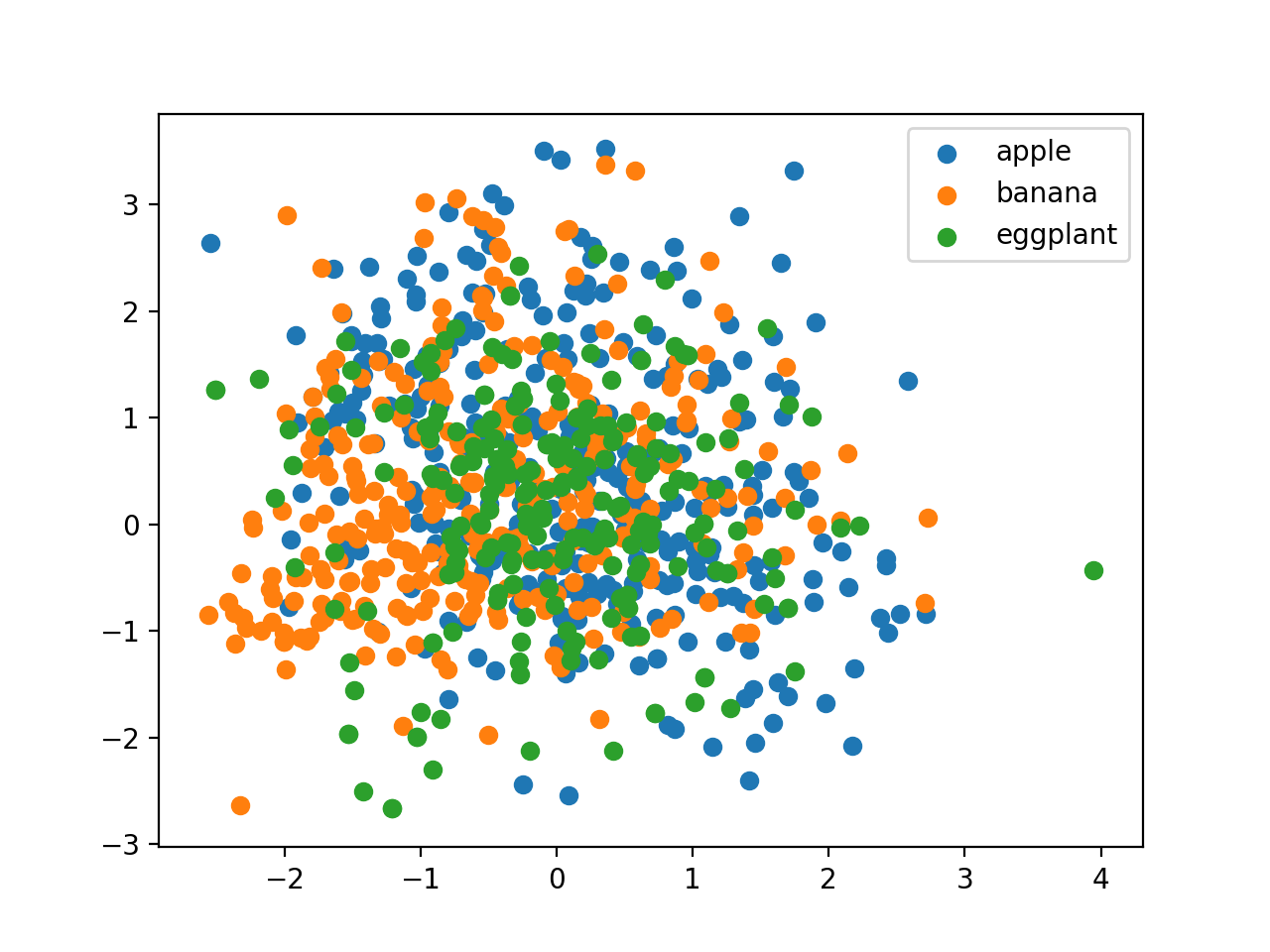
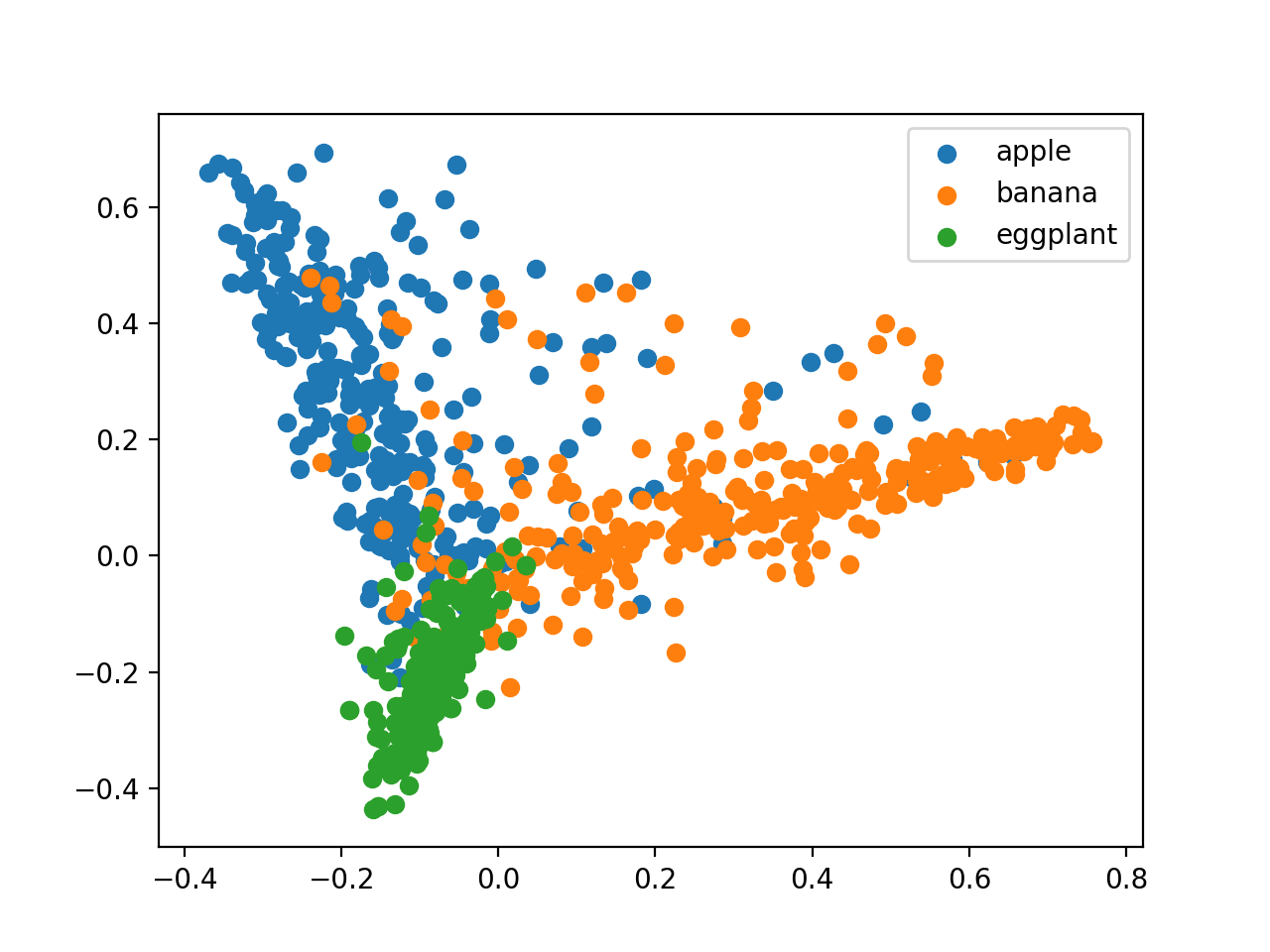
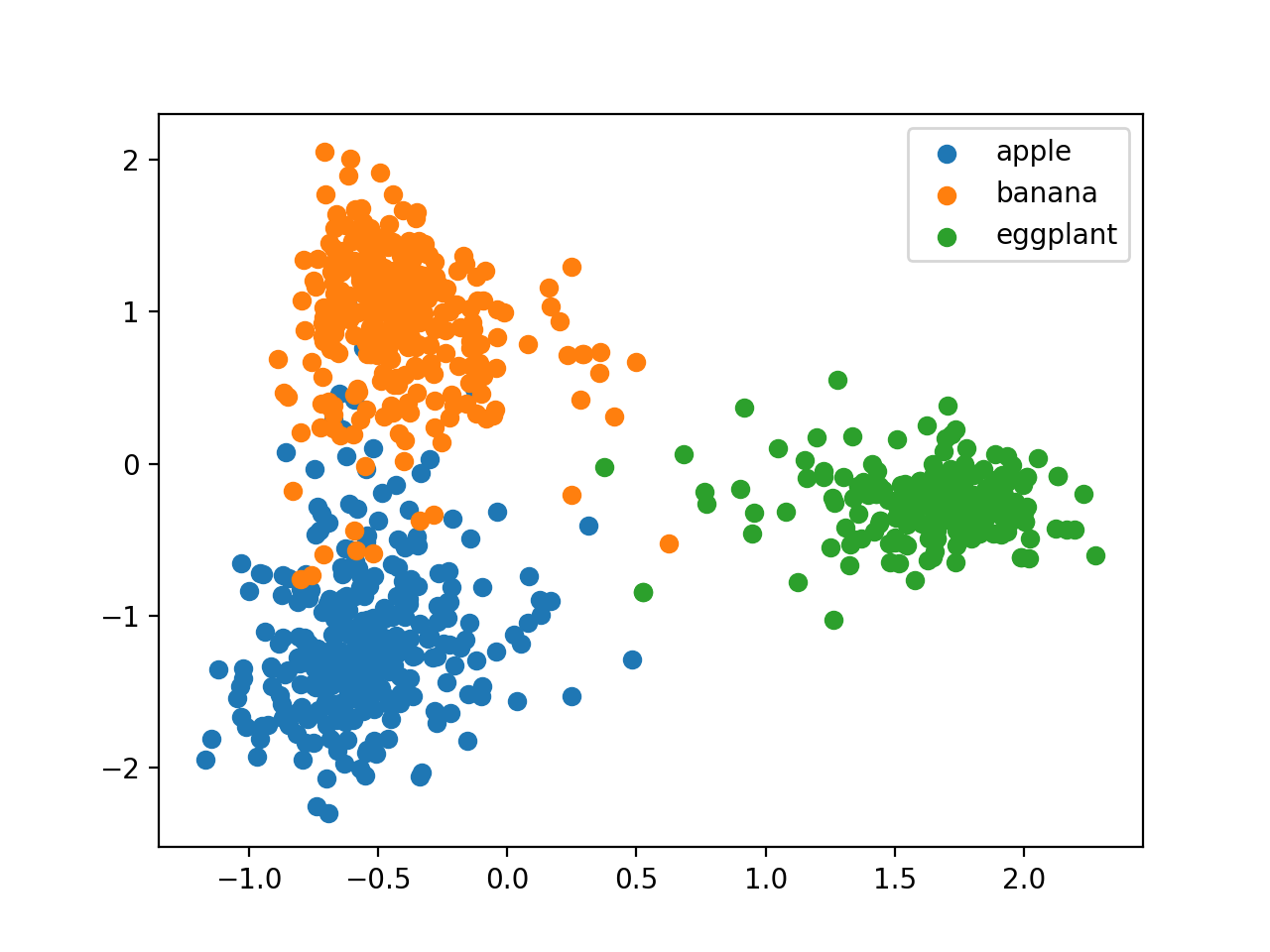
**3a**

****

**3b**

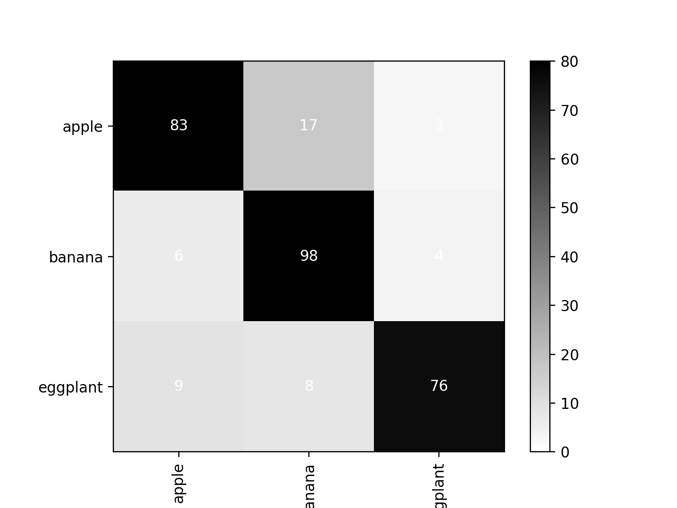
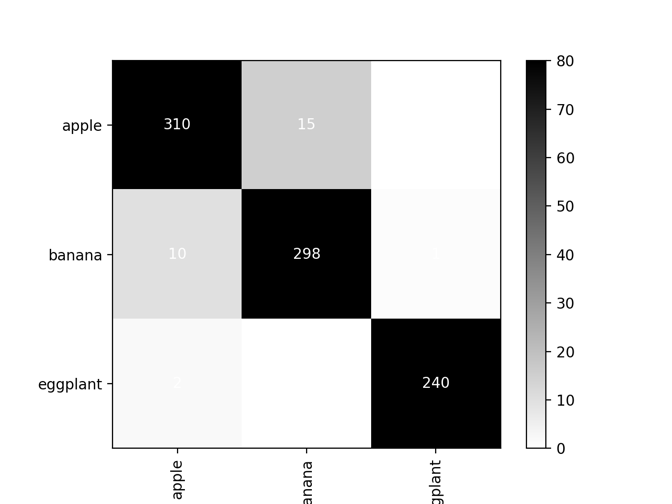
****

**3c**

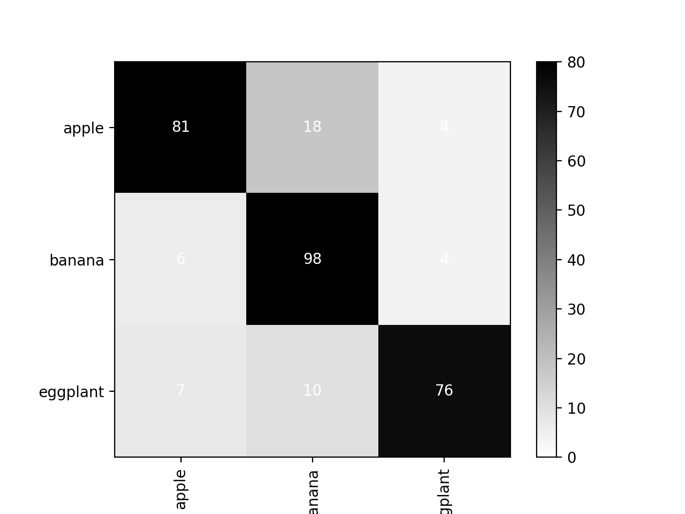
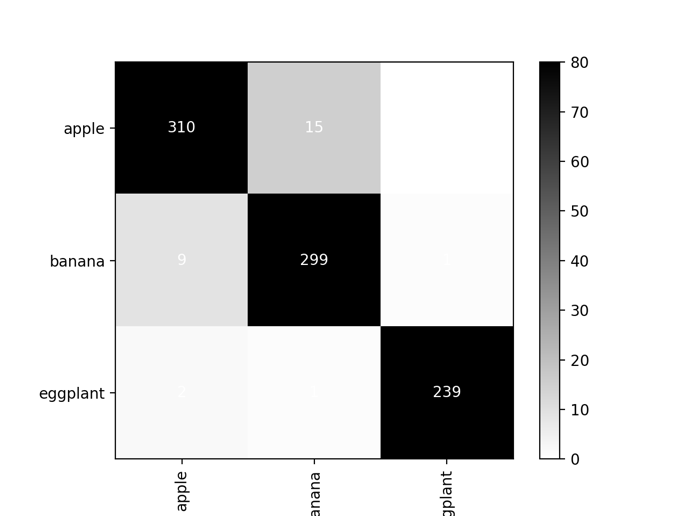
****

CCA is the best, random projection is the worst. Because essentially cca uses the most information(both features and labels), pca only uses features, random projections uses nothing from the data.

**3d**

****

**3e**

****

--------code------------

class LDA\_Model():

def \_\_init\_\_(self,class\_labels):

###SCALE AN IDENTITY MATRIX BY THIS TERM AND ADD TO COMPUTED COVARIANCE MATRIX TO PREVENT IT BEING SINGULAR ###

self.reg\_cov = 0.001

self.NUM\_CLASSES = len(class\_labels)

def train\_model(self,X,Y):

''''

FILL IN CODE TO TRAIN MODEL

MAKE SURE TO ADD HYPERPARAMTER TO MODEL

'''

X = np.array(X)

Y = np.array(Y)

self.means = np.zeros((self.NUM\_CLASSES, X.shape[1]))

X\_demean = np.zeros\_like(X)

for i in range(self.NUM\_CLASSES):

self.means[i, :] = np.mean(X[Y==i], axis = 0)

X\_demean[Y==i] = X[Y==i] - self.means[i, :]

self.covariance = 1/Y.shape[0] \* X\_demean.T @ X\_demean + self.reg\_cov\*np.eye(X.shape[1])

def eval(self,x):

''''

Fill in code to evaluate model and return a prediction

Prediction should be an integer specifying a class

'''

pred\_array = np.zeros(self.NUM\_CLASSES)

x = np.array(x)

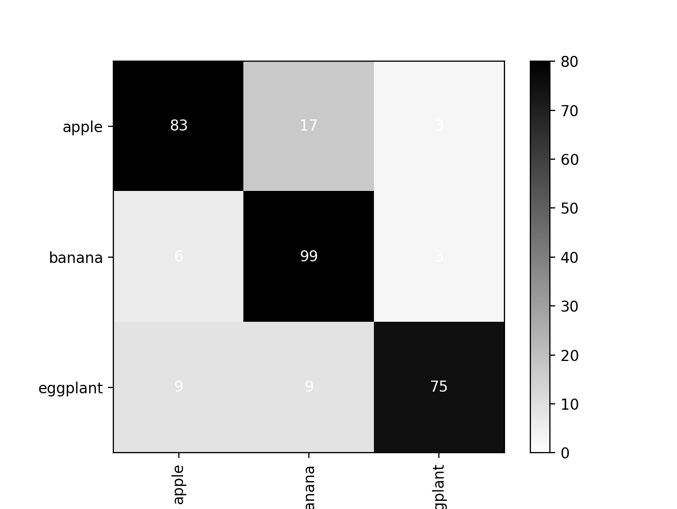
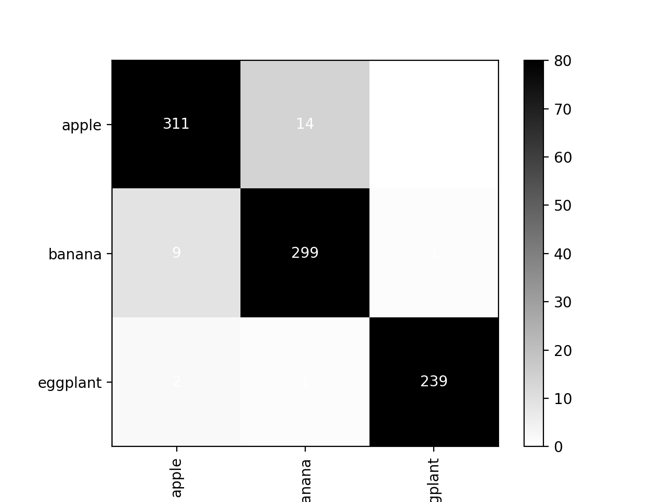
for i in range(self.NUM\_CLASSES):

x\_demean = (x-self.means[i, :]).reshape(x.shape[0], -1)

pred\_array[i] = (x\_demean.T @ inv(self.covariance) @ x\_demean)[0, 0]

return np.argmin(pred\_array)

**3f**

****

--------code------------

class LDA\_Model():

def \_\_init\_\_(self,class\_labels):

###SCALE AN IDENTITY MATRIX BY THIS TERM AND ADD TO COMPUTED COVARIANCE MATRIX TO PREVENT IT BEING SINGULAR ###

self.reg\_cov = 0.001

self.NUM\_CLASSES = len(class\_labels)

def train\_model(self,X,Y):

''''

FILL IN CODE TO TRAIN MODEL

MAKE SURE TO ADD HYPERPARAMTER TO MODEL

'''

X = np.array(X)

Y = np.array(Y)

self.means = np.zeros((self.NUM\_CLASSES, X.shape[1]))

X\_demean = np.zeros\_like(X)

for i in range(self.NUM\_CLASSES):

self.means[i, :] = np.mean(X[Y==i], axis = 0)

X\_demean[Y==i] = X[Y==i] - self.means[i, :]

self.covariance = 1/Y.shape[0] \* X\_demean.T @ X\_demean + self.reg\_cov\*np.eye(X.shape[1])

def eval(self,x):

''''

Fill in code to evaluate model and return a prediction

Prediction should be an integer specifying a class

'''

pred\_array = np.zeros(self.NUM\_CLASSES)

x = np.array(x)

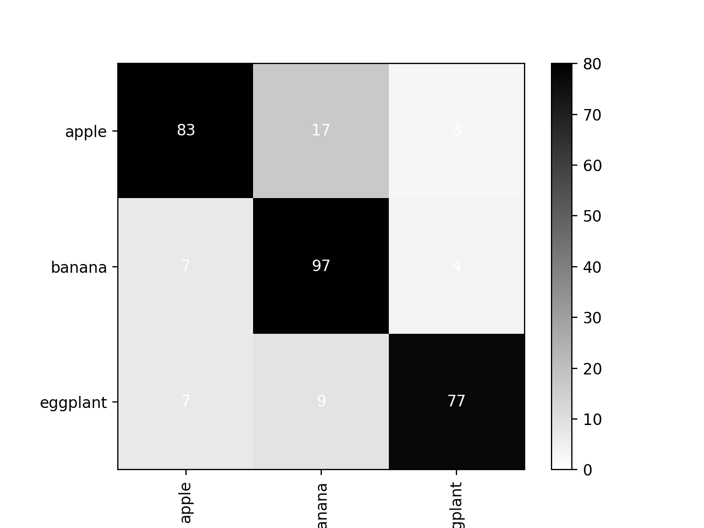
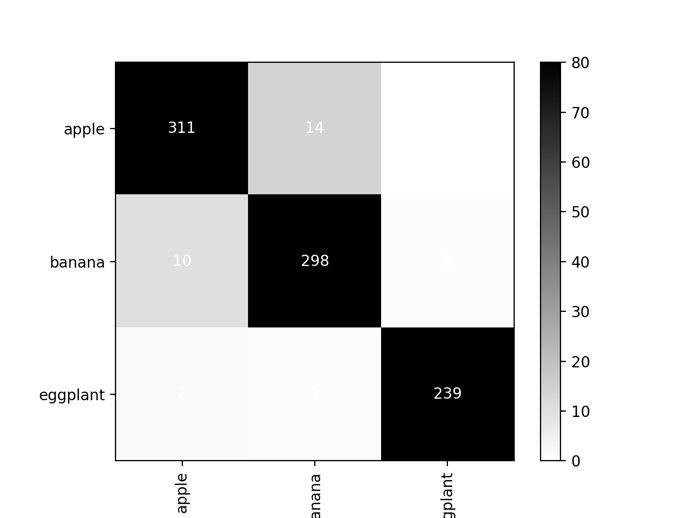
for i in range(self.NUM\_CLASSES):

x\_demean = (x-self.means[i, :]).reshape(x.shape[0], -1)

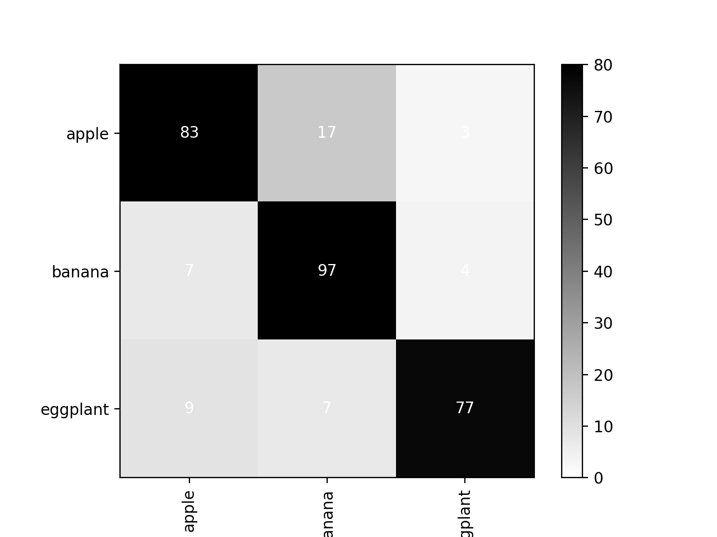
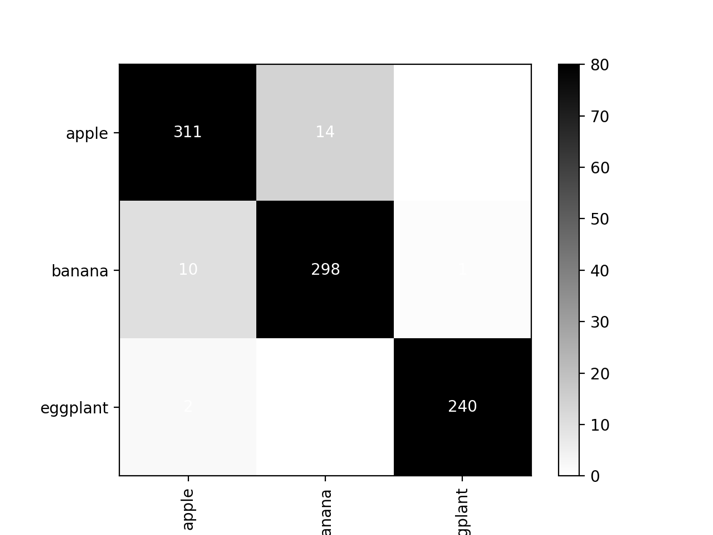
pred\_array[i] = (x\_demean.T @ inv(self.covariance) @ x\_demean)[0, 0]

return np.argmin(pred\_array)

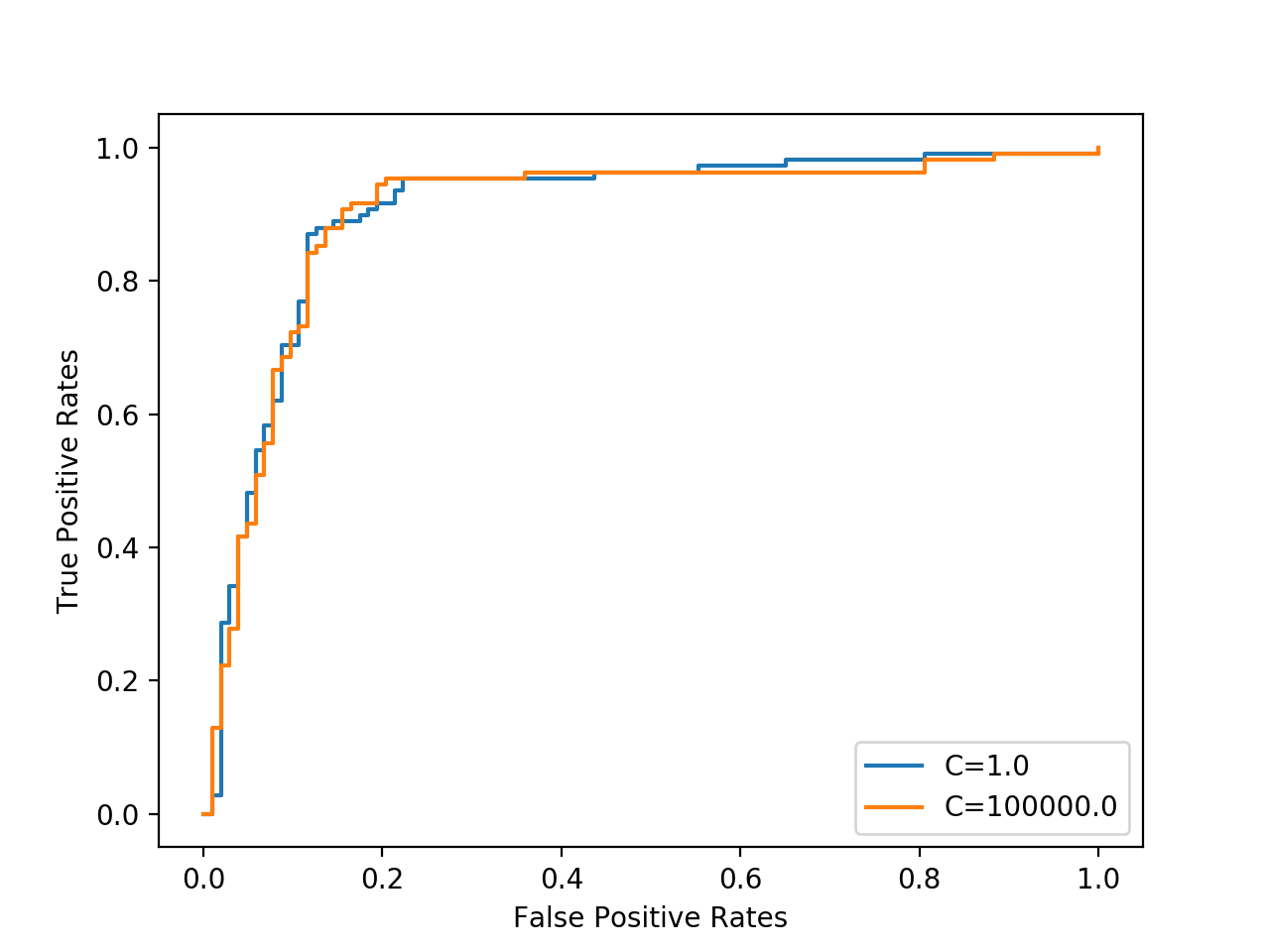
**3g**

****

**3h**

****

**3i**

****

The better one should be the curve that curved most towards (0, 1), which means the area under curve (AUC) should be large for the better one.

But in this case, it seems the AUC for the two curves are similar, so it is hard to decide which one is better.

---------code----------

def ROC(scores, labels):

thresholds = sorted(np.unique(scores))

thresholds = [-float("Inf")] + thresholds + [float("Inf")]

tps = []

fps = []

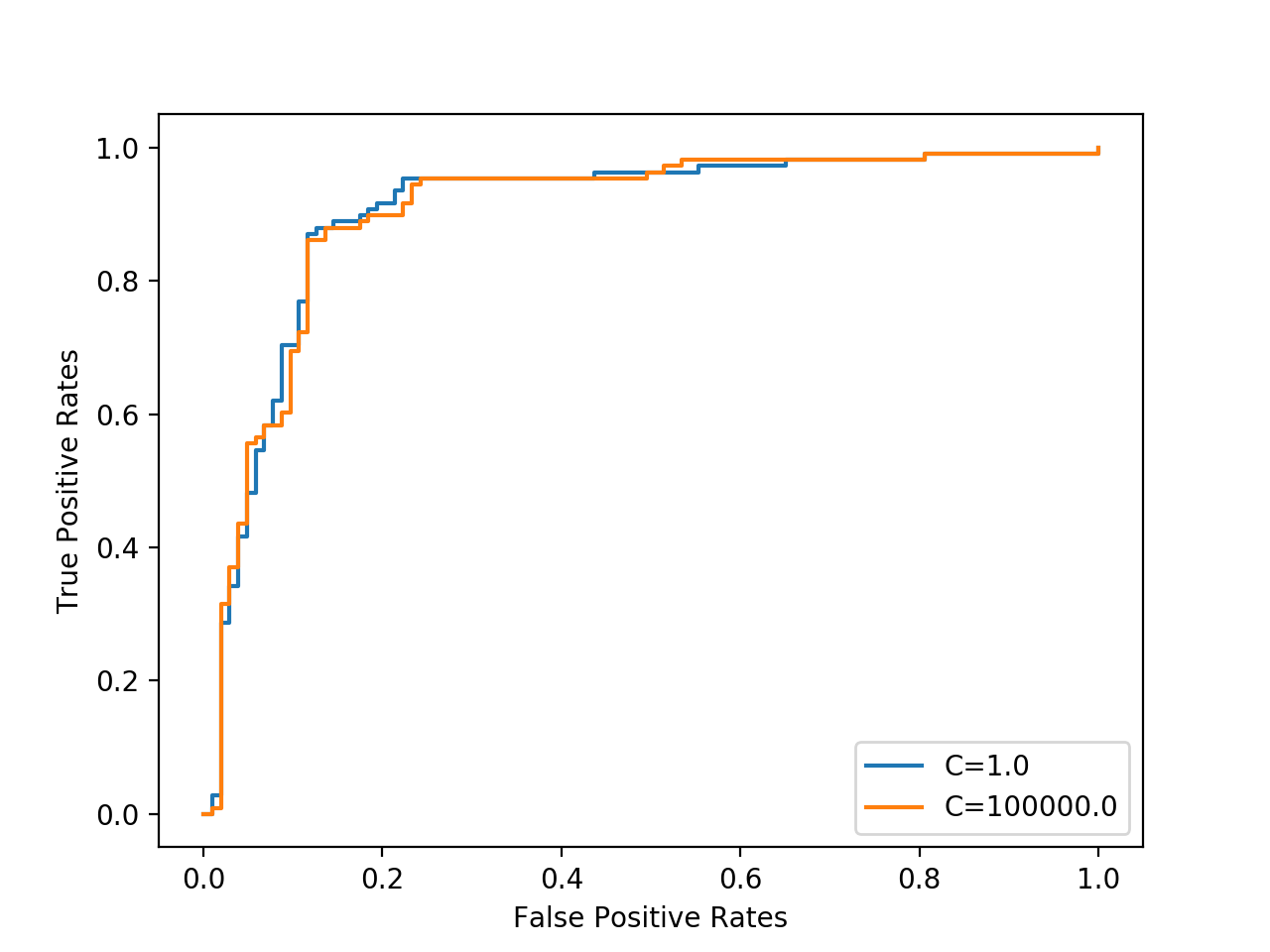
for thres in thresholds:

tpr, fpr = compute\_tp\_fp(thres, scores, labels)

tps.append(tpr)

fps.append(fpr)

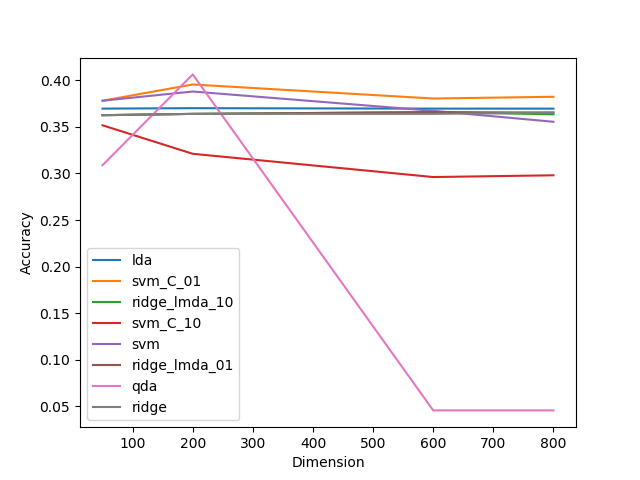
return tps, fps

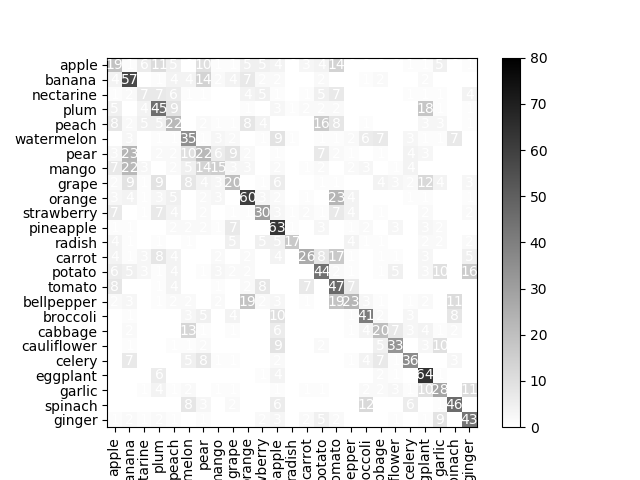
**3j **

The ROC curve will not change, the slight change here are due to randomness.

**3k**

The best model seems to be qda with dimension k=200, but the accuracy is not good, only 40% pictures are classified to the right class.

****

****