HullRobert-HW-1.md

Robert (Quinn) Hull

HW 1, ECE 523

2/1/2021

1 Probability and Discriminant Classifiers

Part 1: Maximum and Posterior v Probability of Chance

PART I: Maximum Posterior vs Probability of Chance Show/explain that $P(\omega_{\max}|\mathbf{x}) \geq \frac{1}{c}$ when we are using the Bayes decision rule, where c is the number of classes. Derive an expression for $p(\text{err})$. Let $\omega_{\max}$ be the state of nature for which $P(\omega_{\max}|\mathbf{x}) \geq P(\omega_i|\mathbf{x})$ for $i =, 1 \ldots, c$. Show that $p(\text{err}) \leq (c-1)/c$ when we use the Bayes rule to make a decision. Hint, use the results from the previous questions.

Answer:

(1)

FIND $P(err)$ FACT: $P(w_{max}|x) \geq \frac{1}{c}$

a) The FACT $P(w_{max}|x) \geq \frac{1}{c}$ states in plan english that for a set of classes $w_1, w_2, w_3, \ldots, w_c$ where x is a matrix of variables representing a sample and c is the number of classes (w):

- the maximum likelihood, of having class w_j given a set of variables x is at least as large as $1/c$.

- This is most intuitive when we think of chance. For example if you have a fair coin the probability of heads v. tails is $1/c$, $c = 2$, $1/2$. Even if the coin were unfair, the maximum probability of the unfair side would have to be at least $1/2$, say $2/3$, or $3/4$.

b) Prove $p(err) \leq (c-1)/c$

recall: $p(x) = \sum_{y \in y} P(X, Y = y)$ (marginal distribution)

substituting "error" for X and x for Y, and using integral notation

$$p(error) = \int_x P(error, x)\, dx$$

using the product rule:

$$p(error) = \int_x p(error|x)\, p(x) \cdot dx$$

$$= \int_x (1 - p(w_{max}|x))\, p(x)\, dx$$

$$= \int_x p(x)\, dx - \int_x p(w_{max}|x)\, dx$$

- given $p(error|x) + p(w_{max}|x) = 1$
- using properties of integrals
- where $p(w_{max}|x) \geq \frac{1}{c}$
- where $p(x) = 1$

$$\boxed{p(err) \leq 1 - \frac{1}{c}}$$ ← proof Complete ✓

Part 2: Bayes Decision Rule Classifier

PART II: Bayes Decision Rule Classifier Let the elements of a vector $\mathbf{x} = [x_1, \ldots, x_d]^\top$ be binary valued. Let $P(\omega_j)$ be the prior probability of the class ω_j $(j \in [c])$, and let

$$p_{ij} = P(x_i = 1|\omega_j)$$

with all elements in $\mathbf{x}$ being independent. If $P(\omega_1) = P(\omega_2) = \frac{1}{2}$, and $p_{i1} = p > \frac{1}{2}$ and $p_{i2} = 1 - p$, show that the minimum error decision rule is

$$\text{Choose } \omega_1 \text{ if } \sum_{i=1}^{d} x_i > \frac{d}{2}$$

Hint: Think back to ECE503 and types of random variables then start out with

$$\text{Choose } \omega_1 \text{ if } P(\omega_1)P(\mathbf{x}|\omega_1) > P(\omega_2)P(\mathbf{x}|\omega_2)$$

Answer

II

$P(w_1) = P(w_2) = 1/2$

$p = P_{i1} (> \frac{1}{2})$ The probability that $x_i = 1$ given class $= 1$

$1-p = P_{i2} (< \frac{1}{2})$ The probability that $x_i = 1$ given class $= 2$

Show that Decision rule is:

Choose w_1 if $\sum_{i=1}^{d} x_i > \frac{d}{2}$, $\Leftrightarrow$ choose w_1 if more than half of elements in X are 1

$$p(w_1) P(x|w_1) > p(w_2) P(X|w_2)$$

$$p(x|w_1) > p(x|w_2)$$

recall x is independent

$$\prod_{i=1}^{d} p(x_i|w_1) > \prod_{i=1}^{d} p(x_i|w_2)$$

drawing from the principles of Bernoulli, because this is a binary variable

$$p^K (1-p)^{(d-K)} > (1-p)^K p^{(d-K)}$$

where $K = \#$ success

$$p^K (1-p)^d (1-p)^{-K} > (1-p)^K p^d p^{-K}$$

$$(1-p)^d p^{-d} > (1-p)^K p^{-K} p^{-K} (1-p)^K$$

$$(1-p)^d p^{-d} > (1-p)^{2K} p^{-2K}$$

$$\left(\frac{1-p}{p}\right)^d > \left(\frac{1-p}{p}\right)^{2K} \qquad f = \left(\frac{1-p}{p}\right)$$

$$f^d > f^{2K} \Rightarrow f^{(d-2K)} > 1$$

$$\left(\frac{1-p}{p}\right)^{(d-2K)} > 1$$

recall $p > \frac{1}{2}$, $p-1 < \frac{1}{2}$ so $\left(\frac{1-p}{p}\right) < 1$

Proof by trial & error
- if condition satisfied, choose w_1
- if condition not satisfied, choose w_2

~~Proof by trial and error~~

$$\left(\frac{1-p}{p}\right)^{d-2K} > 1 \quad , \text{ where } \left[\frac{1-p}{p} < 1\right] = f$$

① Condition $K = 0$

$$\left(f<1\right)^{d} > 1$$

Result: because exponent is a positive #, then left hand side does (L.H.S) < Right hand side (R.H.S). Choose W_2

This condition is if there 0 elements $= 1$

② Condition $K = d$

$$\left(f<1\right)^{(-d)} > 1$$

Result: exponent is negative so L.H.S. > R.H.S. Choose W_1

This condition if all elements $= 1$

③ Condition $K = \frac{d}{2}$

$$\left(f<1\right)^{0} > 1$$

Result: exponent is 0 so L.H.S. < R.H.S Choose W_2

This condition if half of elements $= 1$

from condition 3 we know if $K = \frac{d}{2} + 1$, then we must choose W_1. In plain words, if more than half of elements in $x = 1$, then choose W_1.

This is identical to:

Choose W_1 if $\sum_{i=1}^{d} x_i > \frac{d}{2}$

Part 3: The Ditzler Household Growing Up

PART III: The Ditzler Household Growing Up My parents have two kids now grown into adults. Obviously there is me, Greg. I was born on a Wednesday. What is the probability that I have a brother? You can assume that $\mathbb{P}(\text{boy}) = \mathbb{P}(\text{girl}) = \frac{1}{2}$.

Answer

B (rows) / B (columns):

	M	T	W	Th	F	S	Su
M			x				
T			x				
W	x	x	x	x	x	x	x
Th			x				
F			x				
S			x				
Su			x				

13/49

B (rows) / 6 (columns):

	M	T	W	Th	F	S	Su
M							
T							
W	x	x	x	x	x	x	x
Th							
F							
S							
Su							

7/49

6 (rows) / B (columns):

	M	T	W	Th	F	S	Su
M			x				
T			x				
W			x				
Th			x				
F			x				
Sa			x				
Su			x				

7/49

6 (rows) / 6 (columns):

	M	T	W	Th	F	S	Su
M							
T							
W							
Th							
F							
S							
Su							

0/49

$P(B) = 1/2 \qquad P(6) = 1/2$

$P(BB) = 1/4 \qquad P(B6) = 1/4$

$P(6B) = 1/4 \qquad P(66) = 1/4$

We want: $P(BB|B) = \dfrac{P(B|BB) \cdot P(BB)}{P(B)}$

So $P(BB|B) = \dfrac{\Sigma BB}{\Sigma \text{all}} = \dfrac{13}{13+14} = \dfrac{13}{27} = 48\%$

Part 4: Bayes Classifier

PART IV: Bayes classifier Let consider a Bayes classifier with $p(\mathbf{x}|\omega_i)$ distributed as a multivariate Gaussian with mean μ_i and covariance $\Sigma_i = \sigma^2 I$ (note they all share the same covariance). We choose the class that has the largest

$$g_i(\mathbf{x}) = \log(p(\mathbf{x}|\omega_i)P(\omega_i)) \propto \mathbf{w}_i^\mathsf{T}\mathbf{x} + w_{0i}$$

Find $\mathbf{w}_i$ and w_{0i}. Fact:

$$p(\mathbf{x}|\omega_i) = \frac{1}{(2\pi)^{\frac{d}{2}}|\Sigma_i|^{\frac{1}{2}}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\mu_i)^\mathsf{T}\Sigma_i^{-1}(\mathbf{x}-\mu_i)\right\}$$

Answer?

IV Bayes Classifier

prove that

$$\therefore g_i(x) = \underbrace{\log(p(x|w_i)Pw_i))}\ \propto W_i^T + W_{0i}$$

$$g_i(x) = \log(p(x|w_i)Pw_i)$$

$$g_i(x) = \log(p(x|w_i)) + \log(Pw_i)$$

We know $p(x|w_i) = \frac{1}{2\pi^{d/2}|\Sigma_i|^{\frac{1}{2}}} \exp\left[-\frac{1}{2}(x-u_i)^T\Sigma_i^{-1}(x-u_i)\right]$

$$g_i(x) = \log\left(\frac{|\Sigma_i|^{-1/2}}{(2\pi)^{d/2}} \cdot \exp\left[-\frac{1}{2}(x-u_i)^T\Sigma_i^{-1}(x-u_i)\right]\right) + \log(P(w_i))$$

Now break up log

$$= \underbrace{\cancel{-\frac{1}{2}\log|\Sigma_i|}}_{1} - \underbrace{\cancel{\frac{d}{2}\log(2\pi)}}_{1} + \underbrace{\frac{1}{2}(x-u_i)^T\Sigma_i^{-1}(x-u_i)}_{2} + \log(P(w_i))$$

Now - simplify
1. remove constants
2. $\Sigma_i = \sigma^2 I$ where σ^2 are constant w/in and between classes, so sub σ^2

$$= \frac{-1}{2\sigma^2}(x-u_i)^T(x-u_i) + \log(P(w_i))$$

Further simplify that:

$$g_i x \quad \frac{-1}{2\sigma^2}\left(\cancel{x^T x} - 2u_i^T x + u_i^T u_i\right) + \log(P(w_i))$$

Our task is to maximize $g_i x$ for a given x, so
1. $x^T x$ is constant for all class w_j. As such it reduces to (is subtracted)
2. gets distribut

is the same for all classes

$$g_i x \ \arg\max \left\{ \frac{u_i^T x}{\sigma^2} - \frac{u_i^T u_i}{2\sigma^2} + \log(P(w_i)) \right\}$$

quick check for any more constants that wouldn't affect the comparison. I think though σ^2 is technically identical for each we need to keep it in there because it scales the bias depending on $\log(P(w_i))$, multiplying out

I think:

$$g_i x \ \arg\max \left\{ u_i^T x - \frac{u_i^T u_i}{2} + \sigma^2 \cdot \log(P(w_i)) \right\}$$

where $W_i^T = u_i^T$, $W_{0i} = \frac{u_i^T u_i}{2} + \sigma^2 \log(P(w_i))$

Part 2 and 3 (coding)

- This is Part 2 and 3 (the coding sections) of HW_01
- From Quinn Hull (Robert Hull)

Outline:

2.Linear and Quadratic Classifiers

* 2a. General Function for Random Samples

* 2b. Procedure of the discriminant

* 2c. 2D dataset with three classes and quadratic classifier

* 2d. Mahalanobis Distance:

* 2e. Naive Bayes Classifier

3. Miscellaneous Code

* Problem II (Sampling from a Distribution)

In [1]:
```
### modules used in the below
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial import distance
from numpy.linalg import inv
import numpy as np
from numpy.linalg import det
from numpy.linalg import inv
from scipy.spatial import distance
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn import datasets
from sklearn.naive_bayes import GaussianNB
```

2a. General Function for Random Samples

- The code below generates random samples in d dimensions given a 1-D matrix of means and 2-D matrix of covariance that could conceivably be any real number
- It allows the user to flexibly implement whether or not to set the parameters needed to make this distribution or not

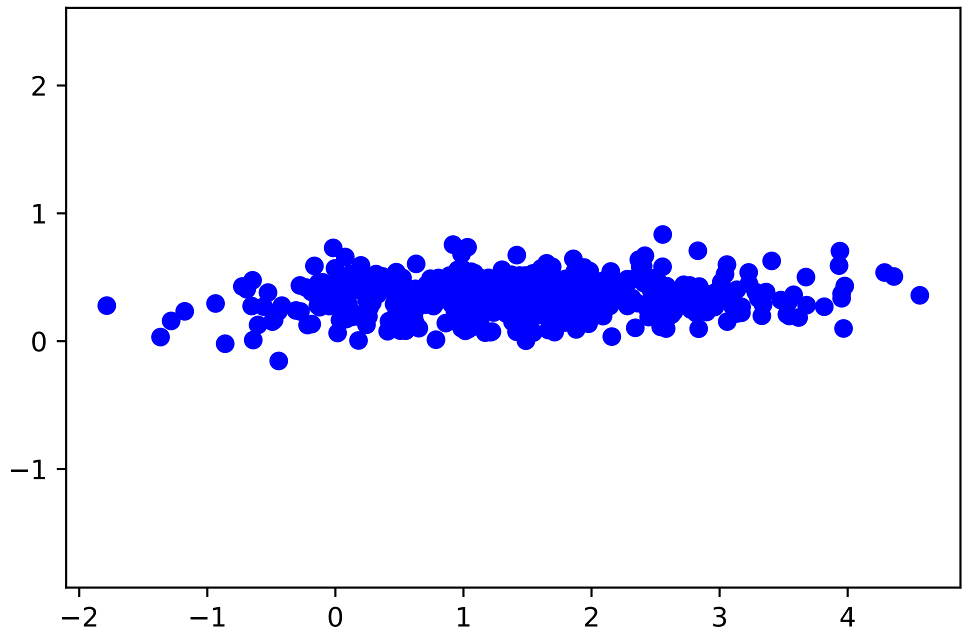
In [2]:
```
# *NOTE What is covariance matrix, really?
def randomsamples(d, size, up=1, down=-1, u=False, sig=False, condin = True, ret
```

```
    """A function to generate random samples
        inputs:
        d -> dimensions (int)
        size -> the size of the sample desired
        up -> the max of range of numbers to generate random
            (default 1)
        down -> the min of range of numbers to generate random
            (default -1)
        u -> optional input mean, a vector of size d
            (if not added, script will generate randomly)
        sig -> optional input covariance matrix, a matrix
            of dimensions d*d
            (if not added, script will generate randomly)
        condin -> conditional independence boolean
            if True (default) then off-diagonal
            values of sigma are zero
            if False, then any values in sigma
            may be a real number
        retall -> boolean for returning u and sigma
            True -> returns distribution, u, sig
            False -> returns distribution
            (default False)


        returns:
        a multivariate matrix sample with gaussian distribution

        and optionally u and sig
        """

    if u is False:
        ## means of dimensions 'd' [0, 1)
        u = np.random.uniform(down, up, size=(d,))

    if sig is False:
        ## covariance matrix of dimension 'd*d' [0,1)
        sig = np.random.uniform(down, up, size=(d,d))
        ## test for conditional independence
        if condin:
            sig = sig*np.identity(d)

    if retall:
        return np.random.multivariate_normal(u, sig, size).T, u, sig

    else:
        return np.random.multivariate_normal(u, sig, size).T
```

In [3]:

```
# returns a 2 dimensional randomly created variable and mean and covariances fro
d_r = 2 # dimensions
x_r, u_r, sig_r = randomsamples(2,500,d_r, retall=True) # returns information ab
print('the mean vector is, ', u_r)
print('the covariance matrix is, ', '\n', sig_r)
plt.plot(x_r[0], x_r[1], 'o', c='b')
plt.axis('equal')
plt.show()
```

```
the mean vector is,  [1.42932346 0.34754684]
the covariance matrix is,
 [[ 1.1613659  0.        ]
 [-0.        -0.0191755]]
```



2b. Procedure of the discriminant

- Although `skLearn` has a helpful library of discriminant related classifier analyses (like quadratic), I couldn't see a function to calculate the discriminant itself (gcx).
- The script below generates a procedure for calculating the discriminant, which is used later in the quadratic classifer problem
- Note also that this assumes that the distributional information about the sample are already known.

In [4]:
```
def discriminant(x, u, sig, pc, d, retall = False):
    """A procedure for calculating the discriminant
        function from a given:

        x -> values
        u -> means
        sig -> covariance
        pc -> prior probability of c
        d -> dimensions
        retall -> option to return spatial info for randomly
            created data
            (default False)

        returns:
        g(x) for a given class c (the discriminant)

        also returns all of the distributional data used to make calc
        (x, u, sig, pc) if retall=True
    """

    # calculate x - u
    xu = np.mat(x - u)
    # determinant of sigma
    detsig = det(sig)
    # inverse of sigma
    invsig = np.mat(inv(sig))
```

```
    # discriminant fnction
    gx = -0.5*xu*invsig*xu.T \
          - d/2*np.log(2*np.pi) - 1/2*np.log(detsig) \
          + np.log(pc)

    if retall:
        return gx, x, u, sig, pc

    else:
        return gx
```

In [5]:

```
# this procedure calculates the discriminant from first element of randomly gene
# note that as written it only works for individual inputs

# see previous randomly generated dataset
x_in = x_r[:,0] # first sample (w/ elements x_1 and x_2) from x_r

pc_r = np.random.uniform(0,1) # prior probability of rando class, randomly gener

print('x input is ', x_in, 'of shape', x_in.shape)
print('mean input is ', u_r, 'of shape', u_r.shape)
print('sigma input is ', sig_r, 'of shape', sig_r.shape)
print('probability input is ', pc_r, 'of ', type(pc_r), '\n')
print('the discriminant of this randomness is...', discriminant(x=x_in, u=u_r, s
```

```
x input is  [2.88384995 0.25615334] of shape (2,)
mean input is  [1.42932346 0.34754684] of shape (2,)
sigma input is  [[ 1.1613659  0.        ]
 [-0.        -0.0191755]] of shape (2, 2)
probability input is  0.7977752253286745 of  <class 'float'>

the discriminant of this randomness is... nan
```

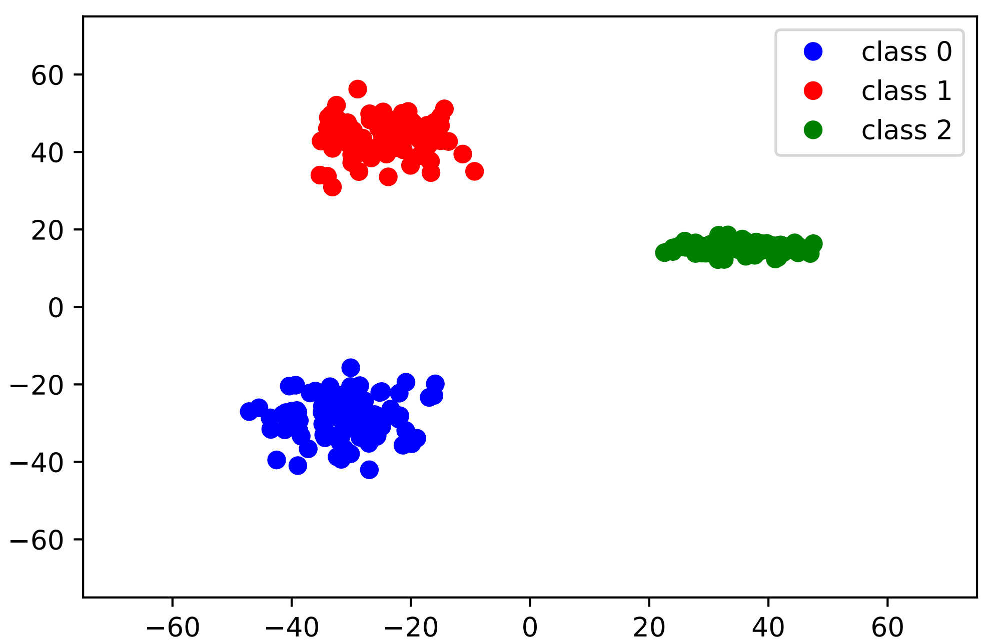
2c. 2D dataset with three classes and quadratic classifier

- The script is broken into several sections
  1. Generate three classes of random variables, 'learns' the distributional information from the sample, and plots them
  2. Uses the distributional information from the samples to **train** a quadratic classifier and then **test** it on some 2-dimensional test point. This is done using the discriminate function from the previous section, and compared to the dedicated function from `sklearn`
  3. Finally, the `sklearn` predict() function is used to generate decision boundary using a random, uniformly distributed dataset that is plotted as a scatterplot and categorized according to the prediction from the classifier

In [6]:

```
# A. generate 3 random variables and plot (will be used for the rest of this)

# 1. globals
d = 2 # dimensions
k = 100 # size of input
n = 15 # size of test subset
class_num = 3 # number of classes
up_in = 50 # upper bound of input data
down_in = -50 # lower bound of input data
```

```
# 2. create bivariate gaussian data with 3 known classes, that are conditionally
x_1, u_true1, sig_true1 = randomsamples(d,k,up=50,down=-50,retall=True)
x_2, u_true2, sig_true2 = randomsamples(d,k,up=50,down=-50,retall=True)
x_3, u_true3, sig_true3 = randomsamples(d,k,up=50,down=-50,retall=True)
x_train_list = [x_1, x_2, x_3]

# 3. reserve some as a test set (n number)
x_1_train, x_1_test = x_1[:,0:-n], x_1[:,-n:]
x_2_train, x_2_test = x_2[:,0:-n], x_2[:,-n:]
x_3_train, x_3_test = x_3[:,0:-n], x_3[:,-n:]
x_test_list = [x_1_test, x_2_test, x_3_test]

# 4. calculate mean and sigma from other (train) set for each of three classes
u_1, u_2, u_3 = np.mean(x_1_train, axis=1), np.mean(x_2_train, axis=1), np.mean(
sig_1, sig_2, sig_3 = np.var(x_1_train, axis=1), np.var(x_2_train, axis=1), np.v
u_list = [u_1, u_2, u_3]
sig_list = [sig_1, sig_2, sig_3]

# 5. list containing priors for each class_c
# assume equal priors (because there are the same
#   numbers in each class, they have a 1/c chance
#   of occuring, 1/3)
pri_list = [(1/3), (1/3), (1/3)]

# 6. plot
# color list for graphing
color_list = ['b', 'r', 'g']
for cla in range(class_num):
    plt.plot(x_train_list[cla][0], x_train_list[cla][1], 'o', c=color_list[cla],

plt.xlim(down_in*1.5, up_in*1.5)
plt.ylim(down_in*1.5, up_in*1.5)
plt.legend()
plt.show()
```



In [7]:

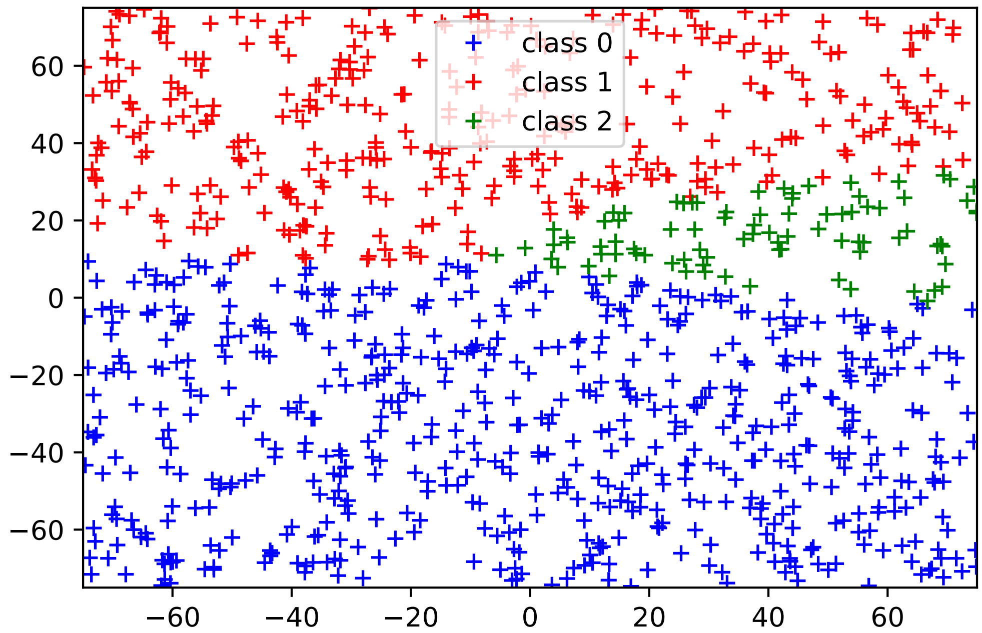
```
# B. train and test quadratic classifier, using both custom and sklearn function

# 1. take a single value from test dataset, from class 0
x_test = x_test_list[0][:,0]
```

```
# 2. convert sigs back to dxd
sig_listdd =[]
for sig in sig_list:
    sig_listdd.append(sig*np.identity(sig.shape[0]))


# 3. discriminant analysis tool, custom
disc_list = []
for i in range(class_num):
    disc = discriminant(x=x_test, u=u_list[i], sig=sig_listdd[i],
                        pc=pri_list[i], d=x_test.shape[0]).item()
    disc_list.append(disc)
    del disc

result = np.where(np.array(disc_list) == np.amax(np.array(disc_list)))[0].item()

print('the point is most likely located in class ', result, 'according to custom

# 4. discriminant analysis tool, sklearn
# *NOTE X and y are used later on in the Naive Baisian Classifer
X = np.concatenate(x_train_list,axis=1).T # training data, predictors
y = np.array([np.full((k),0), np.full((k),1), np.full((k),2)]).flatten() # train
clf = QuadraticDiscriminantAnalysis()
clf.fit(X, y)
result = clf.predict(np.array([x_test])).item()
print('the point is most likely located in class ', result, 'according to sklear
```

```
the point is most likely located in class  0 according to custom quadratic class
ification
the point is most likely located in class  0 according to sklearn quadratic clas
sification
```

In [8]:

```
# 3. Create decision boundaries

# predict decision boundaries from randomized 2x2 dataset
# use the sklearn to compare all values in a dataset
x_test_scatter = np.random.uniform(down_in*1.5, up_in*1.5, size=(d,k*10))

# make prediction
pred_scatter = clf.predict(x_test_scatter.T)

# plot
plt.plot(x_test_scatter[0][pred_scatter==0], x_test_scatter[1][pred_scatter==0],
plt.plot(x_test_scatter[0][pred_scatter==1], x_test_scatter[1][pred_scatter==1],
plt.plot(x_test_scatter[0][pred_scatter==2], x_test_scatter[1][pred_scatter==2],

plt.xlim(down_in*1.5, up_in*1.5)
plt.ylim(down_in*1.5, up_in*1.5)
plt.legend()
plt.show()
```



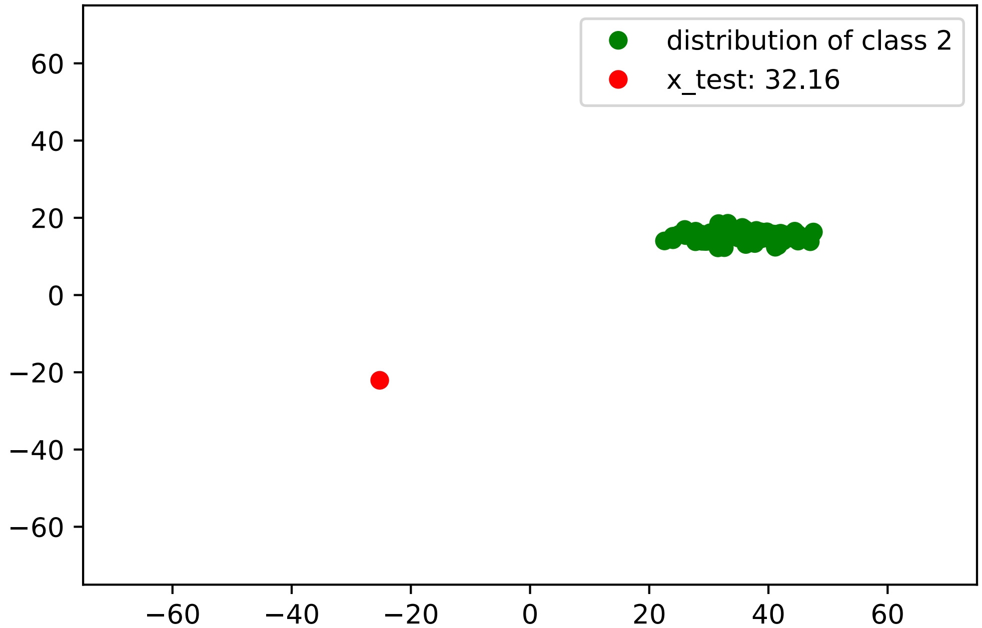
2d. Mahalanobis Distance:

- Two Mahalanobis Distance Equations are explored. The built-in scipy function and a custom one built using numpy arrays, using the mahalanobis equation as described in the scipy function documentation. This includes a square root symbol, which is not the case for the generalized form of the Mahalanobis equation shared in class.

In [9]:
```
# this procedure takes the distance between a random test point from class 0 and

# note, using the test data taken from earlier, class 0, a single point
x_test = x_test_list[0][:,0]
# extract average for later
u_train = u_list[2]
# invert sigma for later
sig_train = inv(sig_listdd[2])

# built in routine *NOTE the sqrt of the function you asked us to make
print("Man - distance, scipy routine")
dist = distance.mahalanobis(x_test,u_train,sig_train)
print(dist)

# custom routine *NOTE the sqrt of the function you asked us to make
print("Man - distance, custom routine")
dist_cust = np.sqrt(np.mat(x_test - u_train)*np.mat(sig_train)*np.mat(x_test-u_t
print(dist_cust)

# plot
plt.plot(x_train_list[2][0], x_train_list[2][1], 'o', c='g', label='distribution
plt.plot(x_test[0], x_test[1], 'o', c='r', label='x_test: '+str(np.round(dist,2)
plt.xlim(down_in*1.5, up_in*1.5)
plt.ylim(down_in*1.5, up_in*1.5)
plt.legend()
plt.show()
```

```
Man - distance, scipy routine
32.157041348223736
```

```
Man - distance, custom routine
32.157041348223736
```



2e. Bayes Classifier:

- given: the posterior probability (P(wj|x)) is proportional to the product of likelihood (P(x|wj)) and prior p(wj)
- assumptions:
  1. each class has an equal prior (given 100 samples in n=3 categories, prior = 100/300 = 1/3)
  2. the distribution of the likelihood (P(x|wj)) is gaussian, w/ known distribution
     - fx = np.exp(-(1/2)*(x-u)**2/sig^2)/(sig*np.sqrt(2*PI)) (1-D) <-- use this definition, and iterate
     - fx = np.exp(-(1/2)*(x-u).T*sig.inv*(x-u))/(sig*np.sqrt(2*PI)) (2-D)
  3. for a given sample X of dimension i=1 to d, the total likelihood of p(X|wj) is equal to the sum of the log of the individual probabilities x1, x2,..., xd given wj. np.sum(np.log(pxi|wj))
- want: to classify a test dataset into n categories based on which posterior (j=1-n) is highest for a given sample X

In [10]:
```
# *NOTE given that we assume conditional independence for this, we can assume th

# *NOTE this problem uses the data generated in previous steps

# 1. create function p(xi|wj) = fx (univariate probability classifier)
def fx(xi, uj, sj):
    return np.exp(-(1/2)*(xi-uj)**2/sj**2)/(sj*np.sqrt(2*np.pi))

# 2. calculate fx (prior) from mean and sigma for e/a class and dimension on all
# 2a. create holding place for posterioir probability for each class
    # col 1 is for like class 1, col 2 is for like class 2, etc...
pxw_list = [np.zeros([3, n]), np.zeros([3, n]), np.zeros([3, n])]

# 2b. loop through all elements and assign posterior probability
```

```
l = 0
for pxw in pxw_list: #all posterior lists for each class
    x_test = x_test_list[l] # select test data
    for allx in range(n): # each element x = n
        for i in range(3): # all three classifier conditions
            temp = 0
            for j in range(2): # each individual element
                # temp *= fx(x_1_test[j,allx],u_list[i][j],sig_list[i][j]) # thi
                temp += np.log(fx(x_test[j,allx],u_list[i][j],sig_list[i][j]))
                # print(i, j, temp)
            pxw[i,allx] = temp
    l += 1 # on to the next x_test

# for easy reference
pxw1_arr = pxw_list[0]
pxw2_arr = pxw_list[1]
pxw3_arr = pxw_list[2]

# 3. identify as class wj and compare to reality (it true, properly classified,
class1_arr = ((pxw1_arr[0]>pxw1_arr[1]) & (pxw1_arr[0]>pxw1_arr[2]))
class2_arr = ((pxw2_arr[1]>pxw2_arr[0]) & (pxw2_arr[1]>pxw2_arr[2]))
class3_arr = ((pxw3_arr[2]>pxw3_arr[1]) & (pxw3_arr[2]>pxw3_arr[0]))

print('using manual method: from', len(class1_arr)+len(class2_arr)+len(class3_ar
      'the number misclassified is ',
      len(class1_arr[class1_arr==False])+len(class2_arr[class2_arr==False])+len(

# 8. compare to sklearn
gnb = GaussianNB()
# *NOTE training X(predictors), and y(target) were generated earlier in this pro
x_pred = np.concatenate(x_test_list,axis=1).T
y_targ = np.array([np.full((n),0), np.full((n),1), np.full((n),2)]).flatten()
y_pred = gnb.fit(X, y).predict(x_pred)
print('using sklearn method: from', x_pred.shape[0],'the number misclassified is
```

```
using manual method: from 45 the number misclassified is  0
using sklearn method: from 45 the number misclassified is  0
```

3 Misc Code

Problem II: Sampling from a Distribution.

- Let the set of `N_arr` (of elements 1... n ... N) be a set of integers and `p` (of elements 1... n ... N) be a probability distribution, where `pk` is the probability of observaing `k` (an array including all real numbers up to `k`).
- Note that since `p` is a distribution then `1Tp` = 1, and 0<= `pk` <=1 for all instances of `n` .
  - I'm interpreting the above to mean that `p` can be interpreted as a cdf, wherein the value `p` for each element is the marginal probability for every value of `n` .

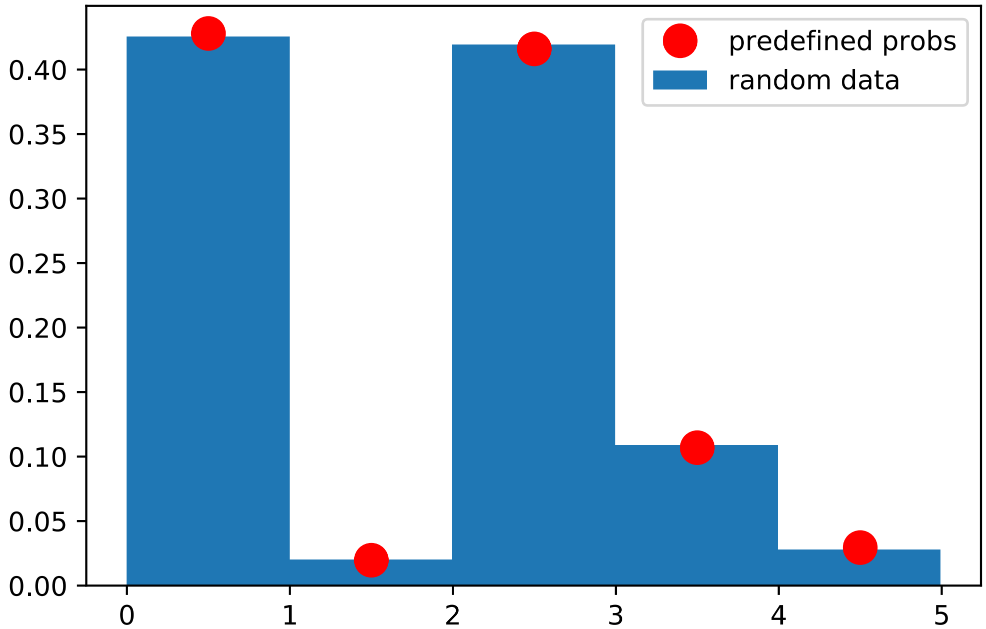
1. Write a function `sample(M, p)` that returns `M` indices sampled from the distributions `p`
2. Provide evidence that your function is working as desired
   - Note that all sampling is assumed to be i.i.d. (indepedendent and identically distributed)

Thoughts and reflection

- This question appears to be asking for us to write a random number generator for a given CDF
- In step 1, this script randomly generates a probability distribution (with a pdf of `p_pdf` and a cdf of `p`) for `N` integers from 1 to N. This probability distribution is used later
- In step 2, it randomly generates `M` values between 1 and `N` based on the probability distribution. For each `m` in `M` , a randomly generated value between 0 and 1 is treated as a 'target' probability. It then looks at the cdf from step 1 and finds the the index `p_i` closest probability to the target probability. It then randomly generates a value between `i` and `i+1` . This is the `mth` random variable in `M` .
- Step 3 compares the pdf of the probability distribution (in red) to a histogram of the `M` output variables to make sure that the distribution matches more or less the probability distribution put into the equation.

In [11]:

```
# 1. create an array of probabilities and samples
M = 10000 # number of samples to return in function
N = 5 # number of categories
N_arr = np.arange(0,N,1) # an array of the categories *NOTE unnecessary?
p = np.zeros(N) # cdf pobability array, assembled below!
p_pdf = np.zeros(N) # pdf probability array
# assign values of p, and p_pdf
for pk in range(N):
    p_cum = p[pk-1]
    # check to see if it is the last iteration
    if pk == N-1:
        p[pk] = 1
        p_pdf[pk] = 1-p_cum
    else:
        p_pdf[pk] = np.random.uniform(0,1-p_cum)
        p[pk] = p_cum+p_pdf[pk]


# 2. make function sample(M,p), random number distribution
def random(M_in,p_cdf):
    output = np.zeros(M_in)
    for m in range(M_in): # loop through all elements in array
        p_in = np.random.uniform(0,1) # generate a random probability
        p_i = np.where(p_cdf >= p_in)[0][0] # extract first index of condition
        output[m] = np.random.uniform(p_i, p_i+1) # randomly generate a number i
    return output

output = random(M,p)

# 3. show results
fig, ax = plt.subplots(1, 1)
ax.plot(N_arr+0.5, p_pdf, 'ro', ms=12, mec='r', label='predefined probs')
ax.hist(output,density=True,bins=N,rwidth=1, label='random data')
plt.legend()
plt.show()
```


predefined probs
random data
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
0
1
2
3
4
5

```
In [ ]:
```