一、 实验题目

Logistic Regression 算法实现

决策树算法实现

二、 实验内容

1. 算法原理

Logistic Regression

之前学过的分类算法 PLA 适用于二元分类，也就是每一次只能把数据分成两类。对于输入 x，我们通过计算更新得到 w，使得 W*X 可以得到两类结果，一类大于 0，一类小于等于 0.

当然，它也可以变身多元分类器，但是可能出现一个样本属于多个类别的情况。

另外，它在可以把数据区分好的时候，就不会继续更新。因为此时它不会找到一个分错的点。这样就导致，可能的权重有无数种，我们找到的不一定好。

这次的逻辑回归和 PLA 其实有些相似。因为它的目的也是找到最合适权重。

但是，它引入一个 sigmoid 函数，把 W.X 映射到 0 到 1 之间。

在 PLA 中，只要 W.X 大于 0，则为 1，否则为-1.这样我们并不知道 W.X 最终的结果是多少。60 和 100 其实可以算是一样的结果。但是实际上，100 的分数比 60 要高很多。我们更愿意拿到 100.Logistic Regression 的目的，就是明确 100 比 60 好。

其实就是，当我们计算出 W.X 大于 0 后，PLA 就会直接把它分成 1，但是，LR 会继续计算，直到令它等于正无穷。

Sigmoid 的函数是$\theta(s) = \frac{1}{1+e^{-s}}$

在这里，W.X 就是 sigmoid 函数里面的 s。

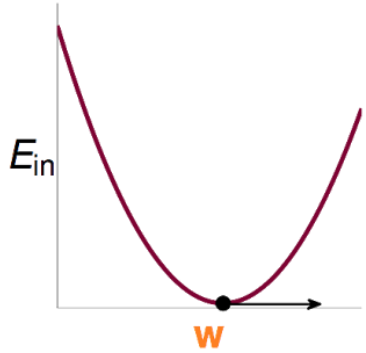
成本函数用的是交叉熵，为什么 Linear Regression 用到的平方误差，其实是因为就 error 是一个关于 w 的非凸函数(non-convex),非凸函数由于存在很多个局部最小点，因此很难去做最优化(解全局最小)。所以 Logistic Regression 没有使用平方误差来定义 error，而是使用极大似然法来估计模型的参数。

极大似然法基本思想：当从模型总体随机抽取 n 组样本观测值后，最合理的参数估计量应该使得从模型中抽取该 n 组样本观测值的概率最大，而不是像最小二乘估计法旨在得到使得模型能最好地拟合样本数据的参数估计量。

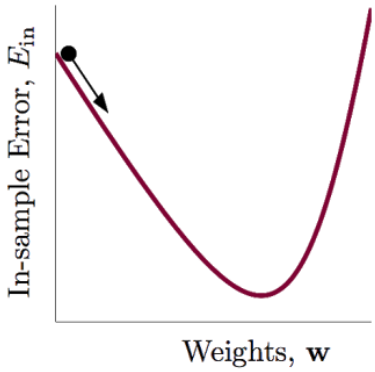
最终我们得到的 error 的梯度是$\nabla \mathrm{Err} = \sum_{n=1}^{N}(\frac{1}{1+e^{-w^T x}} - y_n)(x_{n,i})$

接下来就可以根据梯度下降法更新权重了。

梯度下降法就是使得成本函数最小的一种算法。



可以看出 cost function 是连续，可微的凸函数。按照之前 Linear Regression 的逻辑，由于它是凸函数，如果我们能解出一阶微分(梯度)为 0 的点，这个问题就解决了。



我们初始化的结果可能离山谷很远，山谷就是梯度为 0，我们要把小球滚下山谷。小球的方向应该是它所在点的梯度方向。

方向决定了，那么小球每次滚多远呢？

滚太远，可能导致震荡，太近又很慢。所以要选择一个合适的长度。

最终权重更新：

$$W_{\text{new}} < -W_{old} - \eta.\nabla Err$$

知道成本函数很小，或者迭代次数足够多。

Decision Tree

决策树是一种很直观的算法。对于每一个节点都有是和不是两种选择。其实整个决策树就是一个有 if 和 then 组合起来的东西。

它的可读性高，效率也高，但是很容易过拟合。

我们要实现的决策树有三种：

ID3 中选取使信息增益最大的特征

C4.5 中选取使信息增益比最大的特征

CART 分类树中选取基尼指数最小的特征及其对应的切分点

决策树的构建基本步骤：

1. 开始，遍历每一个特征，找最好的特征点
2. 分割成多份
3. 删掉用过的特征。继续遍历，分割
4. 直到节点的标签一致或者没有可用特征

连续值可以离散化，变成多个特征值。

因为容易过拟合，所以需要剪枝。

前剪枝是在构建树过程中剪枝，效率更高。后剪枝是在建好树之后剪枝，更具有全局观。

2. 伪代码

1) 初始的 logistic regression

```
训练:
    计算error=(sigmoid(W.X) - Y) * X
    假如error小于设定阈值:
        退出
    否则:
        更新权值: weights = weights - eta * error


测试:
    得到测试集的评测指标
```

2) 使用十折交叉验证的 logistic regression

```
数据处理：
    把训练数据集分成十份，一份作验证集，其余作训练集
    在第一维补充1


训练：
    计算error=(sigmoid(W.X) - Y) * X
    假如error小于设定阈值：
        退出
    否则：
        更新权值：weights = weights - eta * error


十折交叉验证：
    每次计算一遍用训练集得到的weights能够使验证集有多大accuracy
    如果大于最大accuracy：
        口袋中的权值换成新的
    得到最好权值
    得到测试集的评测指标
```

3) 加入最佳学习率

```
最佳学习率：
    选择一个合适的学习次数和起始学习率
    每次学习率比上次大十倍
        假如在新学习率得到的error最小：
            把这个学习率放进口袋

```

4) 对每个训练样本更新权重

```
训练：
    对于每一个样本：
        计算error=(sigmoid(W.X) - Y) * X
        更新权值：weights = weights - eta * error
```

5) 随机选择样本更新权重（也即随机选择每个样本出现的顺序）

```
训练：
    随机选择每一个样本：
        计算error=(sigmoid(W.X) - Y) * X
        更新权值：weights = weights - eta * error
        把优化的样本删去，避免再次对他更新
```

6) 改变学习率的梯度下降

```
随着梯度下降而减小的学习率：
    在初始梯度下降算法：
        学习率：4.0 / (1.0 + 训练次数) + 0.01
    在对单个样本更改权值的算法：
        学习率：4.0 / (1.0 + 训练次数 + 取的第n个样本) + 0.01
```

Decision Tree

(1) 初始的 ID3，C4.5 和 CART

```
ID3:
    对于一个数据集：
        计算原始的经验熵
        计算加入新节点之后会有的条件熵
        信息增益 = 经验熵 - 条件熵
        选择信息增益最大的特征作为决策点
        数据集更新成通过决策点划分出来的几个部分
        递归直到数据集里面只有一种标签或者没有特征可以再选

C4.5：
    对于一个数据集：
        计算原始的经验熵
        计算加入新节点之后会有的条件熵
        信息增益 = 经验熵 - 条件熵
        信息增益率 = 信息增益 / 数据集关于新节点的熵
        选择信息增益率最大的特征作为决策点
        数据集更新成通过决策点划分出来的几个部分
        递归直到数据集里面只有一种标签或者没有特征可以再选
CART：
    对于一个数据集：
        计算基尼系数
        选择基尼系数最小的特征作为决策点
        数据集更新成通过决策点划分出来的几个部分
        递归直到数据集里面只有一种标签或者没有特征可以再选
```

(2) 根据每列均值离散特征值

(3) 根据最大值和最小值的中间值离散特征值

```
离散特征值：
    均值：
        计算每一列均值
        大于均值：
            即为1
        小于或等于均值：
            即为0
    最大最小值的中间值：
        计算每一列最大最小值
        求中间值
            大于中间值：
                即为1
            小于或等于中间值：
                即为0
```

(4) 前剪枝

```
前剪枝：
    计算错误率
    计算假如加入新节点的错误率
    如果不加入错误率小：
        停止，不加入
    如果加入的错误率小：
        加入新节点
```

3. 关键代码截图（带注释）

Logistic Regression

1) 初始的 logistic regression

训练

```
def trainLog(dataMatIn, classLabels,opts,weights):
    '''
        LR训练
    '''
    dataMatrix = mat(dataMatIn)
    labelMat = mat(classLabels).T
    m,n = shape(dataMatrix)
    eta = opts['eta']
    maxCycles = opts['maxCycles']

    for k in range(maxCycles):
        # 用梯度下降算法
        if opts['optimizeType'] == 'gradDescent':
            output =(sigmoid(dataMatrix * weights.T))
            error = (output - labelMat).T * dataMatrix
            # 假如误差很小，则不用继续更新了
            if abs(sum(error)) < 0.000001:
                break
            else:
                weights = weights - eta * error
    return weights
```

测试（在 main 函数中）

```
def main():
    testDataSet,testLabels = testData()
    W = ones((1,9))
    m,n = shape(testDataSet)
    weights = zeros((1,n))

    # 学习率，学习次数固定
    opts = {'eta': 0.01, 'maxCycles': 100, 'optimizeType': 'gradDescent'}
    for j1 in range(3):
        # 不同初始化权重
        if(j1 == 0):
            print "------------初始化为0------------"
            weights = zeros((1,n))
        elif(j1 == 1):
            print "------------初始化为1------------"
            weights = ones((1,n))
        else:
            print "------------初始化为随机数------------"
            weights = numpy.random.randn(1,n)
        trainMat,trainLabel,valiMat,valiLabel = trainData()
        W = trainLog(trainMat, trainLabel,opts,weights)
        myLablels = classifyAll(testDataSet, W)
        # 四种指标
        acc=Accuracy(testLabels,myLablels)
        rec=Recall(testLabels,myLablels)
        pre=Precision(testLabels,myLablels)
        f1=F1(testLabels,myLablels)
        print "Accuracy = " , acc
        print "Recall = " ,rec
        print "Precision = " , pre
        print "F1 = " , f1
```

2） 加入十折交叉验证

```
def trainData(i):
    '''
       训练集并采用交叉验证（10 folds）
    '''
    trainLabel = []
    valiLabel= []
    fr = open('train.csv')
    arrayLines = fr.readlines()
    # 十折cross validation
    trainMat = zeros((len(arrayLines) - len(arrayLines)/5,10))
    valiMat = zeros((len(arrayLines)/5,10))
    index = 0
    trainIndex = 0
    valiIndex = 0
    for line in arrayLines:
        dataMat = [];
        lineArr = line.strip().split(',')
        for i in range(9):
            dataMat.append(float(lineArr[i]))
        #添加常数1.0
        dataMat[0:0]=[1.0]
        # 训练集
        if(index < i or index >= i + len(arrayLines)/5):
            trainMat[trainIndex,:] = dataMat
            trainLabel.append((float)(lineArr[9]))
            trainIndex += 1
        # 验证集
        else:
            valiMat[valiIndex,:] = dataMat
            valiLabel.append((float)(lineArr[9]))
            valiIndex += 1
        index += 1
    #输出训练集及其标签，验证集及其标签
    return trainMat,trainLabel,valiMat,valiLabel
```

十折交叉验证：

```
# 十折交叉验证
for i in range(10):
    trainMat,trainLabel,valiMat,valiLabel = trainData(i)
    weights = trainLog(trainMat, trainLabel,opts,weights)
    myLablel = classifyAll(valiMat, weights)
    acc=Accuracy(valiLabel,myLablel)
    rec=Recall(testLabels,myLablel)
    pre=Precision(testLabels,myLablel)
    f1=F1(testLabels,myLablel)
    # 比较这次权重是否更好
    if acc >= best_acc and rec >= best_rec and pre >= best_pre and f1 >= best_f1:
        W = weights
        best_acc = acc
        best_rec = rec
        best_pre = pre
        best_f1 = f1
```

3) 加入最佳学习率

```
# 学习率从0.00001 到 10000 一共九级变化
for i in range(10):
    tempOpts = {'eta': 0.00001*pow(10,i), 'maxCycles': 100, 'optimizeType': 'gradDescent'}
    trainMat,trainLabel,valiMat,valiLabel = trainData(i)
    weights,error = trainLog(trainMat, trainLabel,opts,weights)
    myLablels = classifyAll(valiMat, weights)
    # 假如指标更好，则为更好的学习率
    if best_error > error:
        best_error = error
        opts = tempOpts
print opts
```

4) 类似 PLA 对每个样本进行权值更新

```
# 对每个样本更新权值
elif opts['optimizeType'] == 'plaGradDescent':
    for i in range(m):
        output = sigmoid(dataMatrix[i, :] * weights.T)
        error = output - labelMat[i, 0]
        weights = weights - eta * error * dataMatrix[i, :]
    output =(sigmoid(dataMatrix * weights.T))
    error = (output - labelMat).T * dataMatrix
    if abs(sum(error)) < 0.1:
        break
```

5) 随机梯度下降

```
# 随机下降算法
elif opts['optimizeType'] == 'stocGradDescent':
    dataIndex = range(m)
    for i in range(m):
        # 随机选择一个样本
        randIndex = int(random.uniform(0, len(dataIndex)))
        output = sigmoid(dataMatrix[randIndex, :] * weights.T)
        error = output - labelMat[randIndex, 0]
        weights = weights - eta * error * dataMatrix[randIndex, :]
        # 从列表删除这个更新过得样本
        del(dataIndex[randIndex])
    output =(sigmoid(dataMatrix * weights.T))
    error = (output - labelMat).T * dataMatrix
    if abs(sum(error)) < 0.1:
        break
```

6) 改变学习率的梯度下降

```
for k in range(maxCycles):
    # 梯度下降算法
    eta = 4.0 / (1.0 + k) + 0.01
    if opts['optimizeType'] == 'gradDescent':
        output =(sigmoid(dataMatrix * weights.T))
        error = (output - labelMat).T * dataMatrix
        if abs(sum(error)) < 0.1:
            break
        else:
            weights = weights - eta * error
    # 对每个样本更新权值
    elif opts['optimizeType'] == 'plaGradDescent':
        for i in range(m):
            eta = 4.0 / (1.0 + k + i) + 0.01
            output = sigmoid(dataMatrix[i, :] * weights.T)
            error = output - labelMat[i, 0]
            weights = weights - eta * error * dataMatrix[i, :]
        output =(sigmoid(dataMatrix * weights.T))
        error = (output - labelMat).T * dataMatrix
        if abs(sum(error)) < 0.1:
            break
```

Decision Tree

(1) 初始的 ID3，C4.5 和 CART

```
def calEnt(dataSet):
    """
        计算样本的熵
    """
    numEntries = len(dataSet)
    labelCounts = {}
    for featVec in dataSet:
        dataLabel = featVec[-1]
        # 属于两种标签样本数
        if dataLabel not in labelCounts.keys():
            labelCounts[dataLabel] = 0
        labelCounts[dataLabel] += 1
    entroy = 0.0
    # 计算熵
    for key in labelCounts:
        prob = float(labelCounts[key])/numEntries
        entroy -= prob * log(prob, 2)
    return entroy
```

```
        # 根据基尼系数得到CART
        if(tree == "CART"):
            for value in uniqueVals:
                subDataSet = splitDataSet(dataSet, i, value)
                # 这种类别比例
                prob = len(subDataSet)/float(len(dataSet))
                # 每种类别样本比例
                subProb = len(splitDataSet(subDataSet, -1, 0)) / float(len(subDataSet))
                # 计算基尼系数
                gini += prob * (1.0 - pow(subProb, 2) - pow(1 - subProb, 2))
            # 选择基尼系数小的
            if (gini < bestGini):
                bestGini = gini
                bestFeature = i
        # 根据信息增益得到ID3
        elif(tree == "ID3"):
            for value in uniqueVals:
                subDataSet = splitDataSet(dataSet, i, value)
                prob = len(subDataSet)/float(len(dataSet))
                # 计算熵
                newEntropy += prob * calEnt(subDataSet)
            # 信息增益
            infoGain = baseEntropy - newEntropy
            # 选择信息增益大的
            if (infoGain > bestInfoGain):
                bestInfoGain = infoGain
                bestFeature = i
        # 根据信息增益率得到C4.5
        else:
            for value in uniqueVals:
                subDataSet = splitDataSet(dataSet, i, value)
                prob = len(subDataSet)/float(len(dataSet))
                newEntropy += prob * calEnt(subDataSet)
                # 计算数据集下关于新节点的熵
                splitInfo += -prob * log(prob, 2)
            infoGain = baseEntropy - newEntropy
            if (splitInfo == 0):
                continue
            # 信息增益率
            infoGainRatio = infoGain / splitInfo
            # 选择信息增益率大的
            if (infoGainRatio > bestInfoGainRatio):
                bestInfoGainRatio = infoGainRatio
                bestFeature = i
```

```
def createTree(dataSet, dataLabel,tree):
    """
        递归构建决策树
    """
    flag = 0
    classList = [example[-1] for example in dataSet]
    if classList.count(classList[0]) == len(classList):
        if(tree == "ID3"):
            node1.append(len(classList))
        elif(tree == "C4.5"):
            node2.append(len(classList))
        else:
            node3.append(len(classList))
        # 类别完全相同，停止划分
        return classList[0]

    if len(dataSet[0]) == 1 or flag == 1:
        # 把叶子节点记录
        if(tree == "ID3"):
            node1.append(len(classList))
        elif(tree == "C4.5"):
            node2.append(len(classList))
        else:
            node3.append(len(classList))
        # 遍历完所有特征时返回出现次数最多的
        return majorityCnt(classList)
    bestFeat = chooseBestFeatureToSplit(dataSet,tree)
    bestFeatLabel = dataLabel[bestFeat]
    myTree = {bestFeatLabel:{}}
    del(dataLabel[bestFeat])
    # 得到列表包括节点所有的属性值
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        subLabels = dataLabel[:]
        # 递归建树
        myTree[bestFeatLabel][value] = createTree(splitDataSet(dataSet, bestFeat, value), subLabels,tree)
    return myTree
```

(2) 根据每列均值离散特征值

```
def trainData(k):
    #训练集
    fr = open('train.csv')
    arrayLines = fr.readLines()
    trainMat = []
    myMat = zeros((len(arrayLines),10))
    index = 0
    for line in arrayLines:
        dataMat = [];
        lineArr = line.strip().split(',')
        for i in range(10):
            dataMat.append(int(lineArr[i]))
        myMat[index,:] = dataMat
        index += 1

    meanList = []
    for i in range(len(myMat[0])):
        # 把一列的均值放进列表
        meanList.append(mean(myMat[:,i]))
        #meanList.append((max(myMat[:,i])+min(myMat[:,i]))/2.0)

    index = 0
    for i in range(len(myMat)):
        dataList = []
        for j in range(len(myMat[0])):
            # 假如大于均值为1
            if myMat[i][j] > meanList[j]:
                dataList.append(1)
            # 小于等于均值为0
            else:
                dataList.append(0)
        trainMat.append(dataList)
    trainLabel = [1,2,3,4,5,6,7,8,9]
    return trainMat,trainLabel,meanList
```

(3) 根据最大值和最小值的中间值离散特征值

```
def trainData(k):
    #训练集
    fr = open('train.csv')
    arrayLines = fr.readlines()
    trainMat = []
    myMat = zeros((len(arrayLines),10))
    index = 0
    for line in arrayLines:
        dataMat = [];
        lineArr = line.strip().split(',')
        for i in range(10):
            dataMat.append(int(lineArr[i]))
        myMat[index,:] = dataMat
        index += 1

    meanList = []
    for i in range(len(myMat[0])):
        # 把一列的最大值与最小值的中间值放进列表
        #meanList.append(mean(myMat[:,i]))
        meanList.append((max(myMat[:,i])+min(myMat[:,i]))/2.0)

    index = 0
    for i in range(len(myMat)):
        dataList = []
        for j in range(len(myMat[0])):
            # 假如大于中间值为1
            if myMat[i][j] > meanList[j]:
                dataList.append(1)
            # 小于等于中间值为0
            else:
                dataList.append(0)
        trainMat.append(dataList)
    trainLabel = [1,2,3,4,5,6,7,8,9]
    return trainMat,trainLabel,meanList
```

(4) 前剪枝

```
def errCount(classList):
    """
        计算错误分类个数
    """
    classCount = {}
    for vote in classList:
        if vote not in classCount.keys():
            classCount[vote] = 0
        classCount[vote] += 1
    sorted(classCount.iteritems(), key=operator.itemgetter(1),reverse = True)
    errorCount = 0
    # 多数投票，少的一方即为错误
    for value in classCount.values()[:-1]:
        errorCount += value
    return errorCount,len(classCount)

def prePurning(dataSet, dataLabel,tree):
    """
        前剪枝
    """
    error = 0
    bestFeat = chooseBestFeatureToSplit(dataSet,tree)
    featValues = [example[bestFeat] for example in dataSet]
    uniqueVals = set(featValues)
    for value in uniqueVals:
        splitData = splitDataSet(dataSet, bestFeat, value)
        classList = [example[-1] for example in splitData]
        errorCount,classCount = errCount(classList)
        error = error + (float)(errorCount)
    return error
```

在构建树中：

```
errorCount,classCount = errCount(classList)
# 不加新节点
beforeErr = errorCount + (2 * len(node)) * 1
aftErr = prePurning(dataSet, dataLabel,tree)
# 加入新节点
afterErr = aftErr + (2 * len(node) + 1) * 1
if beforeErr >= afterErr and classCount < 3:
    flag = 1

if len(dataSet[0]) == 1 or flag == 1:
    if(tree == "ID3"):
        node1.append(len(classList))
    elif(tree == "C4.5"):
        node2.append(len(classList))
    else:
        node3.append(len(classList))
```

三、 实验结果及分析

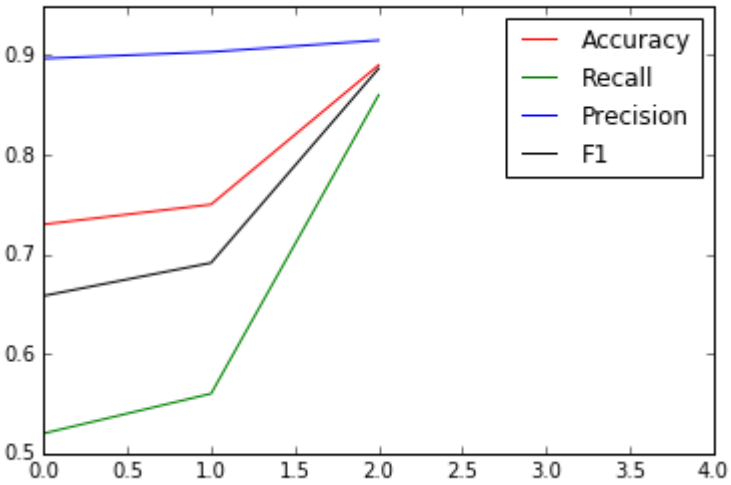
1. 实验结果展示示例（可图可表可文字，尽量可视化）

Logistic Regression

(1) 初始的 logistic regression

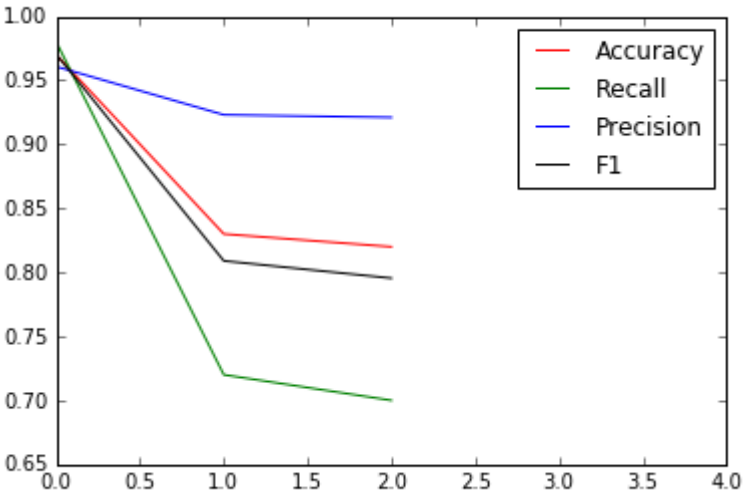
学习次数为 100，学习率为 0.1

```
------------初始化为0------------
Accuracy =  0.73
Recall =  0.52
Precision =  0.896551724138
F1 =  0.658227848101
------------初始化为1------------
Accuracy =  0.75
Recall =  0.56
Precision =  0.903225806452
F1 =  0.691358024691
------------初始化为随机数------------
Accuracy =  0.89
Recall =  0.86
Precision =  0.914893617021
F1 =  0.886597938144
```



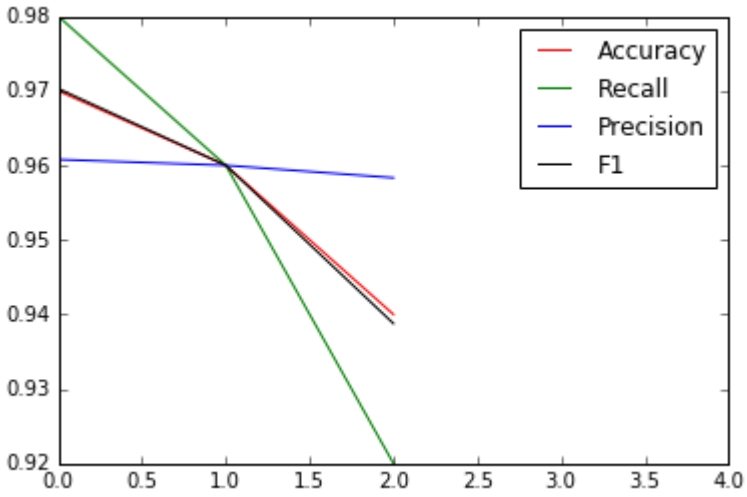
把学习次数增大到 800

```
------------初始化为0------------
Accuracy =  0.97
Recall =  0.98
Precision =  0.960784313725
F1 =  0.970297029703
------------初始化为1------------
Accuracy =  0.83
Recall =  0.72
Precision =  0.923076923077
F1 =  0.808988764045
------------初始化为随机数------------
Accuracy =  0.82
Recall =  0.7
Precision =  0.921052631579
F1 =  0.795454545455
```



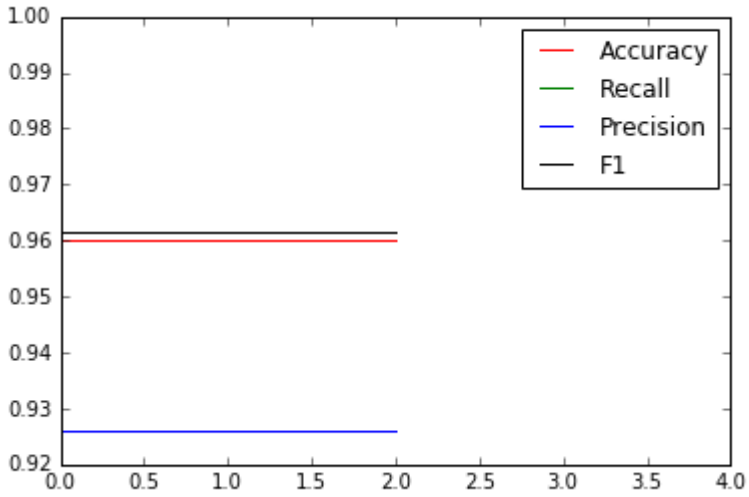
2） 加入十折交叉验证，学习率为 0.1，学习次数为 100

```
------------初始化为0------------
Accuracy =  0.97
Recall =  0.98
Precision =  0.960784313725
F1 =  0.970297029703
------------初始化为1------------
Accuracy =  0.96
Recall =  0.96
Precision =  0.96
F1 =  0.96
------------初始化为随机数------------
Accuracy =  0.94
Recall =  0.92
Precision =  0.958333333333
F1 =  0.938775510204
```



3） 加入最佳学习率

```
{'optimizeType': 'gradDescent', 'maxCycles': 500, 'eta': 0.01}
------------初始化为0------------
Accuracy =  0.96
Recall =  1.0
Precision =  0.925925925926
F1 =  0.961538461538
------------初始化为1------------
Accuracy =  0.96
Recall =  1.0
Precision =  0.925925925926
F1 =  0.961538461538
------------初始化为随机数------------
Accuracy =  0.96
Recall =  1.0
Precision =  0.925925925926
F1 =  0.961538461538
```



（可以看到，在学习次数为 500 的时候，最佳学习率是 0.01.为什么学习次数设成 500？因为数据集本来就不是一个大数据集，我们的目标是寻找 9 维的权重，所以没有必要设成几万的，这样浪费时间资源。100-1000 是比较合适的数。当然，按照我们的算法，学习率小，学习次数多容易得到好的结果，但是，在兼顾好结果的时候，我们也需要考虑一下资源消耗问题。）

4)

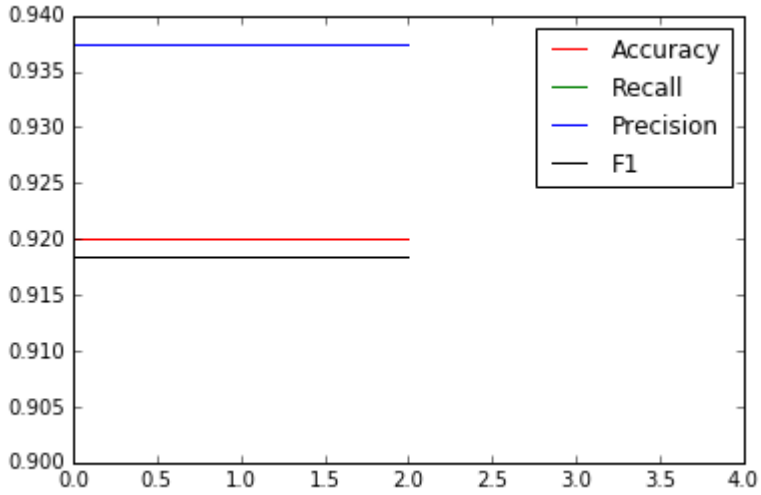
5) 加入对每个样本更新权值和随机更新权值

```
-------------find opts-------------
{'optimizeType': 'gradDescent', 'maxCycles': 100, 'eta': 0.001}
------------初始化为0------------
-------------gradDescent-------------
Accuracy =  0.91
Recall =  0.88
Precision =  0.936170212766
F1 =  0.907216494845
-------------placGradDescent-------------
Accuracy =  0.91
Recall =  0.88
Precision =  0.936170212766
F1 =  0.907216494845
-------------stocGradDescent:-------------
Accuracy =  0.91
Recall =  0.88
Precision =  0.936170212766
F1 =  0.907216494845
```

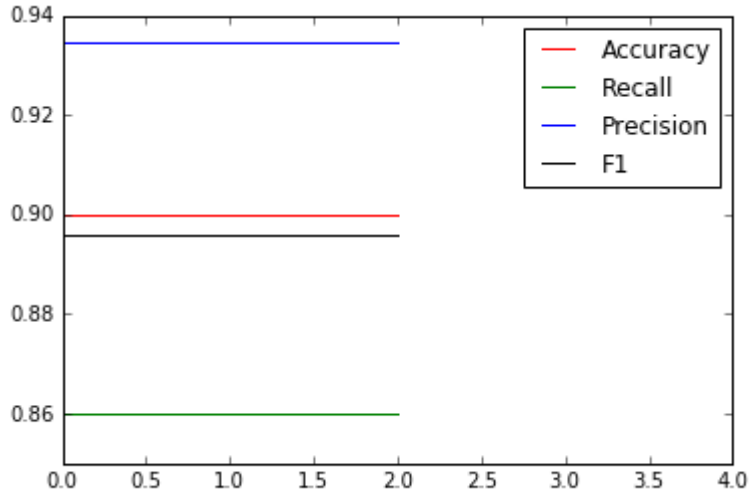
```
------------初始化为1------------
-------------gradDescent-------------
Accuracy =  0.88
Recall =  0.82
Precision =  0.931818181818
F1 =  0.872340425532
-------------placGradDescent-------------
Accuracy =  0.87
Recall =  0.78
Precision =  0.951219512195
F1 =  0.857142857143
-------------stocGradDescent:-------------
Accuracy =  0.89
Recall =  0.84
Precision =  0.933333333333
F1 =  0.884210526316
------------初始化为随机数------------
-------------gradDescent-------------
Accuracy =  0.84
Recall =  0.74
Precision =  0.925
F1 =  0.822222222222
-------------placGradDescent-------------
Accuracy =  0.84
Recall =  0.74
Precision =  0.925
F1 =  0.822222222222
-------------stocGradDescent:-------------
Accuracy =  0.84
Recall =  0.74
Precision =  0.925
F1 =  0.822222222222
```

（上面的结果是四个指标均优于之前最好的，然后才更换口袋里的权值。）

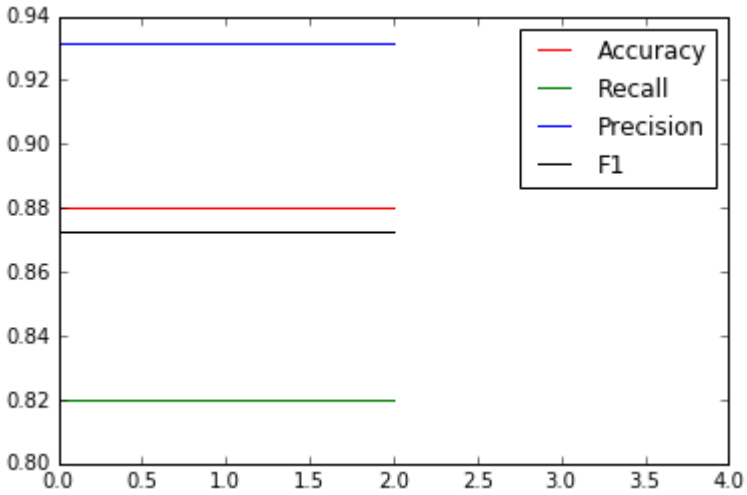
```
-------------find opts-------------
{'optimizeType': 'gradDescent', 'maxCycles': 100, 'eta': 0.001}
------------初始化为0------------
-------------gradDescent-------------
Accuracy =  0.92
Recall =  0.9
Precision =  0.9375
F1 =  0.918367346939
-------------plaGradDescent-------------
Accuracy =  0.92
Recall =  0.9
Precision =  0.9375
F1 =  0.918367346939
-------------stocGradDescent:-------------
Accuracy =  0.92
Recall =  0.9
Precision =  0.9375
F1 =  0.918367346939
```




```
------------初始化为1------------
-------------gradDescent-------------
Accuracy =  0.9
Recall =  0.86
Precision =  0.934782608696
F1 =  0.895833333333
-------------plaGradDescent-------------
Accuracy =  0.9
Recall =  0.86
Precision =  0.934782608696
F1 =  0.895833333333
-------------stocGradDescent:-------------
Accuracy =  0.9
Recall =  0.86
Precision =  0.934782608696
F1 =  0.895833333333
```



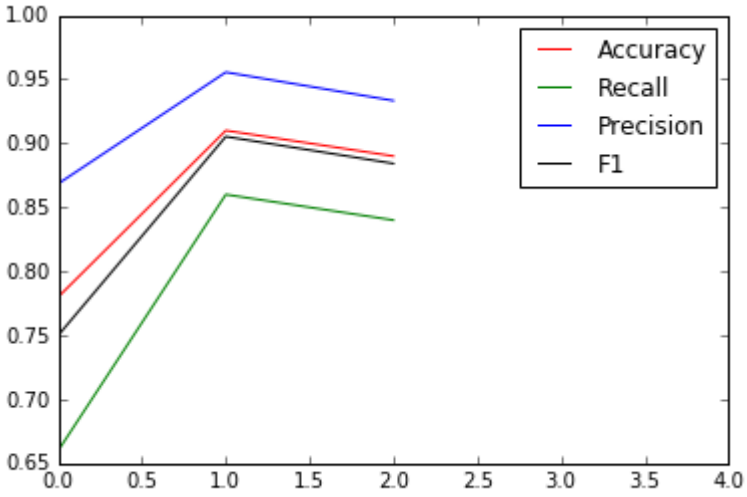
```
------------初始化为随机数------------
-------------gradDescent-------------
Accuracy =  0.88
Recall =  0.82
Precision =  0.931818181818
F1 =  0.872340425532
-------------plaGradDescent-------------
Accuracy =  0.88
Recall =  0.82
Precision =  0.931818181818
F1 =  0.872340425532
-------------stocGradDescent:-------------
Accuracy =  0.88
Recall =  0.82
Precision =  0.931818181818
F1 =  0.872340425532
```



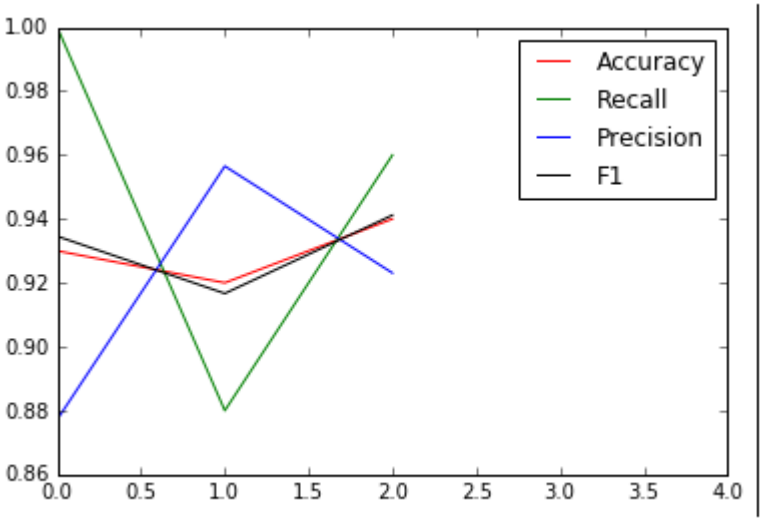
（准确率更高则更换口袋里的权值，从图里面，我们可以看到，三种算法得到的结果竟然一致，而且无论初始权值怎么变化，三种算法结果都是一致。这应该是因为数据集小，所以很快，就有了 1 的准确率，这样一来，后面计算的权值就不会被放进口袋，所以口袋里一直都是那个权值。）

6） 学习率随学习次数下降

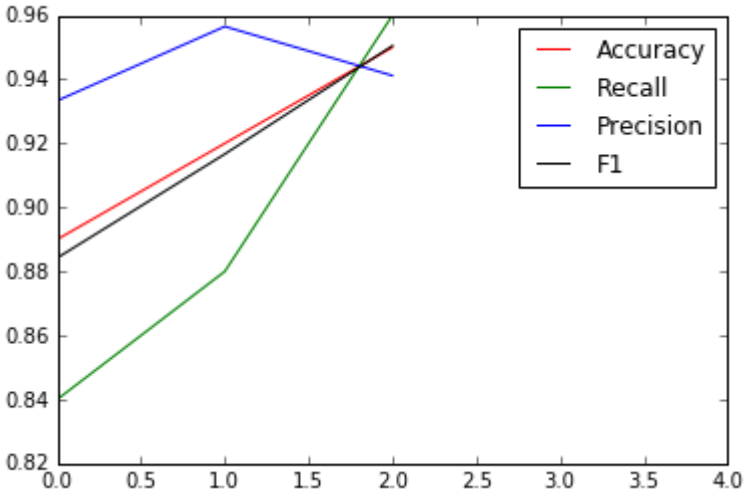
```
-------------find opts-------------
------------初始化为0------------
-------------gradDescent-------------
Accuracy =  0.78
Recall =  0.66
Precision =  0.868421052632
F1 =  0.75
-------------plaGradDescent-------------
Accuracy =  0.91
Recall =  0.86
Precision =  0.955555555556
F1 =  0.905263157895
-------------stocGradDescent:-------------
Accuracy =  0.89
Recall =  0.84
Precision =  0.933333333333
F1 =  0.884210526316
```




```
------------初始化为1------------
-------------gradDescent-------------
Accuracy =  0.93
Recall =  1.0
Precision =  0.877192982456
F1 =  0.934579439252
-------------plaGradDescent-------------
Accuracy =  0.92
Recall =  0.88
Precision =  0.95652173913
F1 =  0.916666666667
-------------stocGradDescent:-------------
Accuracy =  0.94
Recall =  0.96
Precision =  0.923076923077
F1 =  0.941176470588
```

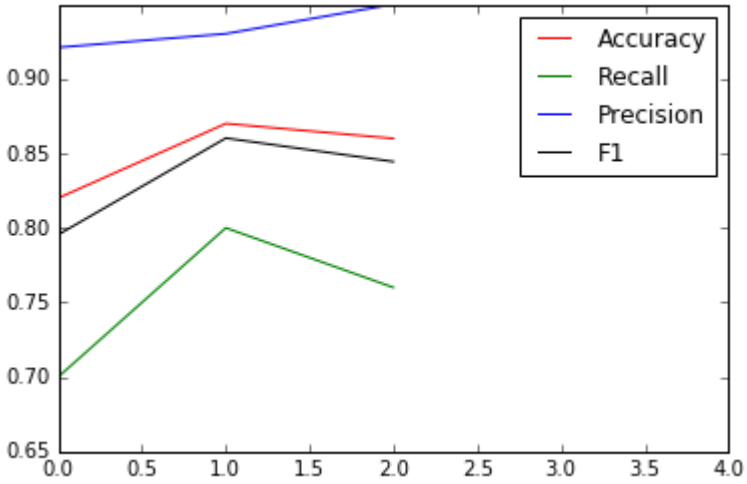


```
------------初始化为随机数------------
-------------gradDescent-------------
Accuracy =  0.89
Recall =  0.84
Precision =  0.933333333333
F1 =  0.884210526316
-------------plaGradDescent-------------
Accuracy =  0.92
Recall =  0.88
Precision =  0.95652173913
F1 =  0.916666666667
-------------stocGradDescent:-------------
Accuracy =  0.95
Recall =  0.96
Precision =  0.941176470588
F1 =  0.950495049505
```

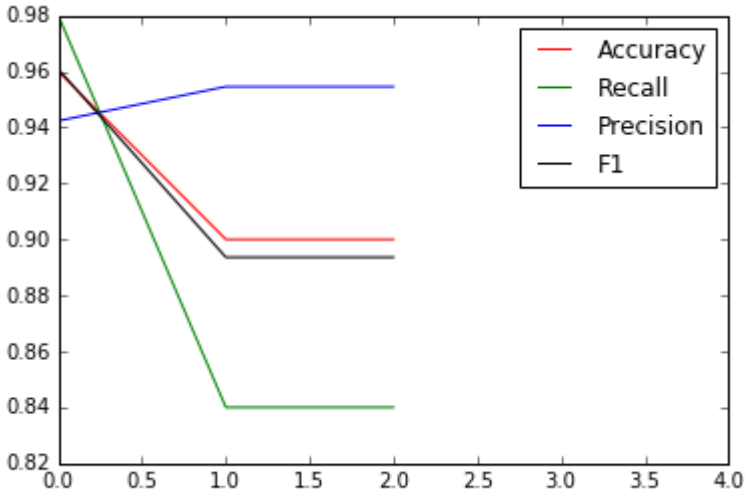


7）5 折交叉验证

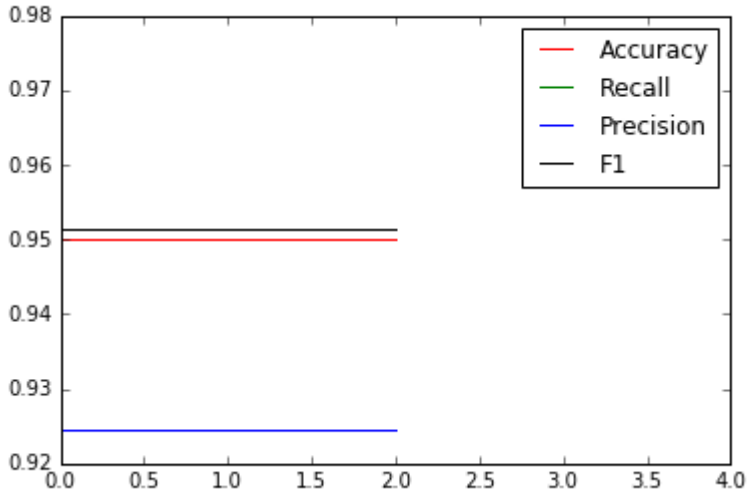
```
------------初始化为0------------
-------------gradDescent-------------
Accuracy =  0.82
Recall =  0.7
Precision =  0.921052631579
F1 =  0.795454545455
-------------plaGradDescent-------------
Accuracy =  0.87
Recall =  0.8
Precision =  0.93023255814
F1 =  0.860215053763
-------------stocGradDescent:-------------
Accuracy =  0.86
Recall =  0.76
Precision =  0.95
F1 =  0.844444444444
```




```
------------初始化为1------------
-------------gradDescent-------------
Accuracy =  0.96
Recall =  0.98
Precision =  0.942307692308
F1 =  0.960784313725
-------------plaGradDescent-------------
Accuracy =  0.9
Recall =  0.84
Precision =  0.954545454545
F1 =  0.893617021277
-------------stocGradDescent:-------------
Accuracy =  0.9
Recall =  0.84
Precision =  0.954545454545
F1 =  0.893617021277
```



```
------------初始化为随机数------------
-------------gradDescent-------------
Accuracy =  0.95
Recall =  0.98
Precision =  0.924528301887
F1 =  0.95145631068
-------------plaGradDescent-------------
Accuracy =  0.95
Recall =  0.98
Precision =  0.924528301887
F1 =  0.95145631068
-------------stocGradDescent:-------------
Accuracy =  0.95
Recall =  0.98
Precision =  0.924528301887
F1 =  0.95145631068
```



五折交叉验证和十折交叉验证是比较常用的两种交叉验证。两者在这个数据集上跑的结果相差不会太远。

我们可以看到，其实我们加了优化之后，对每个样本进行下降之后，结果可能反而不好一点。一句话，优化是理论上、大量反复试验中会好，但是不是一定一定在所有测试集上都有更好结果。

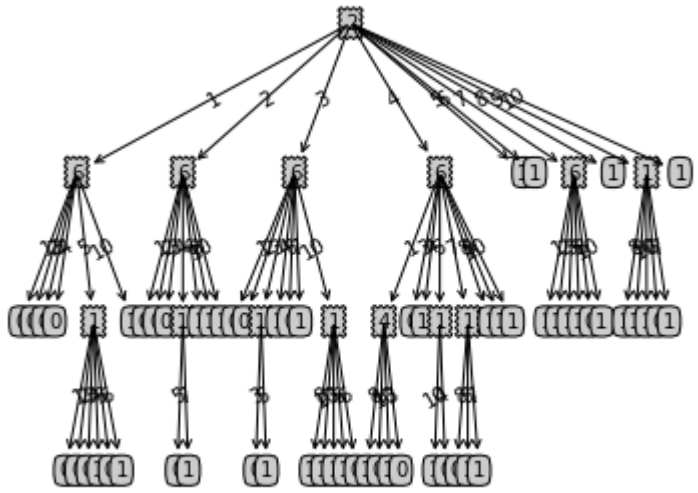
Decision Tree

(1) 初始的三种决策树，数值没有进行离散化

（画图用了《机器学习实战》中的代码）

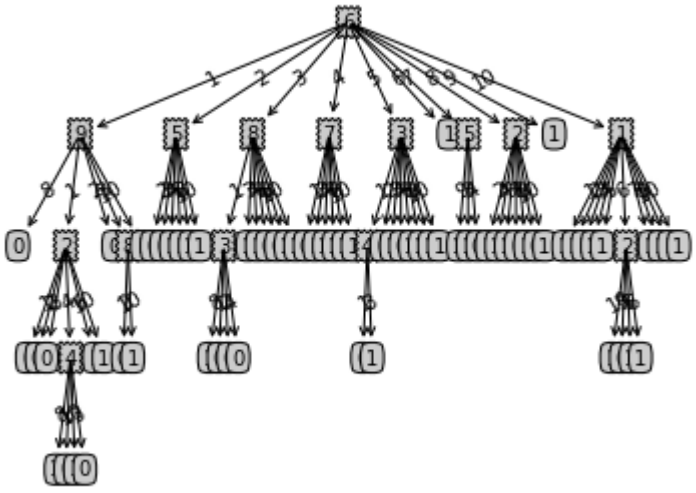
a) 惩罚为 1

```
-------------ID3------------
```



```
Accuracy =  0.81
Error =  0.79
```

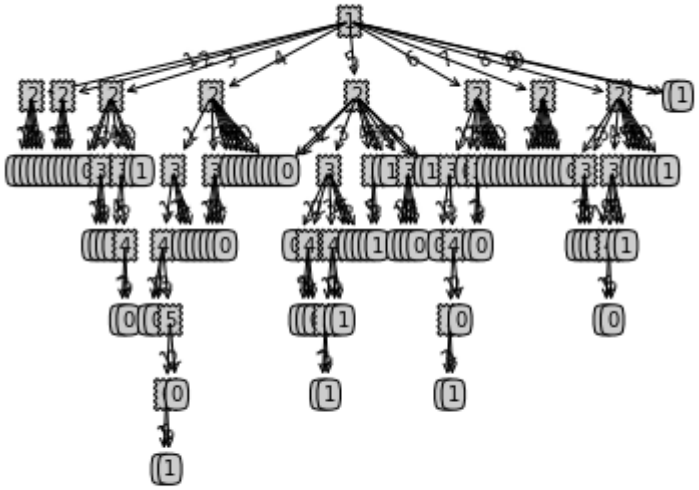
```
-------------C4.5------------
```



```
Accuracy =  0.81
Error =  0.19
```

```
-------------CART------------
```

In [166]:

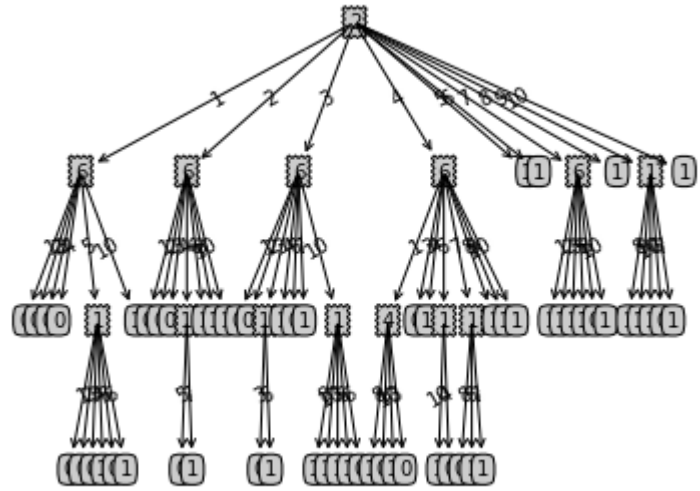


```
Accuracy =  0.83
Error =  1.14
三种决策树叶子节点个数
60 70 97
```

可以看到，叶子节点的个数好多啊。而且树很庞大。

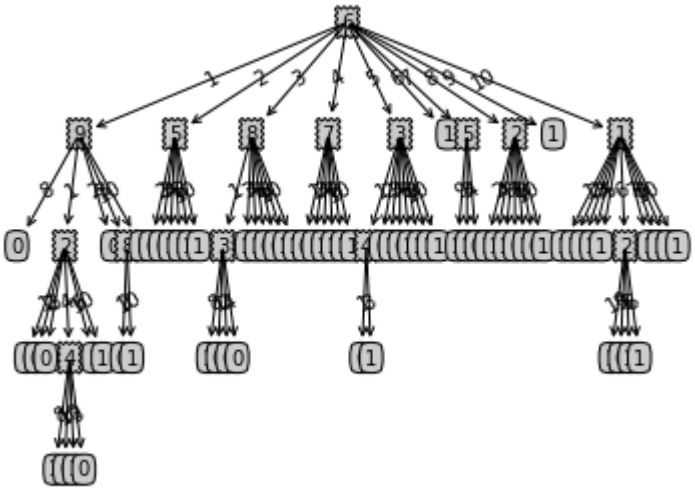
b) 惩罚为 0.5

```
-------------ID3------------
```

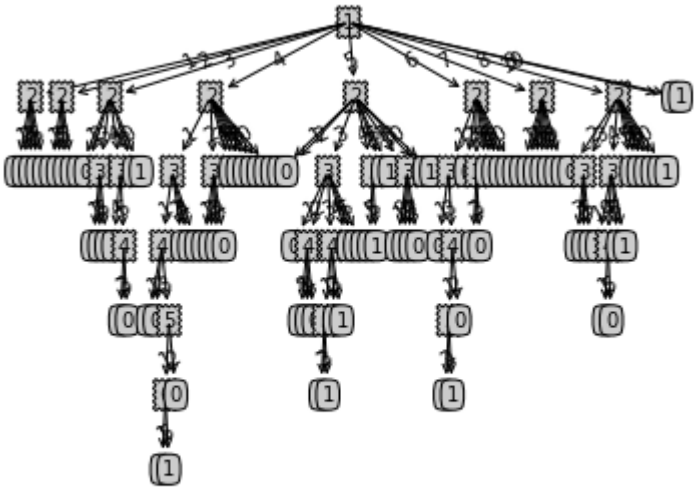


```
Accuracy =  0.81
Error =  0.49
```

```
-------------C4.5------------
```



```
Accuracy =  0.81
Error =  0.19

-------------CART------------
```

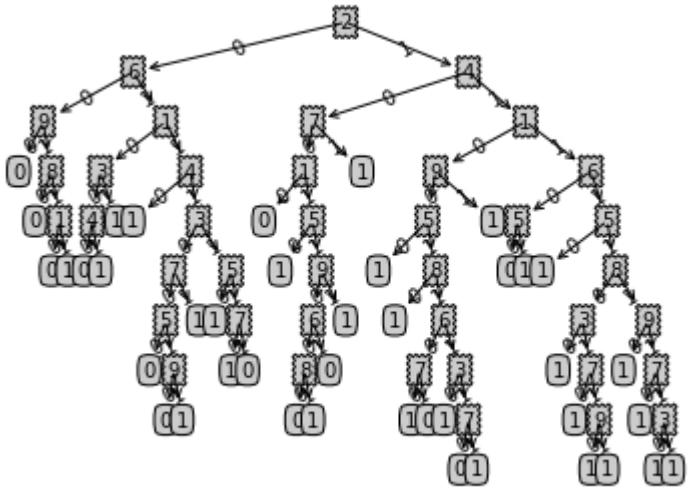


```
Accuracy =  0.83In [167]:
Error =  0.655
三种决策树叶子节点个数
60 70 97
```

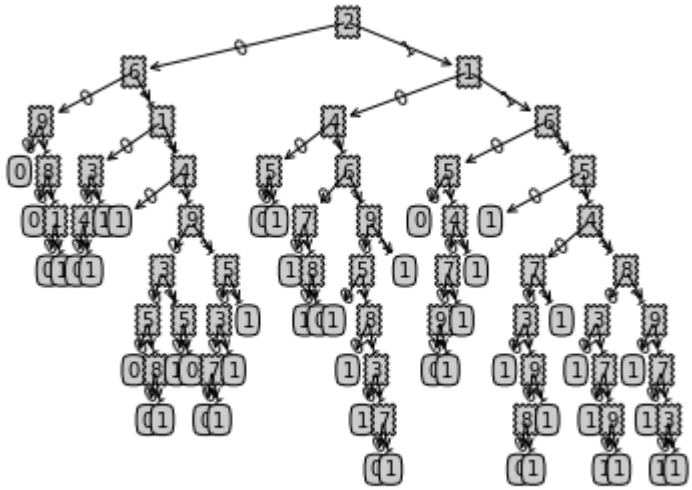
(2) 用均值离散

惩罚为 1

```
-------------ID3------------
```



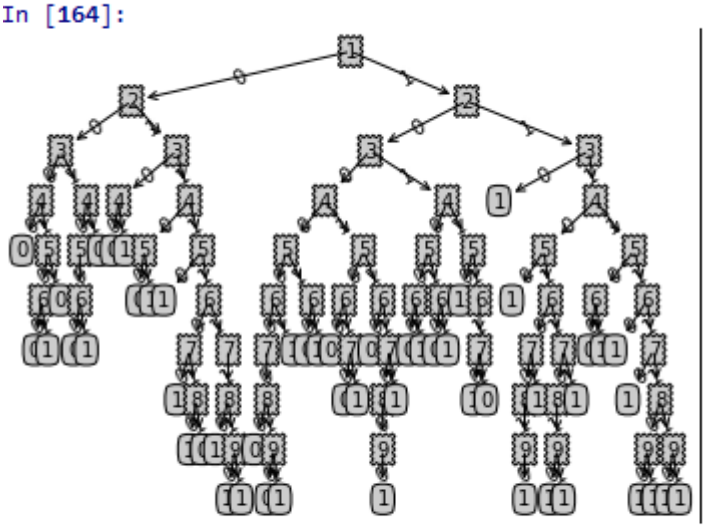
```
Accuracy =  0.89
Error =  0.52

-------------C4.5------------
```



```
Accuracy =  0.87
Error =  0.13
```

```
-------------CART------------
```

In [164]:



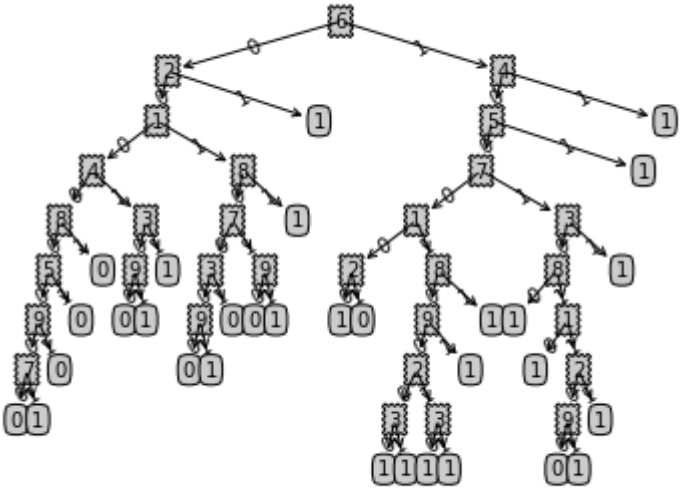
```
Accuracy =  0.89
Error =  0.63
三种决策树叶子节点个数
46 47 52
```

用了离散化之后，叶子节点明显减少

（3）用最大值与最小值的中值

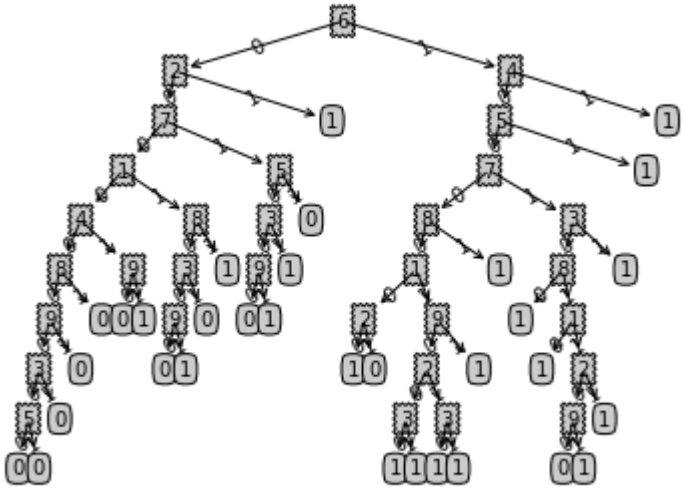
惩罚为 1

```
-------------ID3------------
```

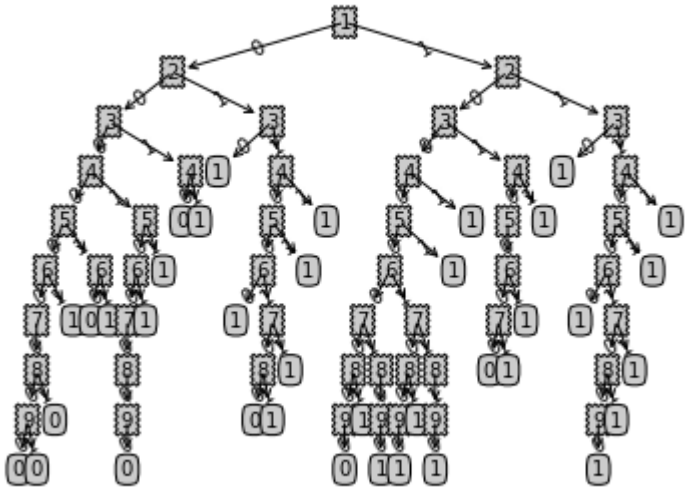


```
Accuracy =  0.83
Error =  0.48
```

```
------------C4.5------------
```



```
Accuracy =  0.87
Error =  0.13


-------------CART------------

In [163]:
```



```
Accuracy =  0.84
Error =  0.53
```

三种决策树叶子节点个数

32 32 37

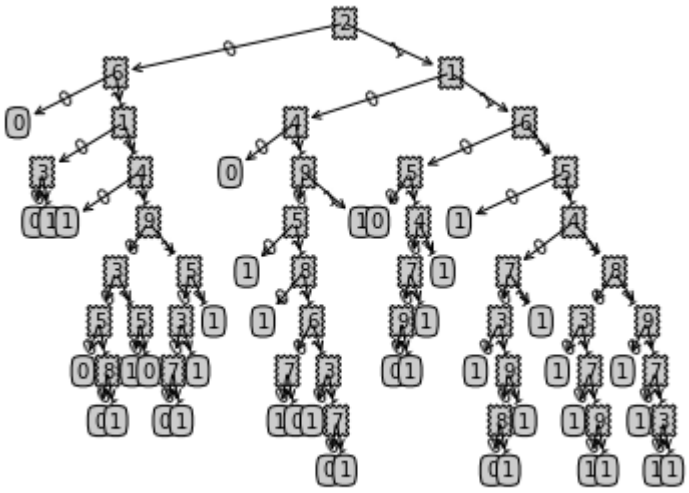
用最大值与最小值的中间值会得到更少叶子节点，但是它的准确率也会更低。

（4） 前剪枝

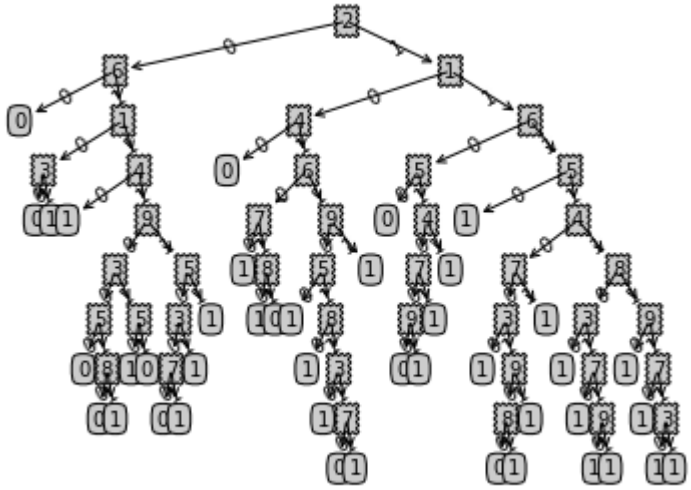
均值做划分点：

惩罚为 1

```
-------------ID3------------
```

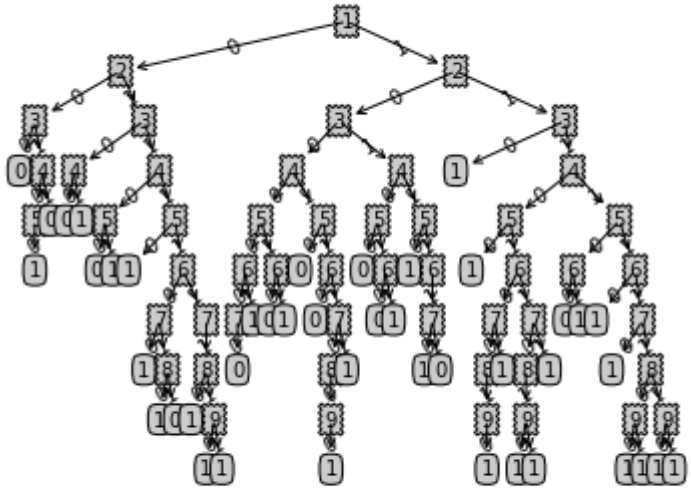


```
Accuracy =  0.86
Error =  0.55

-------------C4.5------------
```



```
Accuracy =  0.86
Error =  0.14
```

```
-------------CART------------
```

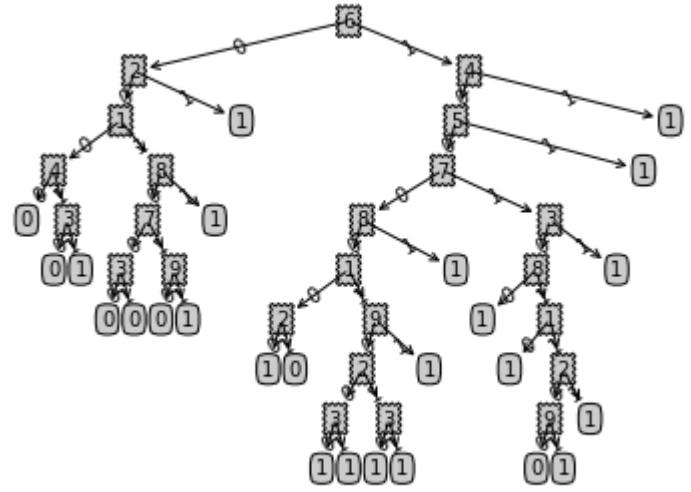


```
Accuracy =  0.87In [171]:
Error =  0.56
三种决策树叶子节点个数
41 42 43
```

叶子节点减少了几个，但是正确率不升反降

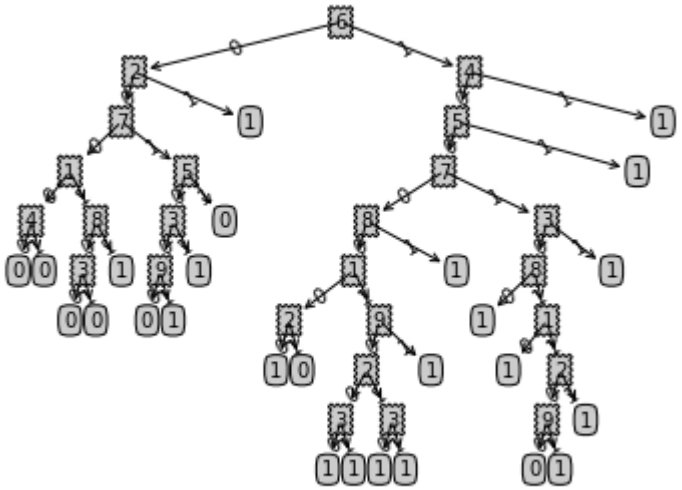
最大值与最小值中间值做划分点：
惩罚为 1：

```
-------------ID3------------
```

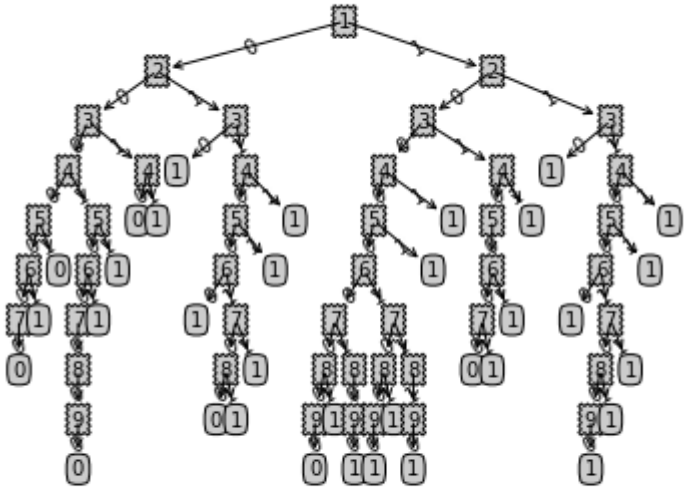


```
Accuracy =  0.85
Error =  0.4
```

```
-------------C4.5------------
```



```
Accuracy =  0.85
Error =  0.15

-------------CART------------
```



```
Accuracy =  0.82In [172]:
Error =  0.52
三种决策树叶子节点个数
25 26 34
```

叶子节点减少了几个，ID3 和 C4.5 正确率有上升。

从上面我们可以看到，即使是简单的剪枝，仍然是有助于减少节点个数，避免过拟合的。

其实剪枝时候的惩罚应该用验证集通过交叉验证找一个合适的，而不是随意放一个上去。

但是因为前面的逻辑回归实验做得过于充分，导致后面做决策树时间不是很足够。于是只能一切从简了。

个人觉得，如果能够有效避免过拟合，决策树的最终成绩不会差。

其实这次还有一些想法，没有时间实现。

譬如说把正则化用在逻辑回归, 或者是在决策树中构建更多类型的树而不是二叉树。离散化的方式改成三类四类的，但是没时间。

大 project 再做吧。

2. 评测指标展示即分析

	Accuracy	Precision	Recall	F-1
LR	0.97	0.9615	1.0	0.9702