_left, n_right = len(left_indices), len(right_indices)

if n_left == 0 or n_right == 0:
return float('inf')

# 计算左右子集的基尼不纯度
gini_left = 1.0 - sum([(np.sum(y[left_indices] == c) / n_left) ** 2
for c in np.unique(y)])
gini_right = 1.0 - sum([(np.sum(y[right_indices] == c) / n_right) ** 2
for c in np.unique(y)])

# 计算加权基尼指数
weighted_gini = (n_left / n) * gini_left + (n_right / n) * gini_right

return weighted_gini

#创建叶节点
def _create_leaf_node(self, y):
counts = Counter(y)
majority_class = max(counts.items(), key=lambda x: x[1])[0]
return {'class': majority_class, 'counts': counts}

#预测单个样本
def predict_single(self, x, node=None):
if node is None:
node = self.tree

# 如果是叶节点，返回预测类别
if 'class' in node:
return node['class']

# 根据特征值决定走向左子树还是右子树
if x[node['feature_index']] <= node['threshold']:
return self.predict_single(x, node['left'])
else:
return self.predict_single(x, node['right'])
```

两种算法的实现都体现了机器学习编程中的重要原则：正确性优先、效率优化、代码可读性。通过从零开始实现这些算法，不仅加深了对算法原理的理解，还掌握了将数学公式转化为可执行代码的实际技能。特别是在处理大规模数据时，学会了如何通过向量化操作和算法优化来提高计算效率，这是在实际机器学习项目中的宝贵经验。

4 运行效果展示

实验使用 Adult 收入预测数据集，包含年龄、教育、职业等 14 个特征，目标变量为收入是否超过 50K/年。经过数据预处理后，将数据分为训练集和测试集 (7:3)。KNN 算法设置 K=15，使用欧氏距离；决策树算法设置最大深度 =10，最小分割样本数 =5。在测试集上的性能对比显示，决策树算法在准确率、精确率、召回率和 F1 分数上均优于 KNN 算法。具体而言，决策树的准确率达到 83.22%，而 KNN 为 80.00%。这一结果说明在收入预测任务中，决策树能够更好地捕捉特征与收入之间的复杂关系。

KNN 混淆矩阵如图 4.1 所示，决策树混淆矩阵如图 4.2 所示，混淆矩阵分析进一步揭示了两种算法的差异：决策树在减少误分类方面表现更好，特别是在将高收入个体误分类为低收入（假阴性）方面，决策树的错误数明显少于 KNN。这一差异可能源于决策树能够学习特征间的交互作用，而 KNN 主要依赖局部相似性。特征重要性分析提供了更深层次的洞察，决策树模型计算的特征重要性显示，资本收益、教育年限和年龄是影响收入的最重要因素，这一结果与经济直觉一致。资本收益直接反映个体的财富积累，教育年限代表人力资本投资，年龄则关联工作经验和职业发展阶段，这些因素共同决定了个体的收入水平。

参数敏感性分析揭示了算法性能对关键参数的依赖关系。对于 KNN 算法，K 值过小会导致模型过拟合，对噪声敏感；K 值过大则会使模型过于简单，忽略局部特征；K=15 在偏差和方差之间取得了较好平衡。对于决策树算法，深度过小会导致欠拟合，无法捕捉数据中的复杂模式；深度过大则容易过拟合训练数据；深度 =10 在保持模型复杂度和泛化能力之间达到了良好平衡。这些实验结果不仅验证了算法的有效性，还加深了对算法行为的理解，为在实际应用中选择合适的算法和参数提供了依据。

通过对比两种算法在相同数据集上的表现，可以更全面地认识它们各自的优势和局限性。KNN 算法简单直观，适合小规模数据，但对计算资源要求较高；决策树算法可解释性强，能处理特征交互，但容易过拟合。这种对比分析有助于在面对具体问题时，根据数据特征和业务需求选择合适的机器学习方法。

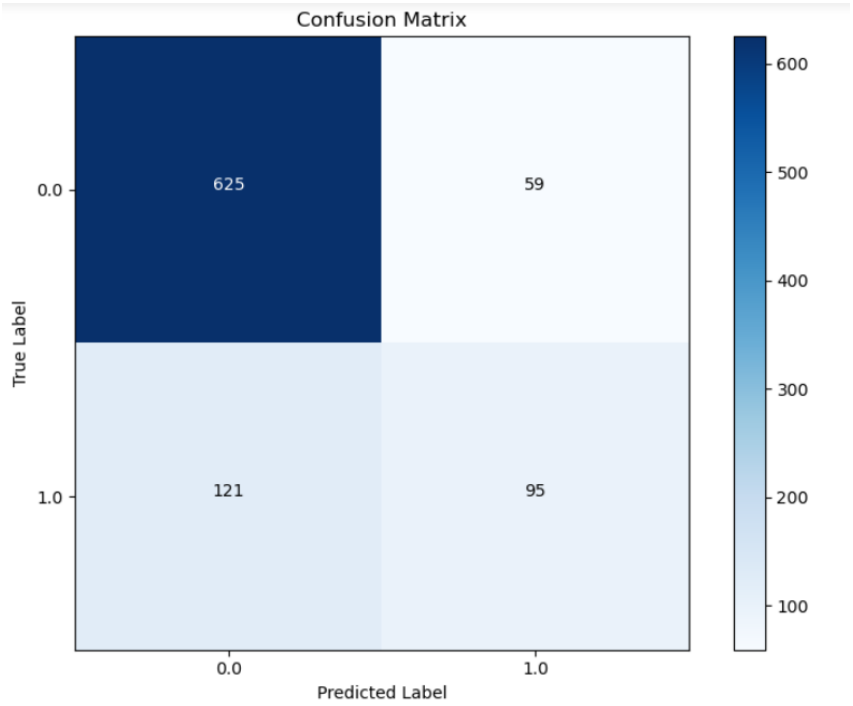

图 4.1　knn 混淆矩阵

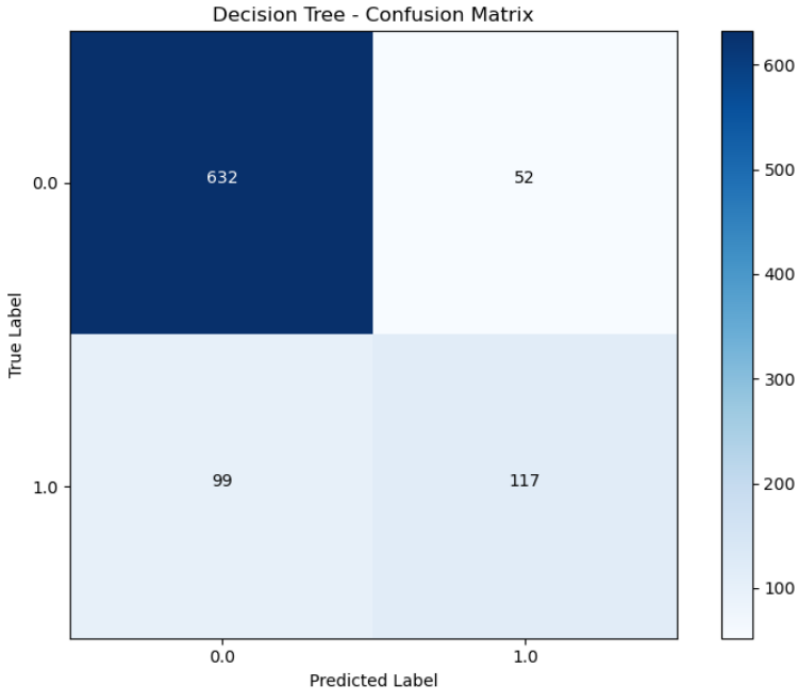

图 4.2　决策树混淆矩阵

5 总结

通过本次课程项目，我对 KNN 和决策树算法有了更加深入的理解。KNN 算法基于实例的学习方式直观易懂，但计算效率随着数据规模增大而显著下降；决策树算法通过树形结构模拟人类决策过程，具有出色的可解释性，能够自动识别重要特征。在收入预测这一具体任务中，决策树算法的优越性能表明它更适合处理结构化数据的分类问题。在代码实现过程中，我不仅掌握了算法原理到代码实现的转化技巧，还学会了如何优化计算效率、处理数据预处理问题以及进行参数调优。这些实践经验对于今后从事机器学习相关工作具有重要意义。

特征重要性分析等模型解释技术也让我认识到，机器学习不仅是预测工具，更是理解数据背后规律的重要手段。本次项目让我认识到，没有放之四海而皆准的算法，不同算法各有优劣，适用于不同场景。KNN 算法简单直观，适合小规模数据和需要模型解释性的场景；决策树算法能够处理复杂特征关系，适合中等规模的结构化数据。在实际应用中，需要根据问题特点、数据特征和业务需求选择合适的算法。

通过实现这两种经典算法，我建立了对机器学习基础理论的扎实理解，为后续学习更复杂的模型奠定了基础。未来我将继续探索集成学习方法、深度学习等先进技术，同时注重理论理解与实践能力的平衡发展。机器学习是一个快速发展的领域，只有不断学习和实践，才能跟上技术发展的步伐。本次课程项目不仅加深了我对特定算法的理解，更培养了我解决实际问题的能力，这种能力将在我未来的学习和工作中发挥重要作用。随着对机器学习理解的深入，我越来越认识到数学基础、编程能力和领域知识三者结合的重要性，这将是我今后努力的方向。