《大数据机器学习》第 2 次实验

姓名：刘培源　　学号：2023214278

题目 1：
基于 KD 树的 KNN 的实现。熟悉 KD 树的原理与构建过程。

答：
KD 树的原理： KD 树（K-dimensional tree）是一种用于组织和搜索 K 维空间中的数据点的二叉树结构。KD 树允许快速检索多维空间数据的方法，特别是对于维数不高的情况。
构建过程： 构建 KD 树的过程可以描述为以下步骤：

1. 选择轴 σ：在深度 d 的节点处，选择 $\sigma = d \mod k$ 作为划分数据集的轴。
2. 划分点 ξ：在轴 σ 上选择中位数 ξ 作为划分点，将数据集分为两个子集：X_{left} 和 X_{right}。
3. 递归构建：对 X_{left} 和 X_{right} 分别递归执行步骤 1 和 2，构建左右子树。

数学公式可以表示为：

$$\sigma = d \mod k$$

$$\xi = \text{median}(X[\sigma])$$

递归过程中构建的节点可以表示为：

$$N(d) = \begin{cases} \text{空}, & \text{如果} X = \text{空集} \\ \{\xi, N_{left}(d+1), N_{right}(d+1)\}, & \text{其他情况} \end{cases}$$

其中，$N_{left}(d+1)$ 和 $N_{right}(d+1)$ 分别是左子树和右子树。
搜索与应用： KD 树搜索时，通过比较目标点和节点数据在划分轴上的值，选择合适的子树进行搜索，同时利用几何属性剪枝，提高搜索效率。KD 树在多维空间的搜索和分类问题中有广泛应用，特别是在 K 最邻近（KNN）算法中，KD 树能够显著降低计算的复杂度。

题目 2：

MNIST 数据集分类。根据提供的代码，理解代码的计算流程，将代码补充完整。

答：

我的核心代码 search_kd_tree 如下（带注释的版本可以在代码文件中查看）：

```python
def search_kd_tree(tree, target, k=3):
    if tree is None:
        return []
    k_nearest, stack = [], [(tree, 0)]
    while stack:
        node, depth = stack.pop()
        if node is None:
            continue
        distance = euclidean_distance(target, node.data)
        if len(k_nearest) < k:
            k_nearest.append((node.data, distance))
        else:
            max_index = max(range(k), key=lambda i: k_nearest[i][1])
            if k_nearest[max_index][1] > distance:
                k_nearest[max_index] = (node.data, distance)
        axis = depth % target.shape[0]
        axis_diff = target[axis] - node.data[axis]
        if axis_diff <= 0:
            stack.append((node.left, depth + 1))
            if len(k_nearest) < k:
                stack.append((node.right, depth + 1))
            else:
                max_index = max(range(k), key=lambda i: k_nearest[i][1])
                if k_nearest[max_index][1] > abs(axis_diff):
                    stack.append((node.right, depth + 1))
        else:
            stack.append((node.right, depth + 1))
            if len(k_nearest) < k:
                stack.append((node.left, depth + 1))
            else:
                max_index = max(range(k), key=lambda i: k_nearest[i][1])
                if k_nearest[max_index][1] > abs(axis_diff):
                    stack.append((node.left, depth + 1))
    return [data for data, _ in k_nearest]
```

算法伪代码如下：

Algorithm 1: 搜索 KD 树以找到 k 个最近邻点的伪代码

Input: KD 树：tree，目标点：target，最近邻个数：k

Output: 目标点的 k 个最近邻点：k_nearest

Function *search_kd_tree(tree, target, k)*:
- k_nearest ← []
- stack ← [(tree, 0)]
- **while** *stack 不为空* **do**
  - node, depth ← stack.pop()
  - **if** *node 不为 None* **then**
    - distance ← `EuclideanDistance`(*target, node.data*)
    - 更新 k_nearest 列表
    - axis ← depth % len(target)
    - axis_diff ← target[axis] - node.data[axis]
    - 根据 axis_diff 和 k_nearest 更新 stack
  - **end if**
- **end while**
- **return** *k_nearest 中的数据点*

最终得到的分类准确率为 92.8%，这与 sklearn.neighbors 中 KNeighborsClassifier 的实现的分类准确率是一摸一样的，这验证了我实现的有效性。

题目 3： 对 KNN 分类器进行超参数的搜索，选取你的最优的超参数下的 KNN 分类器的优化结果 **答：**

KNN 分类器的超参数只有一个 k，因此我探究了 k 从 1 到 10 对于分类准确度的影响，实验代码如下：

```
# 超参数搜索，搜索k从1到10，输出使得模型性能最好的k

best_acc = -np.inf

for k in range(1, 11):
    print(f"Searching for k = {k}")
    y_pred_k = knn_classifier(X_train, y_train, X_test, k)
    accuracy = accuracy_score(y_test, y_pred_k)
    print(f"KNN Accuracy when k = {k}: {accuracy * 100:.2f}%")

    if accuracy > best_acc:
        best_acc = accuracy
        best_k = k

print(f"The best k is {best_k} with accuracy {best_acc * 100:.2f}%")
```

我搜索的结果如表1：

k 值	1	2	3	4	5	6	7	8	9	10
Acc	91.70%	91.30%	92.80%	91.90%	92.60%	92.20%	92.40%	91.70%	91.90%	91.30%

Table 1: 不同 k 值条件下的预测准确率

可以看到最优的预测准确率在 $k = 3$ 时取到。

Bonus1：尝试使用不同的策略来构建 KD 树，使得在分类阶段可以有更快的分类效率。

答：

我进一步优化 search_kd_tree，采用最大堆优化来加速 KD 树的搜索，实现代码如下：

```python
from heapq import heappush, heappop

# 搜索KD的最大堆实现
def search_kd_tree_heap(tree, target, k=3):
    best_nodes = []  # 使用最大堆来存储最近邻
    def visit_node(node, target, depth):
        if node is None:
            return
        node_distance = euclidean_distance(target, node.data)

        # 使用负距离以实现最大堆
        if len(best_nodes) < k:
            heappush(best_nodes, (-node_distance, tuple(node.data)))
        elif -node_distance > best_nodes[0][0]:
            heappop(best_nodes)
            heappush(best_nodes, (-node_distance, tuple(node.data)))

        axis = depth % len(target)
        next_node = node.left if target[axis] < node.data[axis] else node.right
        other_node = node.right if next_node == node.left else node.left

        visit_node(next_node, target, depth + 1)

        # 检查另一侧的子树是否有可能包含更近的点
        if (len(best_nodes) < k or abs(target[axis] - node.data[axis]) < -best_nodes[0][0]):
            visit_node(other_node, target, depth + 1)

    visit_node(tree, target, 0)

    return [data for _, data in best_nodes]
```

可以看到，我用了 recursion 的方法以及最大堆来加提升搜索的速度，在同一实验条件下（数据集，k 值）的条件下所获得的结果与 sklearn 的实现与未优化的实现均一摸一样，同时对于未优化的实现有一定的加速效果。