一. (30 points) **计算学习理论**

本题探讨计算学习理论章节中 VC 维的性质。

1. (10 points) 请在样本空间 $\mathcal{X} = [0, 1]$ 上构造一个有限假设空间 $\mathcal{H}$ 使得 $\mathrm{VC}(\mathcal{H}) = \lfloor \log_2(|\mathcal{H}|) \rfloor$.
2. (10 points) 定义轴平行四边形概念类 $\mathcal{H} = \{h_{(a_1,a_2,b_1,b_2)}(x, y) : a_1 \leq a_2 \wedge b_1 \leq b_2\}$, 其中

$$h_{(a_1,a_2,b_1,b_2)}(x, y) = \begin{cases} 1 & \text{if } a_1 \leq x \leq a_2 \wedge b_1 \leq y \leq b_2 \\ 0 & \text{otherwise} \end{cases}$$

   请证明 $\mathcal{H}$ 的 VC 维为 4.
3. (10 points) 请证明最近邻分类器的假设空间的 VC 维可以为无穷大.

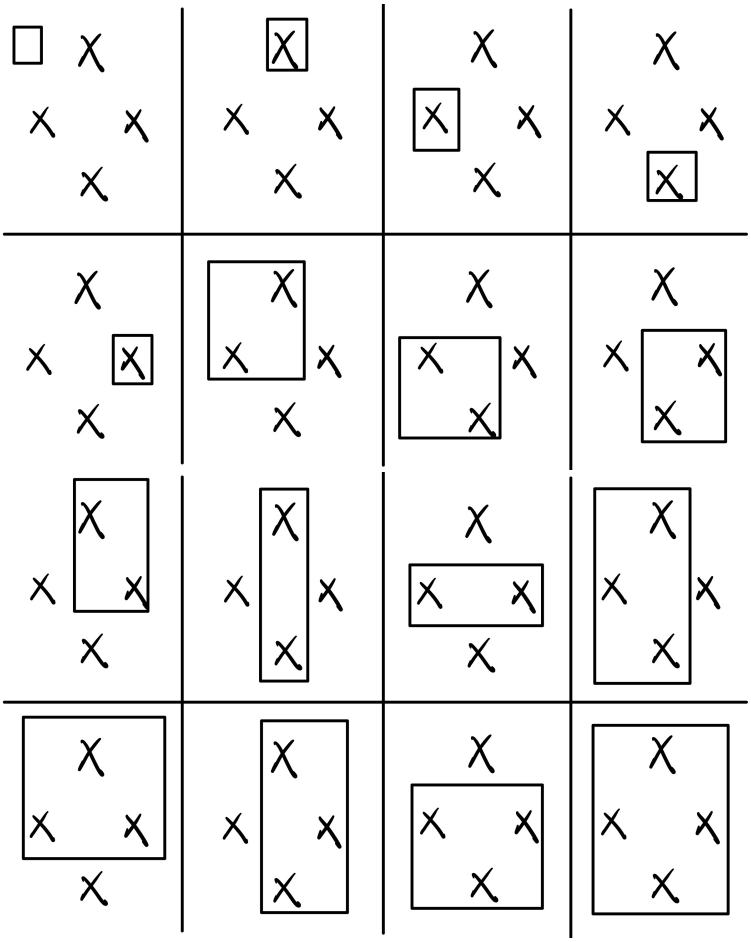
解:

1. 构造有限假设空间 $\mathcal{H} = \{h_1,\ h_2\}$ ，其中

$$h_1(x) = \begin{cases} 1 & \text{if } x < 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$h_2(x) = \begin{cases} 1 & \text{if } x \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

   对于任意一个 $[0, 1]$ 上的样本， h_1、h_2 会分别给出不同的标记，故大小为 1 的示例集可以被该有限假设空间打散。而对于任意两个 $[0, 1]$ 上的样本，该假设空间无法给出四种标记组合，所以 $\mathrm{VC}(\mathcal{H}) = \lfloor \log_2(|\mathcal{H}|) \rfloor = 1$.
2. 按如下 16 种方法划分,由轴平行四边形概念类 $\mathcal{H} = \{h_{(a_1,a_2,b_1,b_2)}(x, y) : a_1 \leq a_2 \wedge b_1 \leq b_2\}$ 的定义, 落在轴平行四边形内样本标记为 1, 外面的样本标记为 0, 可以得到全部 16 种标记组合。故 $\mathcal{H}$ 的 VC 维至少为 4。



而当样本集大小为 5 时，一定会有一个样本落在所有包含其余四个样本的轴平行四边形内部，故该样本的标记无法在其余四个样本的标记均为 1 时为 0，这样也就无法实现所有对分。故 $\mathcal{H}$ 的 VC 维为 4。

3. 对于最近邻分类器，测试集中某个样本的标签仅由训练集中离该测试样本最近的那个样本决定。故可以假设一个特殊的训练集，其每个样本无限接近于测试集中的某个样本，由于采用最近邻，即测试集中每个样本的标记完全由无限接近于它的那个训练集中的样本的标记决定，只要对训练集的样本尝试所有标记组合，即可得到测试集样本的所有标记组合。所以最近邻分类器的假设空间的 VC 维可以为无穷大。

二. (40 points) **编程题：特征工程和规则学习**

本题涉及一个简单的图像识别问题，主要是对示例图 1中的 10 种图案

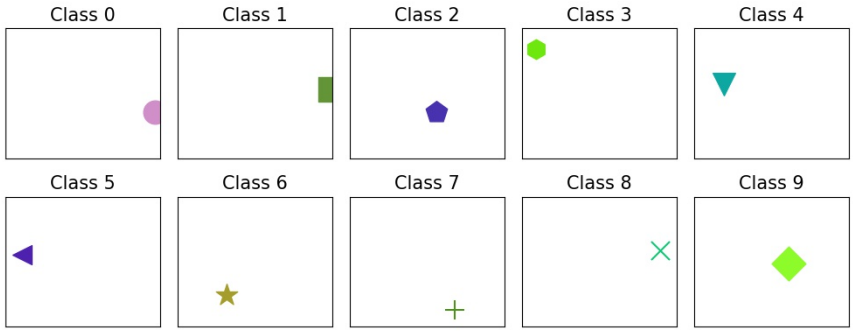


Figure 1: 图案分类问题示例图

进行识别，图案有不同大小、不同颜色，并且处在图像中不同位置，即不一定在中心位置，可能出现部分观测 (某部分可能缺失，例如图中的 Class 0)。分类依据主要是其形状，例如五角星、三角形、加号、正方形等等。本题代码文件夹在 “problem2-code” 目录下。**请在作答中贴出核心代码文件和结果，以方便批改**。

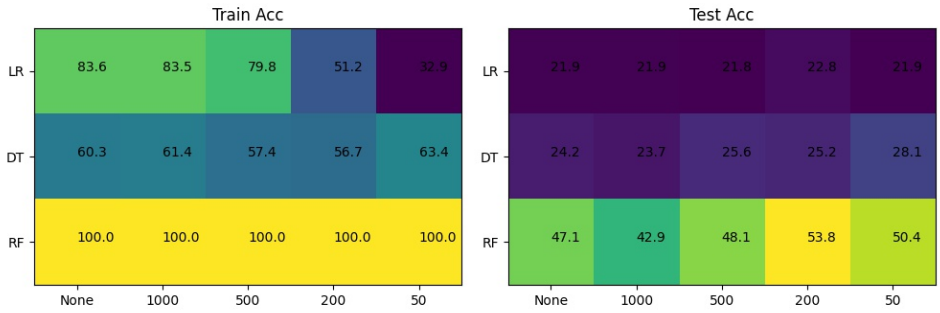
1. (10 points) 根据 “generate_samples.py” 生成训练测试图片，其中 n_samples 控制了训练测试样本中每个类别的数目，示例文件设置为 200，那么训练测试集各有 2000 个样本。生成好图像之后，“load_data.py” 提供了读取图像数据的示例代码，“train_p0.py” 提供了使用对数几率回归 (Logistic Regression)、决策树 (Decision Tree)、随机森林 (Random Forest) 进行分类的示例代码。该代码还提供了可做 PCA 降维的选项。该代码没有提取任何特征，只是对图像缩放为 32 * 32 大小的彩色图像，然后使用所有像素点组成的向量进行分类。尝试运行该代码，将结果记录下来。此外，鼓励使用更多的分类器、更多的超参数设置、更多的数据进行训练，对比一下这些设置是否可以提升性能？例如：增加训练数据数量是否会提升性能?
2. (10 points) 考虑到图像分类只和其形状有关，和其大小、位置、颜色无关，所以尝试将图像变为灰度图、定位目标的中心位置并进行裁剪等操作，然后再进行同样的训练。试着比较这种数据预处理带来的性能提升。提示：高阶的数据预处理还可以将灰度图变为轮廓图或者提取边缘信息等。
3. (10 points) 上述虽然去除了一些和分类任务无关的信息 (例如：颜色)，但是其使用的特征依然是像素点信息。尝试手动提取高质量的特征，例如：图案中直线的数量、直线角度、直线之间的位置关系、

图案的面积等等。试着以尽量少的特征获得更好的分类性能。提取的特征越少，性能越好，分数越高。

4. (10 points) 在上一问中，使用决策树进行分类。本质上决策树就是一系列规则，尝试将决策树训练得到的模型使用文本或者图形将这一系列规则描述出来，观察是否符合预期 (例如: 有一条水平和一条垂直线的是加号)，从而更加深入地了解该模型。

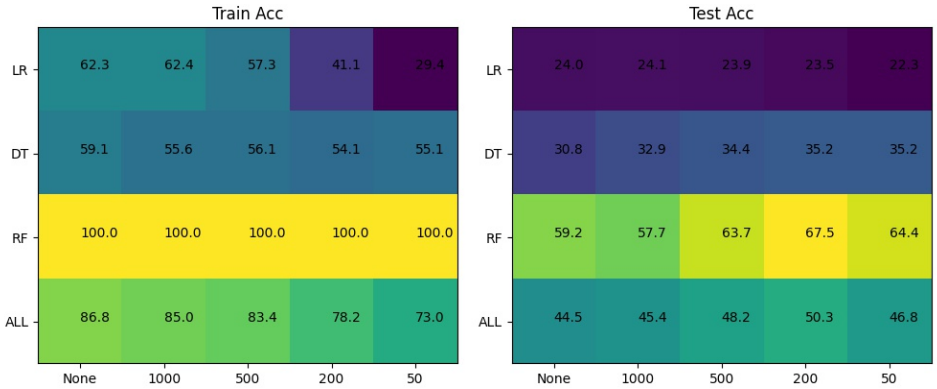
解：

1. 先执行 “generate_samples.py” 生成训练测试图片，再执行 “train_p0.py” 对这些数据进行训练和预测。使用默认参数得到的结果如下（样本数量为 2000）



分析性能图可以发现随机森林在该问题下的性能更好，训练集准确率可以做到 100%，降维到 200 维左右时测试集准确率更高。对于对数几率回归，降维后的维数越低，训练集准确率越低，而测试集准确率则变化不大。对于决策树，降维后训练集准确率和测试集准确率都有一些不大的波动。

尝试将对数几率回归、决策树和随机森林进行集成，对同一个样本，以三者中出现次数更多的那个预测值作为集成模型的预测值（若三者的预测值都不相同则随机选取一个模型的预测值输出），并将样本容量增加到 4000 后再进行一次测试（本来是想增加到 10000 或者 20000 的，但训练的时间开销实在难以承受），结果如下

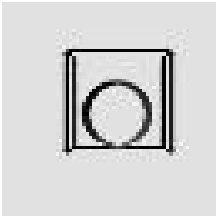


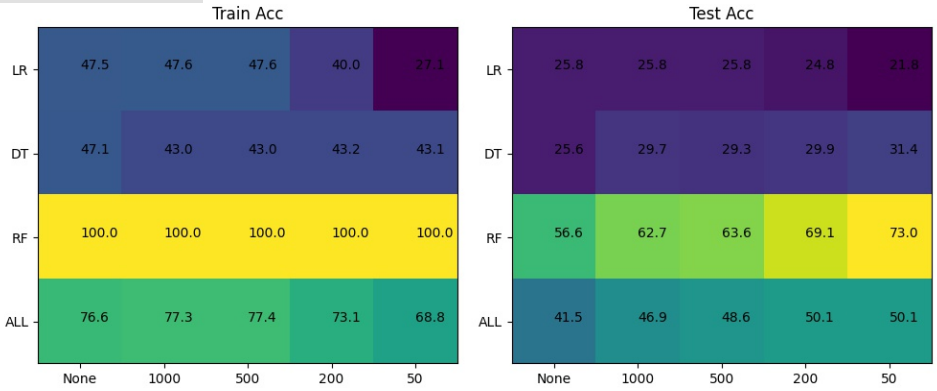
可以看出训练数据量增大后，对数几率回归和决策树的训练集准确率都有所下降，但测试集准确率则有一定提高。随机森林的训练集准确率依然为 100%，测试集准确率有明显提高，说明数据量增大会提升性能。而新增的集成方法表现并不如随机森林，但比其他两种方法要好。

集成模型的相关代码如下

```
if algo == "ALL":
    modelList = [LogisticRegression(
        multi_class="multinomial", C=10.0,
        solver="lbfgs", max_iter=5000
    ), DecisionTreeClassifier(
        max_depth=10
    ), RandomForestClassifier(
        n_estimators=500
    )]

    pred_train_ys_list = []
    pred_test_ys_list = []
    for model in modelList:
        model.fit(train_xs, train_ys)
        pred_train_ys_list.append(model.predict(train_xs))
        pred_test_ys_list.append(model.predict(test_xs))

    pred_train_ys = []
    pred_test_ys = []
    for i in range(0, len(train_ys)):
        if pred_train_ys_list[0][i] == pred_train_ys_list[1][i]:
            pred_train_ys.append(pred_train_ys_list[0][i])
        elif pred_train_ys_list[1][i] == pred_train_ys_list[2][i]:
            pred_train_ys.append(pred_train_ys_list[1][i])
        elif pred_train_ys_list[0][i] == pred_train_ys_list[2][i]:
            pred_train_ys.append(pred_train_ys_list[0][i])
        else:
            pred_train_ys.append(pred_train_ys_list[random.randint(0, 2)][i])

        if pred_test_ys_list[0][i] == pred_test_ys_list[1][i]:
            pred_test_ys.append(pred_test_ys_list[0][i])
        elif pred_test_ys_list[1][i] == pred_test_ys_list[2][i]:
            pred_test_ys.append(pred_test_ys_list[1][i])
        elif pred_test_ys_list[0][i] == pred_test_ys_list[2][i]:
            pred_test_ys.append(pred_test_ys_list[0][i])
        else:
            pred_test_ys.append(pred_test_ys_list[random.randint(0, 2)][i])

    pred_train_ys = np.array(pred_train_ys)
    pred_test_ys = np.array(pred_test_ys)
```

2. 先将图像变为轮廓图，由于变换后的图像只保留了边的信息，故不太容易判断目标的中心位置并裁剪，所以直接使用轮廓图进行训练，变换后的轮廓图示例和测试结果如下





和之前的测试结果对比可以发现，变换为轮廓图后，对数几率回归和决策树的训练集和测试集准确率均有所下降。而随机森林的训练集准确率依然为 100%，经过降维处理后的测试集准确率总体有所提高，与之前不同的是，表格中测试集准确率最高的不再是降维到 200 维了，而是降维到 50 维。集成方法的训练集准确率有所下降，而降维后的测试集准确率总体有所提升，但依然不如单独的随机森林性能好。

对图像进行变换的代码如下

```
        a = np.array(Image.open(os.path.join(
            "figs", split, "class{}-{}.jpg".format(c, i)
        )).convert('L')).astype('float')

        depth = 10.
        grad = np.gradient(a)
        grad_x, grad_y = grad

        grad_x = grad_x * depth / 100.
        grad_y = grad_y * depth / 100.
        A = np.sqrt(grad_y ** 2 + grad_y ** 2 + 1)
        uni_x = grad_x / A
        uni_y = grad_y / A
        uni_z = 1. / A

        vec_el = np.pi / 2.2
        vec_az = np.pi / 4
        dx = np.cos(vec_el) * np.cos(vec_az)
        dy = np.cos(vec_el) * np.sin(vec_az)
        dz = np.sin(vec_el)

        b = 255 * (dx * uni_x + dy * uni_y + dz * uni_z)
        b = b.clip(0, 225)

        im = Image.fromarray(b.astype('uint8'))

        im.save(os.path.join(
```

3. 考虑直线斜率、直线间夹角和直线之间的交点是否为顶点。直线数量和图案面积在图片被边缘裁切时会发生改变，故暂不考虑。对于 Class0，无直线；对于 Class1，相交直线间呈 90 度，且在顶点处相交，有竖线或横线；对于 Class2，相交直线之间呈 108 度，且在顶点处相交；对于 Class3，相交直线之间呈 120 度，且在顶点处相交；对于 Class4，相交直线之间呈 60 度，且在顶点处相交，且有一条横线；对于 Class5，相交直线之间呈 60 度，且在顶点处相交，且有一条竖线；对于 Class6，相交直线间的角度同时包含 36 度和 108 度，且在顶点处相交；对于 Class7，相交直线间呈 90 度，但不在顶点处相交，有竖线或横线；对于 Class8，相交直线间呈 90 度，但不在顶点处相交，无竖线和横线；对于 Class9，相交直线间呈 90 度，且在顶点处相交，有竖线或横线。可利用以上特征进行分类。

4. 决策树可依次按是否有直线，相交的直线交点是否为边的顶点，相交直线之间的角度，是否有横线，是否有竖线进行划分（不一定每个特征都要用到）。划分结果参照上一问中每个 Class 的特征，可得决策树训练得到的模型如下：
   无直线的是 Class0；
   相交的直线交点不是边的顶点，且有横线或竖线的是 Class7；
   相交的直线交点不是边的顶点，且没有横线或竖线的是 Class8；

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44	45	46	47

Figure 2: 悬崖行走问题示例图

相交的直线交点是边的顶点，相交直线间呈 90 度，且有横线或竖线的是 Class1；
相交的直线交点是边的顶点，相交直线间呈 90 度，且没有横线和竖线的是 Class9；
相交的直线交点是边的顶点，相交直线间呈 108 度的是 Class2；
相交的直线交点是边的顶点，相交直线间呈 120 度的是 Class3；
相交的直线交点是边的顶点，相交直线间呈 60 度，且有横线的是 Class4；
相交的直线交点是边的顶点，相交直线间呈 60 度，且无横线有竖线的是 Class5；
相交的直线交点是边的顶点，相交直线间的角度同时包含 36 度和 108 度的是 Class6；

三. (30 points) **编程题：强化学习**

悬崖行走问题定义如下: 问题环境为如图 2所示的 4 * 12 方格，智能体初始在 36 号位置，每次可上下左右移动一格，遇到墙则原地不动。每次移动获得-1 奖励。37-46 号位置为悬崖，若移动至悬崖，获得-100 奖励并回到原点 36 号位置。47 号位置为终点，运行至此为结束。总奖励为各步奖励之和。问题参数：$\epsilon = 0.1, \alpha = 0.5, \gamma = 1$。

请在悬崖行走问题中分别实现 Sarsa 算法与 Q-learning 算法，并分别输出运用两种算法训练得到的路径。本题代码文件夹在 “problem3-code” 目录下。问题环境可自己写，也可直接利用 “cliff_walking.py” 文件中的 Env 类。**请在作答中贴出核心代码文件和结果，以方便批改**。

解：

Sarsa 算法实现如下

```
def sarsa():
    # Q值矩阵
    qTable = defaultdict(lambda: [0.0, 0.0, 0.0, 0.0])

    envs = Env(3, 0)

    ob = envs.getPosition()
    action, _ = getBestAction(qTable, ob, envs)
    for i in range(0, 100000):
        next_ob, transition = envs.transition(action)
        flag = (transition == -100 or next_ob == 47)
        if np.random.rand() < epsilon:
            action_ = random.choice(envs.getStateAction(next_ob))
        else:
            action_, _ = getBestAction(qTable, next_ob, envs)
        oldQ = qTable[str(ob)][action]
        if flag:
            newQ = transition
        else:
            newQ = transition + discountFactor * qTable[str(next_ob)][action_]
        qTable[str(ob)][action] += learningRate * (newQ - oldQ)
        ob = next_ob
        action = action_
        if flag:
            envs.reset()
            ob = envs.getPosition()
            action, _ = getBestAction(qTable, ob, envs)

    envs.reset()
    ob = envs.getPosition()
    path = [ob]
    while not envs.isGoal():
        action, _ = getBestAction(qTable, ob, envs)
        ob, _ = envs.transition(action)
        path.append(ob)

    print("Sarsa:")
    print(path)
```

Q-learning 算法实现如下

```python
def qLearning():
    # Q值矩阵
    qTable = defaultdict(lambda: [0.0, 0.0, 0.0, 0.0])

    envs = Env(3, 0)

    for i in range(0, 10000):
        ob = envs.getPosition()
        if np.random.rand() < epsilon:
            action = random.choice(envs.getStateAction(ob))
        else:
            action, _ = getBestAction(qTable, ob, envs)

        next_ob, transition = envs.transition(action)
        flag = (transition == -100 or next_ob == 47)
        oldQ = qTable[str(ob)][action]
        if flag:
            newQ = transition
        else:
            _, maxQ = getBestAction(qTable, next_ob, envs)
            newQ = transition + discountFactor * maxQ
        qTable[str(ob)][action] += learningRate * (newQ - oldQ)
        if flag:
            envs.reset()

    envs.reset()
    ob = envs.getPosition()
    path = [ob]
    while not envs.isGoal():
        action, _ = getBestAction(qTable, ob, envs)
        ob, _ = envs.transition(action)
        path.append(ob)

    print("Qlearning:")
    print(path)
```

问题参数和上述算法用到的选取最佳可用动作的 getBestAction 方法

```python
# 参数
epsilon = 0.1
learningRate = 0.5
discountFactor = 1


def getBestAction(qTable, ob, envs):
    stateQ = qTable[str(ob)]
    actions = envs.getStateAction(ob)
    maxList = []
    maxValue = stateQ[actions[0]]
    maxList.append(actions[0])
    for i in actions[1:]:
        if stateQ[i] > maxValue:
            maxList.clear()
            maxValue = stateQ[i]
            maxList.append(i)
        elif stateQ[i] == maxValue:
            maxList.append(i)
    return random.choice(maxList), maxValue
```

cliff_walking.py 中的改动如下，新增了获取可用动作的 getStateAction 方法和获取当前位置的 getPosition 方法

```python
def getStateAction(self, state):
    # 左上角
    if state == 0:
        return [1, 3]
    # 右上角
    if state == 11:
        return [1, 2]
    # 左下角
    if state == 36:
        return [0, 3]
    # 右下角
    if state == 47:
        return [0, 2]
    # 左边界
    if state in [12, 24]:
        return [0, 1, 3]
    # 右边界
    if state in [23, 35]:
        return [0, 1, 2]
    # 上边界
    if state in range(1, 11):
        return [1, 2, 3]
    # 下边界
    if state in range(37, 47):
        return [0, 2, 3]

    return [0, 1, 2, 3]

def getPosition(self):
    return self.state
```

输出结果（路径）如下