作业 3: Aliens 游戏

季千焜（221300066、qkjiai@smail,nju.edu.cn）

(南京大学 人工智能学院, 南京 210093)

摘 要: 本次作业目的是使用监督学习来模仿人玩游戏的动作。需要对于现有的特征提取方法，收集训练数据，尝试三种以上的学习方法，撰写学习方法的介绍，报告性能对比，并尝试修改特征提取方法，得到更好的学习性能。

关键词: 监督学习、特征提取方法、RandomForest 算法、NaiveBayes 算法、SimpleLogistic 算法

1 任务一

对于现有的特征提取方法，收集训练数据，尝试三种以上的学习方法，撰写学习方法的介绍，报告性能对比。

我的游戏策略如下：开局时右移到地图中间偏右侧的空窗区定点射击，但漏掉的外星人较多，当外星人下降高度后，采用移动攻击的策略，期间需要躲避炸弹，当只剩最后一个外星人时，采用追击策略将其击落，并尽可能使整个过程时间最短。接下来将对得到的训练数据进行学习。

1.1 学习方法的介绍

选择了 RandomForest 算法、NaiveBayes 算法、SimpleLogistic 算法，接下来将分别介绍。.

1.1.1 RandomForest 算法

随机森林算法是一种基于集成学习的思想，利用多个决策树来进行分类或回归的算法。它的基本原理是：对于一个给定的数据集，随机森林算法会生成多个不同的子集，每个子集都是从原始数据集中有放回地随机抽取的。然后，对于每个子集，随机森林算法会构建一棵决策树，这棵决策树也是随机的，即在每个节点处，随机森林算法会从所有的特征中随机选择一部分特征，然后根据这些特征来划分数据。这样，随机森林算法就得到了一个由多棵决策树组成的森林。对于一个新的输入，随机森林算法会让每棵决策树都对其进行预测，然后根据预测的结果进行投票，最终输出票数最多的结果作为随机森林算法的预测。

1.1.1.1 随机森林算法的优点是：

它可以有效地处理高维数据，因为它会随机地选择特征，从而降低了特征之间的相关性。

它可以有效地处理缺失值和异常值，因为它会利用多棵决策树的投票机制，从而减少了单棵决策树的偏差和方差。

它可以有效地防止过拟合，因为它会利用多棵决策树的集成效应，从而提高了泛化能力。

它可以有效地提供特征重要性的评估，因为它会统计每个特征在决策树中的分裂次数，从而反映了特征的贡献度。

1.1.1.2 随机森林算法的缺点是：

它的计算量较大，因为它需要生成和训练多棵决策树，这会消耗更多的时间和空间。

它的可解释性较差，因为它的预测结果是基于多棵决策树的投票，而不是基于单棵决策树的规则，这会使得难以理解随机森林算法的内部逻辑。

1.1.1.3 随机森林算法的主要参数是：

树的数量，即随机森林算法中决策树的个数。树的数量越多，随机森林算法的性能越好，但是也会增加

计算量和过拟合的风险。

样本的数量，即随机森林算法中每棵决策树的训练数据的个数。样本的数量越多，每棵决策树的性能越好，但是也会增加每棵决策树之间的相关性。

特征的数量，即随机森林算法中每棵决策树在每个节点处随机选择的特征的个数。特征的数量越多，每棵决策树的复杂度越高，但是也会增加每棵决策树的方差。.

1.1.2 NaiveBayes 算法

朴素贝叶斯算法是一种基于概率的分类算法，它利用贝叶斯定理来计算给定样本特征下，样本属于不同类别的概率，并选择概率最大的类别作为分类结果。它的基本原理是：假设样本有 n 个特征，分别为 x1, x2, …, xn，样本的类别为 y，那么根据贝叶斯定理，样本属于类别 y 的概率为：

$P(y|x1,x2,...,xn)=P(x1,x2,...,xn)P(x1,x2,...,xn|y)P(y)$

其中，P(y)是类别 y 的先验概率，即在所有样本中，类别 y 的比例；P(x1, x2, …, xn)是样本特征的边缘概率，即在所有样本中，具有这些特征的样本的比例；P(x1, x2, …, xn|y)是样本特征的条件概率，即在类别 y 的样本中，具有这些特征的样本的比例。

1.1.2.1 朴素贝叶斯算法的核心假设是：

样本的各个特征之间是相互独立的，即一个特征的取值不会影响另一个特征的取值。这个假设虽然在实际情况中往往不成立，但是却大大简化了计算，使得朴素贝叶斯算法可以快速地处理高维数据。基于这个假设，样本特征的条件概率可以分解为各个特征的条件概率的乘积，即：

$P(x1,x2,...,xn|y)=P(x1|y)P(x2|y)...P(xn|y)$

因此，样本属于类别 y 的概率可以写为：

$P(y|x1,x2,...,xn)=P(x1,x2,...,xn)P(x1|y)P(x2|y)...P(xn|y)P(y)$

由于样本特征的边缘概率对于所有的类别都是相同的，所以在比较不同类别的概率时，可以忽略它，只需要比较分子部分的大小。因此，朴素贝叶斯算法的分类规则是：

$y\hat{}=argymaxP(x1|y)P(x2|y)...P(xn|y)P(y)$

其中，y^是预测的类别，argmaxy 表示求使得后面的表达式最大的 y 的值。

1.1.2.2 朴素贝叶斯算法的优点是：

它可以有效地处理高维数据，因为它只需要计算各个特征的条件概率和类别的先验概率，而不需要考虑特征之间的关系。

它可以有效地处理缺失值和异常值，因为它可以根据已有的特征来计算概率，而不需要完整的特征向量。

它可以有效地防止过拟合，因为它基于概率的统计，而不是基于复杂的规则，这会使得它更加稳健和泛化。

1.1.2.3 朴素贝叶斯算法的缺点是：

它的核心假设是样本的各个特征之间是相互独立的，这在实际情况中往往不成立，这会导致概率的估计不准确，从而影响分类的效果。

它的概率的计算依赖于样本的分布，如果样本的分布不平衡，或者某些特征的取值很少，那么概率的计算会出现偏差或者零概率的情况，这会影响分类的效果。

1.1.2.4 朴素贝叶斯算法的主要参数是：

特征的类型，即样本的各个特征是离散的还是连续的。对于离散的特征，可以直接计算条件概率和先验概率；对于连续的特征，需要假设特征服从某种分布，如正态分布，然后根据分布的参数来计算条件概率和先验概率。

平滑的方法，即当某些特征的取值很少或者没有出现时，如何避免出现零概率的情况。常用的方法有拉普拉斯平滑，即在计算概率时，给每个特征的取值加上一个小的常数，从而使得概率不为零。

1.1.3 SimpleLogistic 算法

简单逻辑回归算法是一种基于逻辑回归的分类算法，它利用 LogitBoost 算法来优化逻辑回归模型的参数。

它的基本原理是：假设样本有 n 个特征，分别为 x1, x2, …, xn，样本的类别为 y，y 的取值为 0 或 1，那么样本属于类别 1 的概率为：

P(y=1 | x1,x2,...,xn)=1+e−f(x1,x2,...,xn)1

其中，f(x1, x2, …, xn)是一个线性函数，即：

f(x1,x2,...,xn)=w0+w1x1+w2x2+...+wnxn

其中，w0, w1, …, wn 是模型的参数，需要通过数据来估计。SimpleLogistic 算法的目标是找到一组最优的参数，使得样本属于类别 1 的概率尽可能接近真实的类别。

1.1.3.1 SimpleLogistic 算法的核心思想是：

使用 LogitBoost 算法来迭代地更新参数，每次迭代都会增加一个简单的线性回归模型，从而提高模型的拟合能力。LogitBoost 算法的具体步骤如下：

初始化参数为 0，即 w0 = w1 = … = wn = 0。

对于每个类别 y，重复以下步骤，直到达到最大迭代次数或者分类误差不再下降为止：

计算每个样本的权重，权重等于样本属于类别 y 的概率乘以样本不属于类别 y 的概率，即：

wi=P(y | xi)(1−P(y | xi))

计算每个样本的残差，残差等于样本的真实类别减去样本属于类别 y 的概率，即：

ri=yi−P(y | xi)

使用加权最小二乘法，根据样本的特征、权重和残差，拟合一个简单的线性回归模型，即：

h(xi)=b0+b1xi

更新参数，将拟合的线性回归模型加到原来的线性函数上，即：

f(xi)=f(xi)+h(xi)

计算分类误差，分类误差等于所有样本的权重和残差的乘积的绝对值之和，即：

E=i=1∑n | wiri |

输出最终的参数，即 w0, w1, …, wn。

1.1.3.2 SimpleLogistic 算法的优点是：

它可以有效地处理高维数据，因为它使用线性函数来表示样本的概率，而不需要考虑特征之间的关系。

它可以有效地防止过拟合，因为它使用 LogitBoost 算法来控制迭代的次数和分类误差，从而避免模型过于复杂。

它可以有效地提供特征选择的功能，因为它可以根据每个特征在线性函数中的系数的大小，来判断特征的重要性。

1.1.3.3 SimpleLogistic 算法的缺点是：

它的核心假设是样本的概率服从逻辑分布，这在实际情况中往往不成立，这会导致概率的估计不准确，从而影响分类的效果。

它的计算量较大，因为它需要对每个类别都进行迭代，而且每次迭代都需要拟合一个线性回归模型，这会消耗更多的时间和空间。

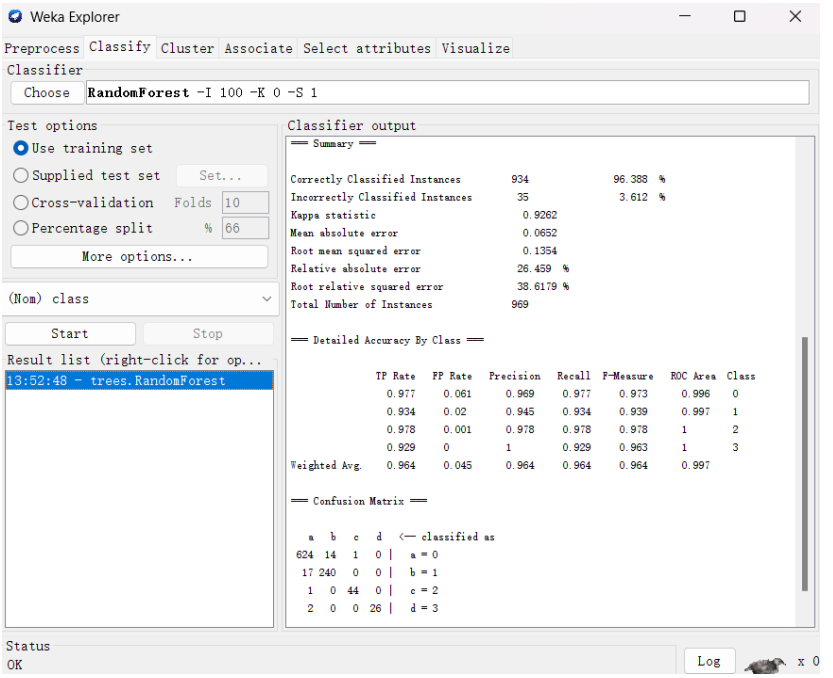
1.1.3.4 SimpleLogistic 算法的主要参数是：

最大迭代次数，即 LogitBoost 算法中每个类别的最多迭代次数。最大迭代次数越大，模型的拟合能力越强，但是也会增加计算量和过拟合的风险。

分类误差的阈值，即 LogitBoost 算法中判断是否停止迭代的条件。分类误差的阈值越小，模型的拟合能力越强，但是也会增加计算量和过拟合的风险。

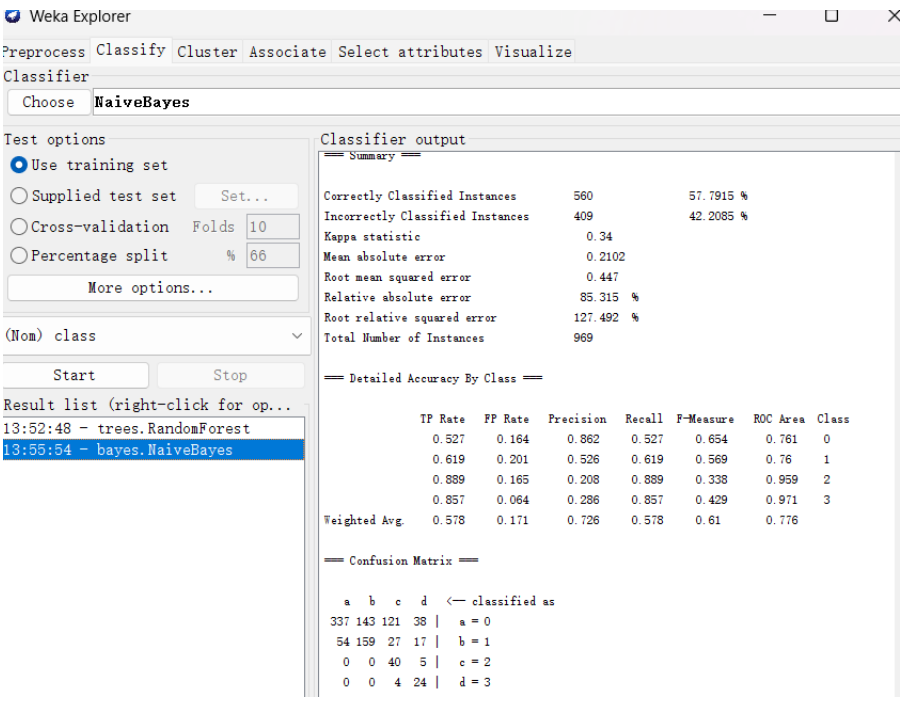
1.2 性能对比

1.2.1 RandomForest 算法



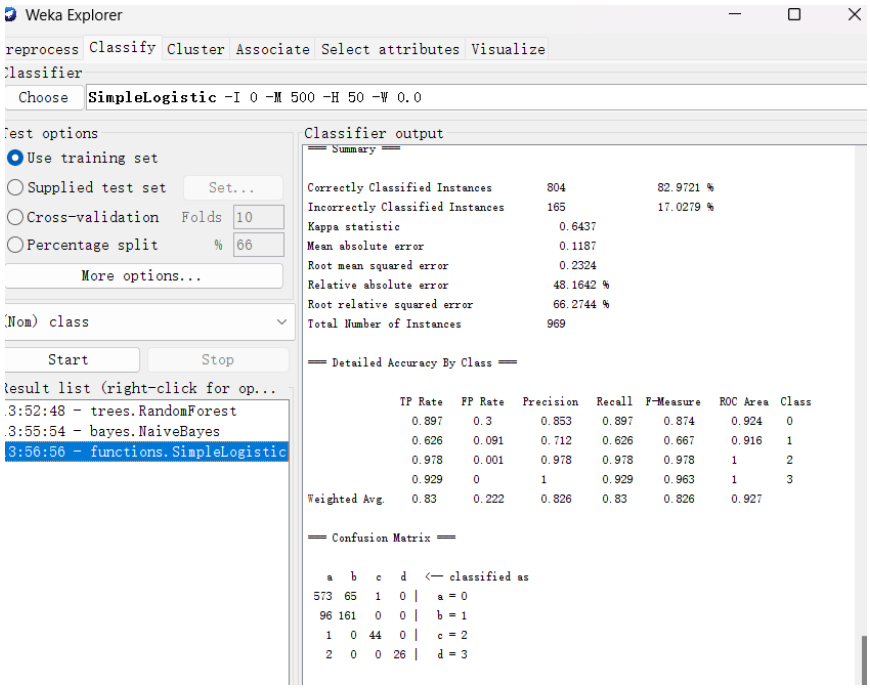
多次运行 Test.java，观察精灵的行为，发现精灵会一开局精灵移动到最右边的位置并简单地采用连续射击的最短间隔，并且 Alien 减少后精灵也会追击目标或者对炸弹做出闪避，有时候也会触发追击，效果比较好，需要 450-650ticks 就能获胜。

1.2.2 NaiveBayes 算法



多次运行 Test.java，观察精灵的行为，精灵的动作具有比较大的随机性，每次测试间的规律性并不强。除了一开局移动到最右方并开始连续射击以外，后面的动作每次都不太相同。通过观察精灵的移动规律，发现其运动有一定的追击性质。只剩最后一个目标时，精灵也会开始四处移动，追击最后一个目标。然而精灵依然没有学习到躲避炸弹的策略。游戏时间最快只需 422ticks，最慢需要 700ticks 以上。

1.2.3 SimpleLogistic 算法



多次运行 Test.java，观察精灵的行为，精灵一开始也是移动到最右侧并开始连续射击，然后开始无规则运动，并且射击偶尔会出现较长间隔。不能确定这种无规则运动是不是为了追击，也不能确定是否与躲避炸弹有关，但是能观察到精灵确实有因运动而躲避过炸弹，一般需要 600-800ticks 才能完成游戏。

1.2.4 对比结果

对比三种算法的运行结果，将学习效果并从好到坏排序，我认为 RandomForest 算法最好，它会向右移动射击，并且躲避和追击都学到了；再次是 NaiveBayes 算法，学到了向右移、连续射击和追击，但不会躲避炸弹；最差的是 SimpleLogistic 算法，学到了向左移和连续射击，并且有疑似追击和躲避的行为。

2 任务二

尝试修改特征提取方法，得到更好的学习性能。

2.1.1 思路

旧版的特征提取中直接使用了 448 个方格内的物体属性作为特征来进行学习。这样做虽然最大化了特征的完整性，但是过于具体，缺乏抽象性，可能不具备良好的学习性质。同时游戏状态是不会重复的，因而收集到的特征参数也不尽相同，这样的朴素知识获取方法，近一步降低了训练模型的抽象性与泛化能力，非常容易出现过拟合情况。

因此特征提取方法的改进就是，通过人为方法提取一些更加“抽象”同时自认为“有用”的特征，让机器根据这些我们人类组织出来的经验进行学习，以期望获得更好的学习成果。

2.1.2 改进特征

首先修改 Recorder 类中的 featureExtract 函数存在一处错误，将 4 个属性分别写入 4 个不同的位置。

```
        // 4 states
        feature[448] = obs.getGameTick();
        feature[449] = obs.getAvatarSpeed();
        feature[450] = obs.getAvatarHealthPoints();
        feature[451] = obs.getAvatarType();

        return feature;
    }
```

由任务一可知，需要着重从有助于学习躲避炸弹和追击的角度对特征提取方法进行修改。具体操作如下：

1. 将 featureExtract 函数中 feature 数组的长度改为 455，然后在 featureExtract 函数中加入对为所在的列是否有炸弹和精灵左右两侧外星人数量的差值这两个特征的提取和存储。
2. 修改 datasetHeader 函数。

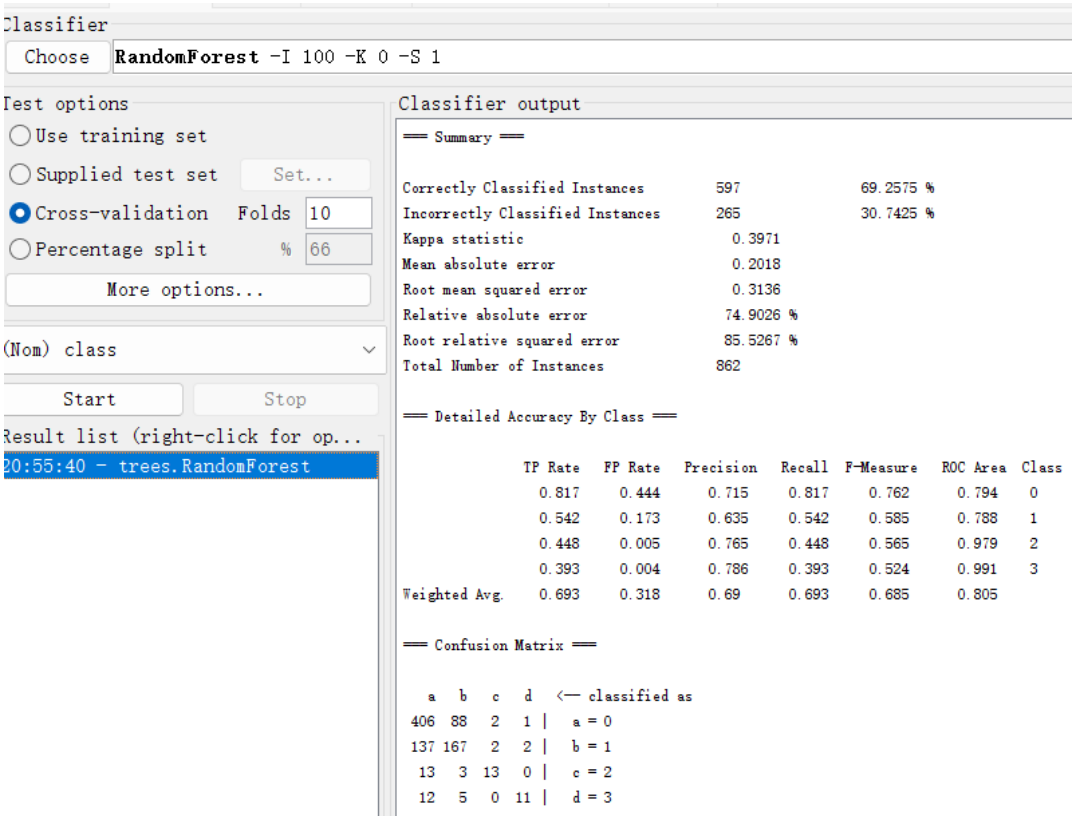
修改后的代码如下：

```
public static double[] featureExtract(StateObservation obs) {
    double[] feature = new double[456];  // 448 + 4 + 1(class) + 3 (left, right, left - right)

    // 448 locations
    int[][] map = new int[32][14];
    // Extract features
    boolean bomb = false;
    int left = 0;
    int right = 0;
    double avatar_X = obs.getAvatarPosition().x;
    LinkedList<Observation> allobj = new LinkedList<>();

    // Combine all positions into a single list
    allobj.addAll(obs.getImmovablePositions().stream().flatMap(List::stream).collect(Collectors.toList()));
    allobj.addAll(obs.getMovablePositions().stream().flatMap(List::stream).collect(Collectors.toList()));
    allobj.addAll(obs.getNPCPositions().stream().flatMap(List::stream).collect(Collectors.toList()));

    for (Observation o : allobj) {
        Vector2d p = o.position;
        int x = (int) (p.x / 25);
        int y = (int) (p.y / 25);
        map[x][y] = o.itype;
        if (o.itype == Types.TYPE_FROMAVATAR && Math.abs(p.x - avatar_X) <= 25) {
            bomb = true;
        }
        if (obs.getNPCPositions() != null && o.category == Types.TYPE_NPC) {
            double NPC_X = o.position.x;
            if (NPC_X < avatar_X) {
                left++;
            } else if (NPC_X > avatar_X) {
                right++;
            }
        }
    }

    // Flatten the map into the feature array
    for (int y = 0; y < 14; y++) {
        for (int x = 0; x < 32; x++) {
            feature[y * 32 + x] = map[x][y];
        }
    }

    // 8 states
    feature[448] = obs.getGameTick();
    feature[449] = obs.getAvatarSpeed();
    feature[450] = obs.getAvatarHealthPoints();
    feature[451] = obs.getAvatarType();
    feature[452] = bomb ? 1000.0 : -1000.0;
    feature[453] = left;
    feature[454] = right;
    feature[455] = left - right;
    return feature;
}
```

```
public static Instances datasetHeader(){
    FastVector attInfo = new FastVector();
    // 448 locations
    for(int y=0; y<14; y++){
        for(int x=0; x<32; x++){
            Attribute att = new Attribute("object_at_position_x=" + x + "_y=" + y);
            attInfo.addElement(att);
        }
    }
    Attribute att = new Attribute("GameTick" ); attInfo.addElement(att);
    att = new Attribute("AvatarSpeed" ); attInfo.addElement(att);
    att = new Attribute("AvatarHealthPoints" ); attInfo.addElement(att);
    att = new Attribute("AvatarType" ); attInfo.addElement(att);
    att = new Attribute("bomb" ); attInfo.addElement(att);
    att = new Attribute("left" ); attInfo.addElement(att);
    att = new Attribute("right" ); attInfo.addElement(att);
    att = new Attribute("left - right" ); attInfo.addElement(att);
```

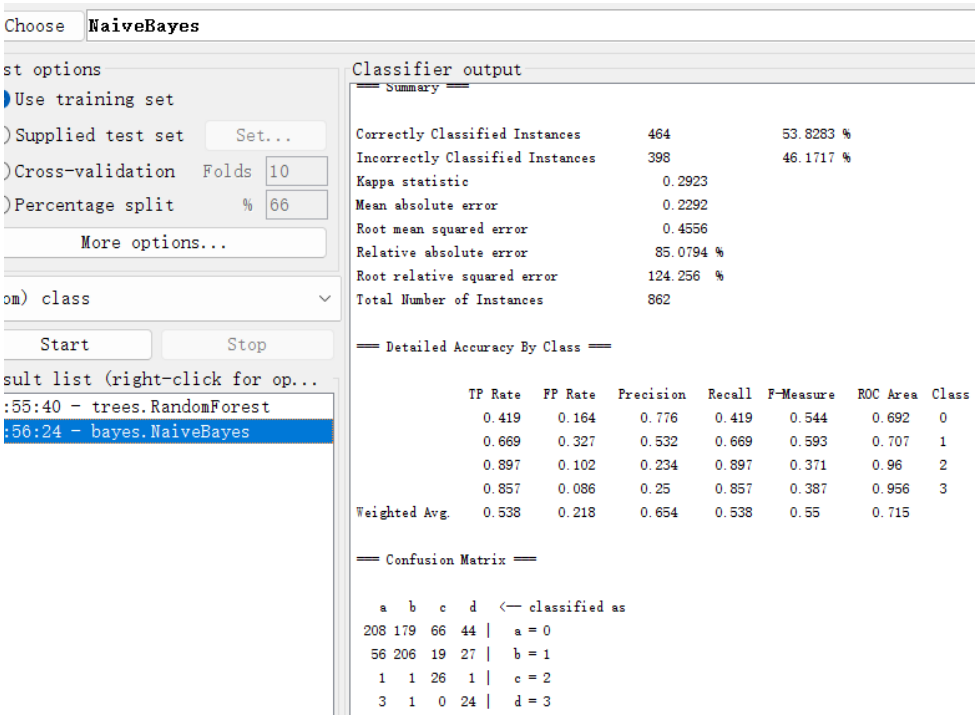
2.1.3 评估性能

2.1.3.1 RandomForest 算法：



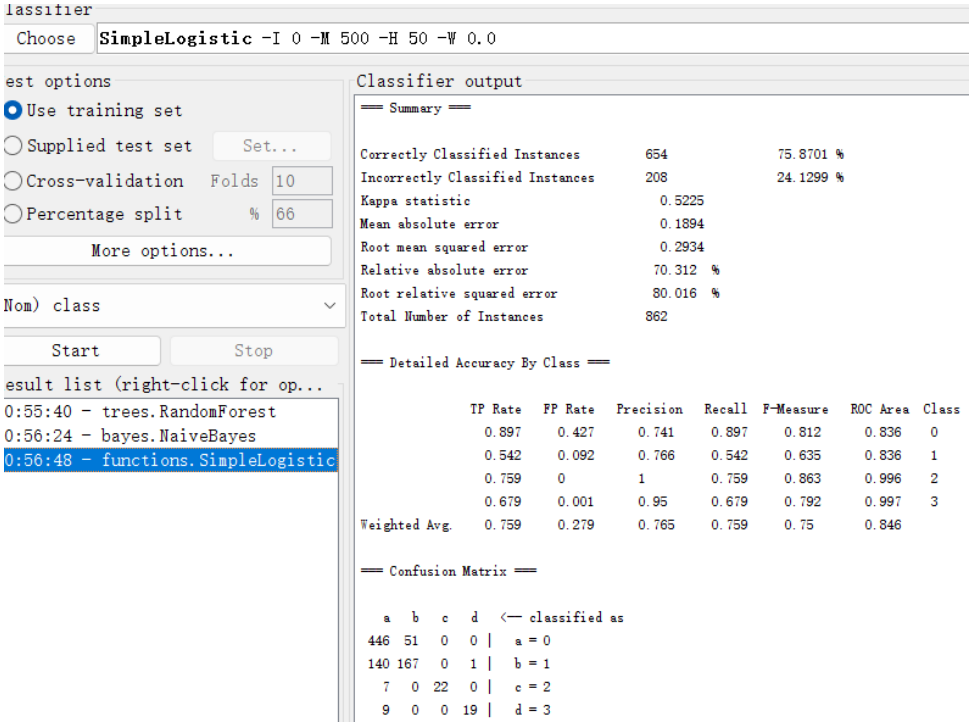
多次运行 Test.java，观察精灵的行为，发现在一开局精灵会移动到偏右的位置，并且不断射击。然后会左右小范围移动，然后继续保持静止。这是由于我在进行游戏时先向右移动导致的，而出现的左右移动可能是触发了追击策略。但是当只剩下最后几个目标时又不会触发追击策略，而是依旧停留在原地一直射击。一般需要 450-500ticks 完成游戏。

2.1.3.2 NaiveBayes 算法：



多次运行 Test.java，发现精灵在一开始会移动到最左侧，并开始连续射击，然后会在右半部分小范围左右移动。当剩余一两个外星人时，精灵则会迅速地不规则地大范围地左右移动，是追击策略发挥作用了，但是运动太无规律导致击杀效率很低，往往可能因此而失败。游戏时间需要 700-800ticks。

2.1.3.3 SimpleLogistic 算法：



多次运行 Test.java，观察精灵的行为，一开始精灵会移动到右侧空窗处并开始连续射击，然后开始左右移动追击。但是追击效果会差一些，当只剩下一两个外星人而且它们都在较低层时，射击会出现较长间隔，导致有可能因不能击杀最后一个目标而失败。对闪避炸弹的学习也一般，有时可以躲开，有时则不能。一般需要 550-850ticks 才能完成游戏。

2.1.4 对比修改前

修改后的三种算法学习均获得了不同程度的提升，尤其是其模仿人类的能力大大增强了。三者在开局阶段表现都很好，都会向右移并连续射击，之后也会进行追击。但是当只剩下一两个外星人时，三种学习方法下精灵的表现各不相同，但效果都有些差，常常导致游戏失败，可能是由于数据收集过少（只有一轮游戏）且样本中没有涉及到只剩下一两个外星人并且都在较低层的情况的处理，所以无法准确地应对。还有一个问题是特征提取太杂且不够优化。

References:

[1] https://zhuanlan.zhihu.com/p/22212026.

[2] https://zhuanlan.zhihu.com/p/120043523.

[3] https://blog.csdn.net/sheziqiong/article/details/1253135552.