实验三：聚类与分类实验报告

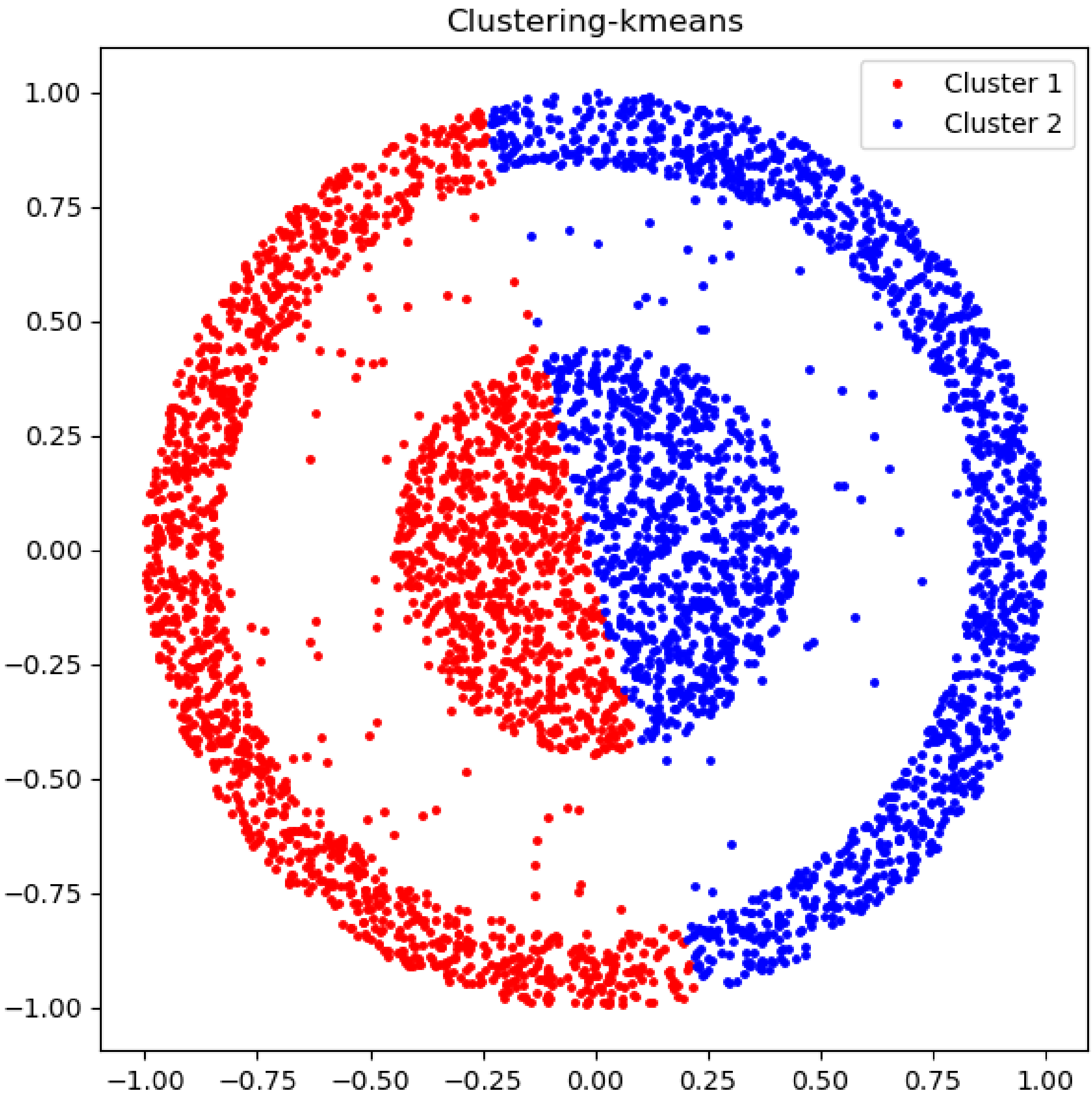
聚类

kmeans

核心代码

```
def kmeans(X, k):
    def dis(vector1, vector2):
        return np.sqrt(sum((vector2 - vector1) ** 2))  #定义欧几里得距离
    N, P = X.shape
    cent = np.zeros((k, P))   #k个中心
    for i in range(k):
        index = int(np.random.uniform(0, N))
        cent[i, :] = X[index, :]

    info = np.array(np.zeros((N, 2)))   #存某个点的中心和到中心的距离
    change_cent = True   #每一轮迭代是否change中心
    stop=0
    while change_cent:
        if(stop>100):     #设置迭代次数最多100次
            break
        stop=stop+1
        print(stop)
        change_cent = False
        for i in range(N):          #更新最小距离
            mindis = 999999999.0
            minidx = 0
            for j in range(k):
                distance = dis(cent[j, :], X[i, :])
                if distance < mindis:
                    mindis = distance
                    info[i, 1] = mindis
                    minidx = j
                if info[i, 0] != minidx:
                    change_cent = True
                    info[i, 0] = minidx
            for j in range(k):
                indexes = np.nonzero(info[:, 0] == j)
                points = X[indexes]
                cent[j, :] = np.mean(points, axis=0)
    return info[:,0]
```

运行结果

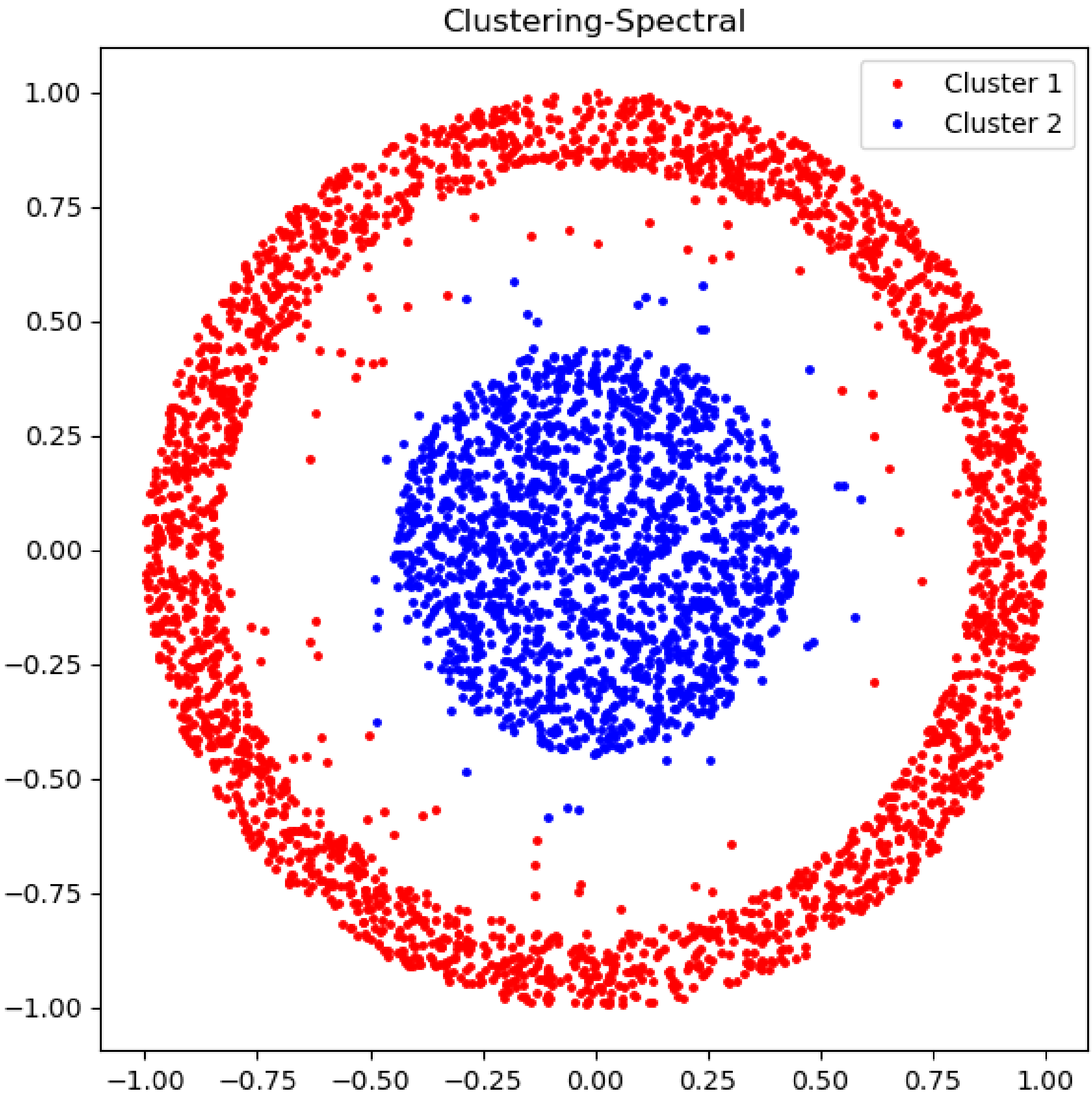


spectral

核心代码

```
def spectral(W, k):
    N = W.shape[0]
    D = np.diag(np.sum(W, axis=1))  #计算度数矩阵
    L = D - W
    sqrtD = np.power(np.linalg.matrix_power(D,-1),0.5)
    lap = np.dot(np.dot(sqrtD, L), sqrtD)      #计算拉普拉斯矩阵
    lam, E = np.linalg.eig(lap)        #计算特征矩阵和特征值
    dim = len(lam)
    dictEigval = dict(zip(lam, range(0, dim)))
    kEig = np.sort(lam)[0:k]        #取最小的特征值
    ix = [dictEigval[k] for k in kEig]
    X=E[:, ix]          #取最小的特征值对于特征向量
    X=X.astype(float)       #处理虚数不能做kmeans的问题
    idx = kmeans(X, k)
    return idx
```

运行结果



两种方法的对比

对于测试数据，显然kmeans无法正确地聚类，因为kmeans根据点到聚类中心的距离决定聚类，其聚类结果必定是凸的。事实上，如果不设置stop，kmeans对于一些特殊的数据可能会无限迭代下去。而spectral方法是根据类内点与点之间的距离（或者说距离所代表的相似度）来聚类的，对于测试数据这样的圆环图，能够正确聚类。

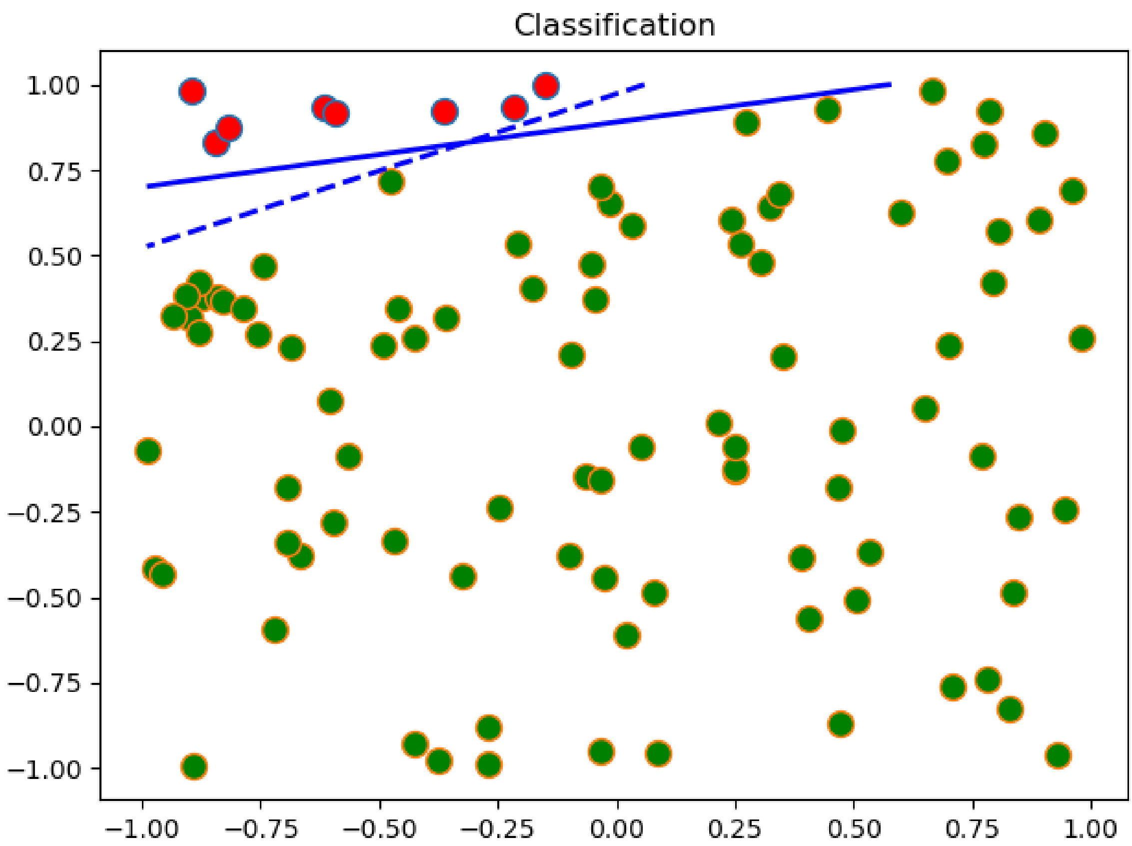
分类

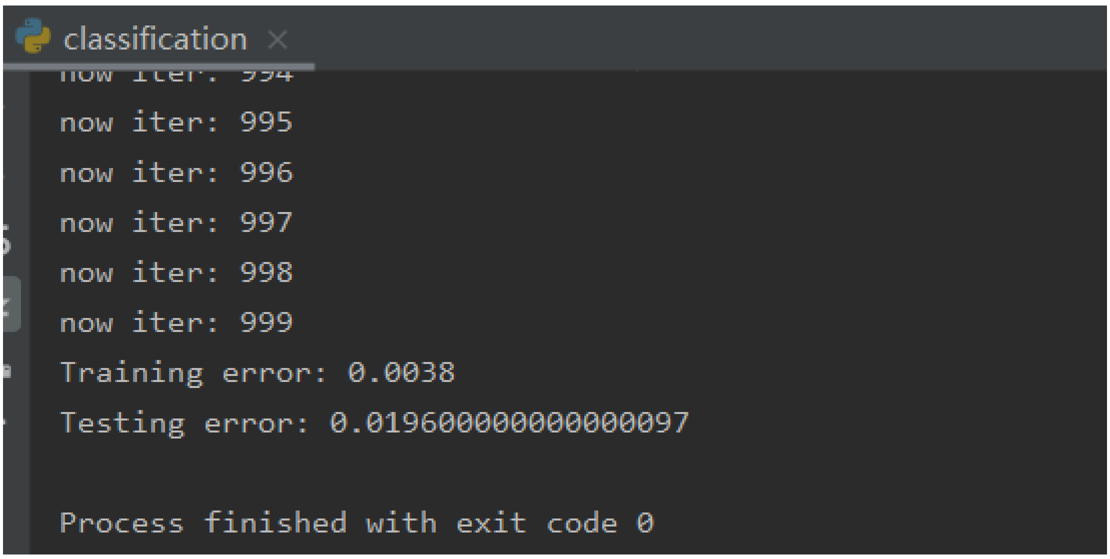
LR

核心代码

```
def LR(XX,yy):
    P, N = XX.shape
    X=numpy.zeros((N,P+1),dtype='float64')
    for i in range(N):
        X[i][0]=1
        for j in range(P):
            X[i][j+1]=XX[j][i]
    y=yy.transpose()         #将X，y处理成适应LR计算式的shape
    return 
numpy.dot(numpy.linalg.inv(numpy.dot(numpy.transpose(X),X)),numpy.dot(numpy.tran
spose(X),y))
```

运行结果



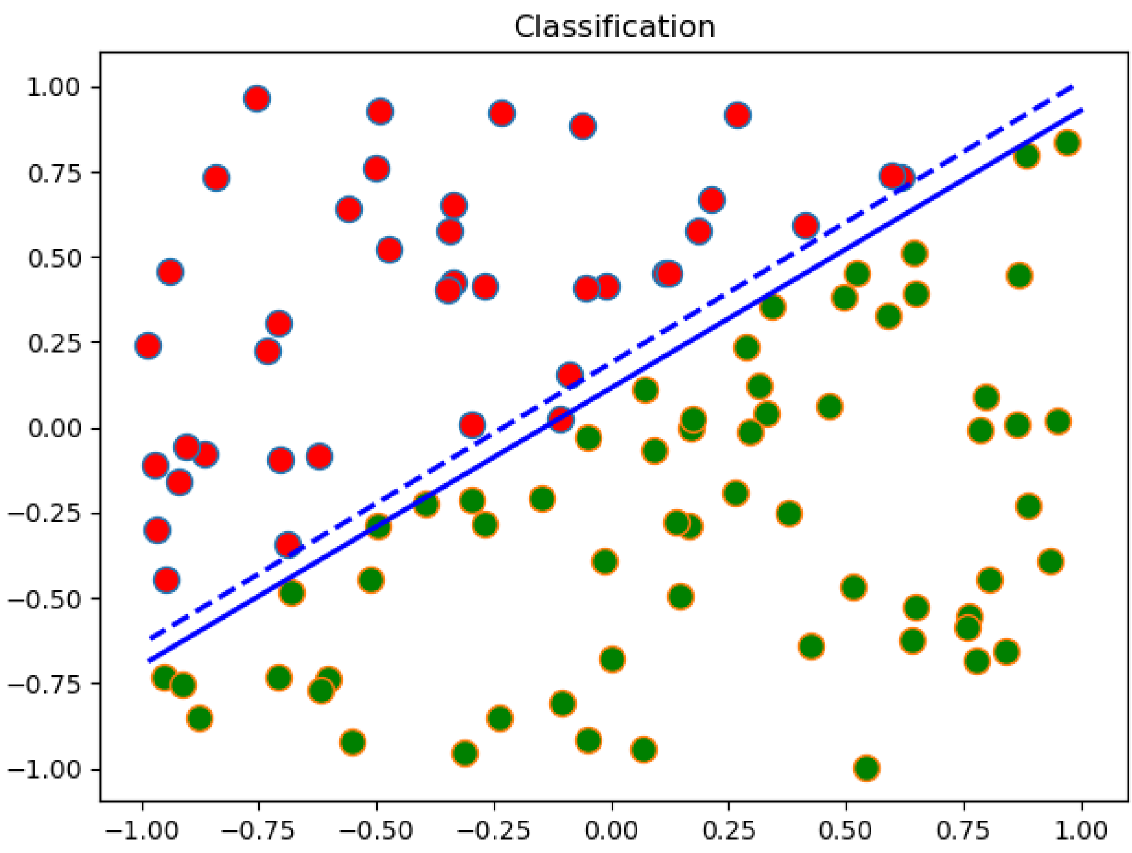


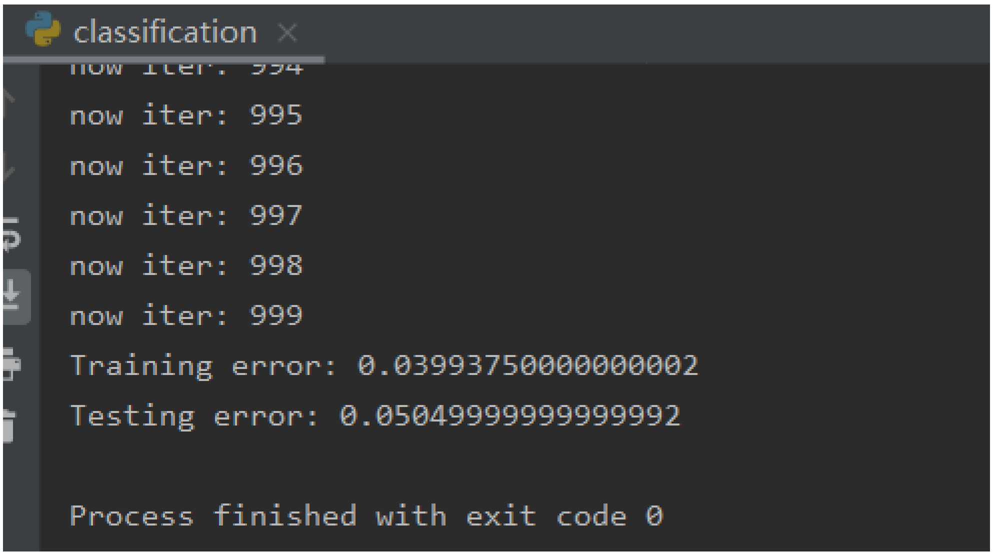
感知机（bonus）

核心代码

```
def func(X, y):
    n_epoch=40        #定义迭代次数，通过尝试发现100花费时间过多，20准确度不足，50左右是合适的位置
    P, N = X.shape
    w = numpy.zeros((P + 1, 1), dtype='float64')
    epoch=0
    rho=0.13          #定义rho=0.13，尝试发现0.5准确度不足，0.05花费时间过多，0.1左右是合适的位置
    while(epoch<n_epoch):
        for i in range(N):
            S=w[0]
            for j in range(P):
                S=S+w[j+1]*X[j][i]
            if S>0:
                y_predict=1
            else:
                y_predict=-1          #计算预测的y
            for j in range(P):
                w[j+1]=w[j+1]+rho*X[j][i]*(y[0][i]-y_predict)
            w[0]=w[0]+rho*(y[0][i]-y_predict)           #根据预测更新w
        epoch=epoch+1
    return w
```

运行结果



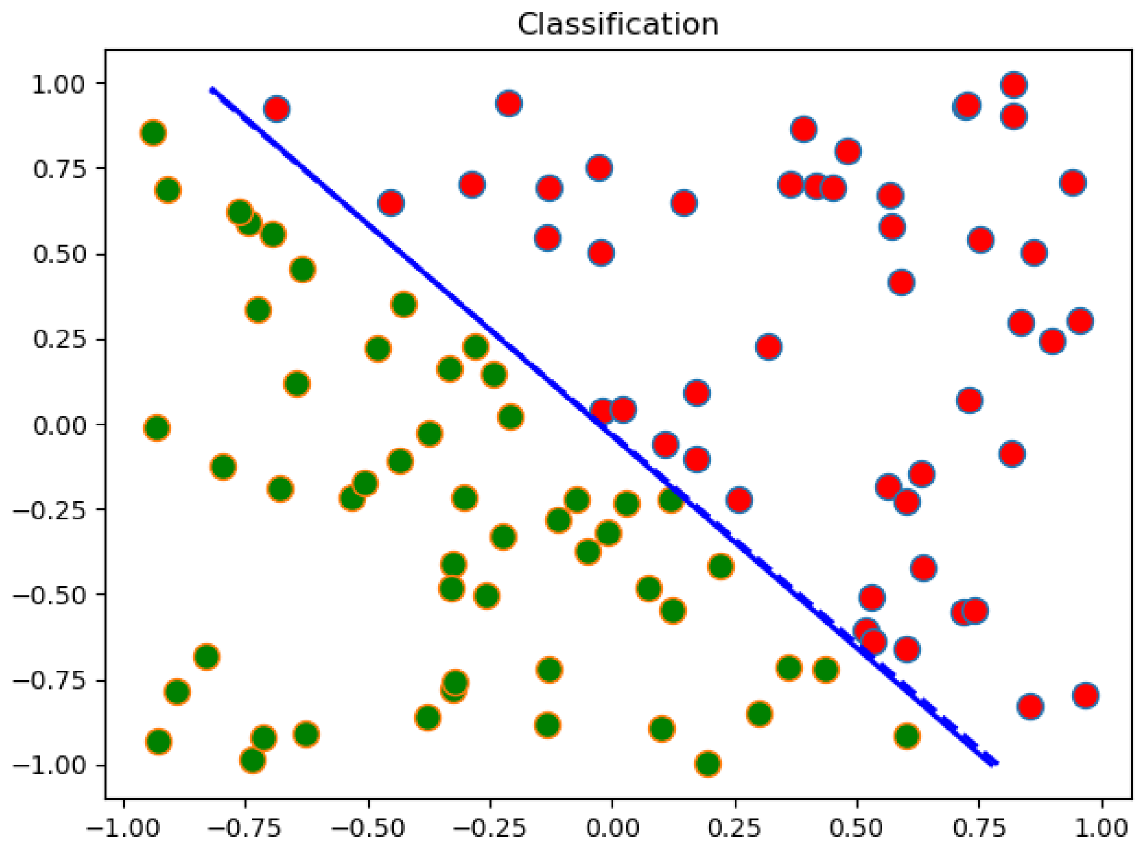


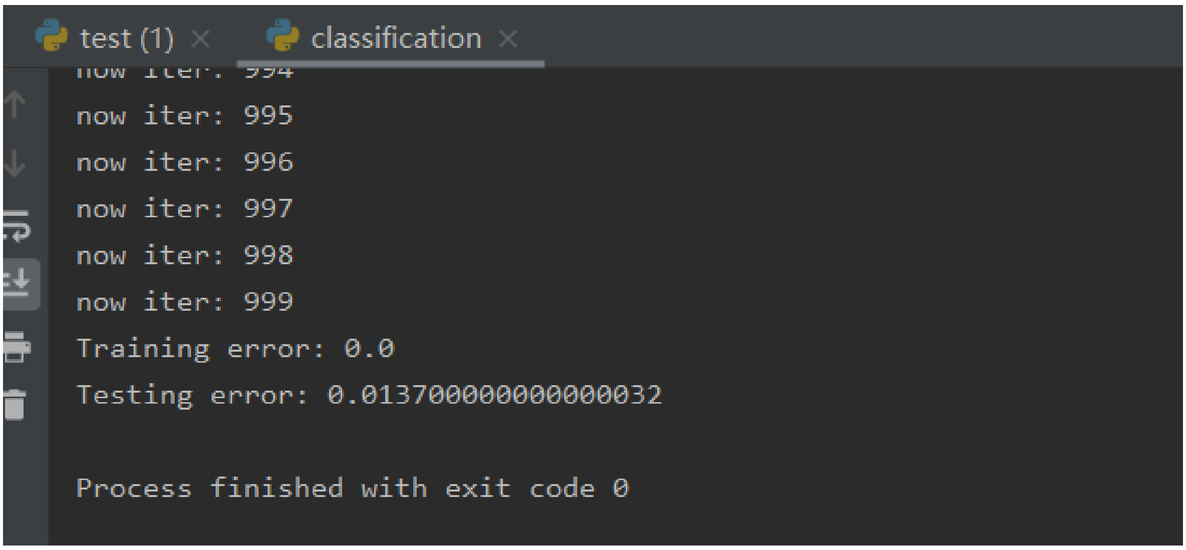
SVM（bonus)

核心代码

```
from scipy.optimize import minimize
def SVM(XX,yy):
    P, N = XX.shape
    X=numpy.zeros((N,P+1),dtype='float64')
    for i in range(N):
        X[i][0]=1
        for j in range(P):
            X[i][j+1]=XX[j][i]
    w = numpy.zeros((P + 1, 1), dtype='float64')
    y = numpy.zeros((N,))
    for i in range(N):
        y[i] = yy[0][i]         #调整X，y的shape为minimize函数适应的shape

    def fun(w, X, y):
        object_value = 0.5 * numpy.sum(w ** 2)      #最小化对象：1/2*(w**2)
        return object_value

    def constraint(w, X, y):
        return y * numpy.dot(X,w) - 1       #约束：对于每一个yi，yi*Xi*w >= 1

    solver = minimize(fun, w, args=(X, y),
                      constraints=({'type': 'ineq', 'args': (X, y),
                                    'fun': lambda w, X, y: constraint(w, X, y)}))
    return solver.x
```

运行结果





三种方法的对比

对于LR，可以看到正确率较高，但是直线拟合并不好，预测的直线完全贴近分类边缘的两个点，在很多情况下（训练数据和真实数据有一定差距的情况下）会导致不好的分类结果。

对于感知机，可以看到正确率不算高，直线拟合结果也不算好。将迭代次数增加，rho减小可以一定程度上改善这个问题。

对于SVM，可以看到正确率相当高，直线拟合结果也非常好。

除此之外，虽然没有在运行结果中显示运行时间，事实上速度是LR>感知机>SVM。特别是LR和SVM，有明显的速度差异。1000个iter对于SVM来说一秒左右就运行完毕，而LR运行了相当长的时间。