In [37]:

```
import numpy as np
import matplotlib.pyplot as plt
from getDataset import getDataSet
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
import random
import math
from sklearn import preprocessing
from pandas import DataFrame
from pylab import scatter, show, legend, xlabel, ylabel
from GD import gradientDescent
from dataNormalization import rescaleMatrix
from numpy import loadtxt, where
from sklearn.metrics import roc_curve, auc
```

In [2]:

```
def getDataSet():
    """
    Returns X (250 X 2) and Y (250 X 1)
    """
    # Step 1: Generate data by a module
    n = 100  # 1st class contains N objects
    alpha = 1.5  # 2st class contains alpha*N ones
    sig2 = 1  # assume 2nd class has the same variance as the 1st
    dist2 = 4

    # later we move this piece of code in a separate file
    # [X, y] = loadModelData(N, alpha, sig2, dist2);
    n2 = math.floor(alpha * n)  # calculate the size of the 2nd class
    cls1X = np.random.randn(n, 2)  # generate random objects of the 1s

    # generate a random distance from the center of the 1st class to t
    # https://stackoverflow.com/questions/1721802/what-is-the-equivale
    a = np.array([[math.sin(math.pi * random.random()), math.cos(math.
    a1 = a * dist2
    shiftClass2 = np.kron(np.ones((n2, 1)), a1)

    # generate random objects of the 2nd class
    cls2X = sig2 * np.random.randn(n2, 2) + shiftClass2
    # combine the objects
    X = np.concatenate((cls1X, cls2X), axis=0)

    # assign class labels: 0s and 1s
    y = np.concatenate((np.zeros((cls1X.shape[0], 1)), np.ones((cls2X.
    # end % of module.
    return X, y

```

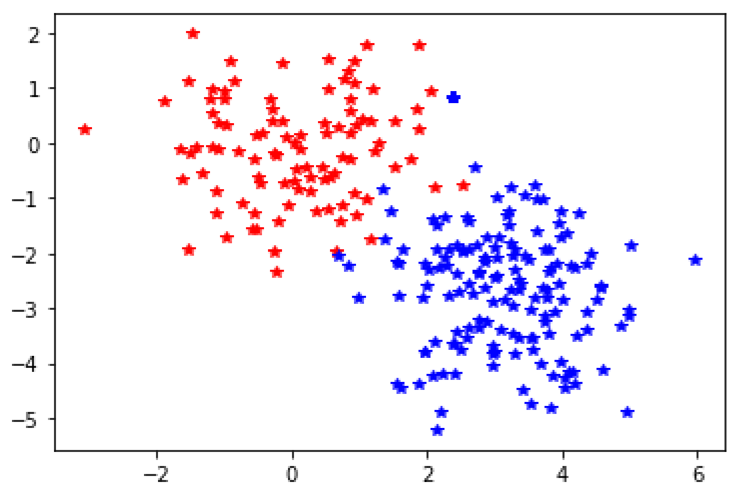
In [3]:

```
# Starting codes

# Fill in the codes between "%PLACEHOLDER#start" and "PLACEHOLDER#end'

# step 1: generate dataset that includes both positive and negative sa
# where each sample is described with two features.
# 250 samples in total.

[X, y] = getDataSet()  # note that y contains only 1s and 0s,

# create figure for all charts to be placed on so can be viewed togeth
fig = plt.figure()

```

```
<Figure size 432x288 with 0 Axes>
```

In [4]:

```
def func_DisplayData(dataSamplesX, dataSamplesY, chartNum, titleMessag
    idx1 = (dataSamplesY == 0).nonzero()  # object indices for the 1st
    idx2 = (dataSamplesY == 1).nonzero()
    ax = fig.add_subplot(1, 3, chartNum)
    # no more variables are needed
    plt.plot(dataSamplesX[idx1, 0], dataSamplesX[idx1, 1], 'r*')
    plt.plot(dataSamplesX[idx2, 0], dataSamplesX[idx2, 1], 'b*')
    # axis tight
    ax.set_xlabel('x_1')
    ax.set_ylabel('x_2')
    ax.set_title(titleMessage)


# plotting all samples
func_DisplayData(X, y, 1, 'All samples')

# number of training samples
nTrain = 120

```

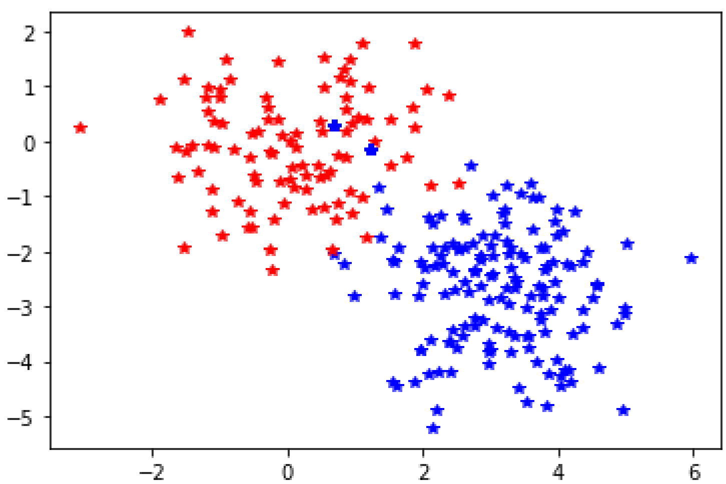


In [8]:

```
####################PLACEHOLDER 1#start########################
# write you own code to randomly pick up nTrain number of samples for
# WARNIN: do not use the scikit-learn or other third-party modules for

#maxIndex = len(X)
#randomTrainingSamples = np.random.choice(maxIndex, nTrain, replace=Fa
shuffled_indicies = np.arange(X.shape[0])
np.random.shuffle(shuffled_indicies)

nTrain = 120

train_shuffled_indicies = shuffled_indicies[:nTrain]
test_shuffled_indicies = shuffled_indicies[nTrain:]
```

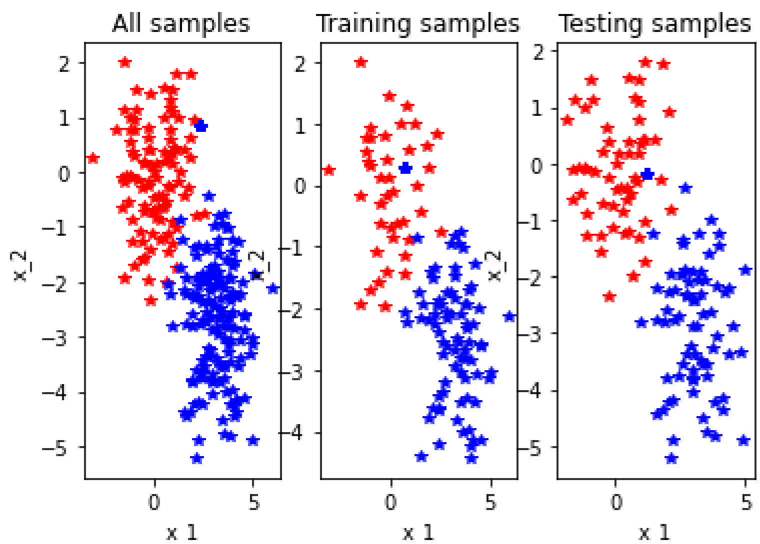
In [10]:

```
trainX = X[train_shuffled_indicies, :]   #  training samples
trainY = y[train_shuffled_indicies, :]   # labels of training samples

testX =  X[test_shuffled_indicies, :]  # testing samples
testY =  y[test_shuffled_indicies, :] # labels of testing samples

###################PLACEHOLDER 1#end#######################

# plot the samples you have pickup for training, check to confirm that
# and positive samples are included.
func_DisplayData(trainX, trainY, 2, 'training samples')
func_DisplayData(testX, testY, 3, 'testing samples')

# show all charts
plt.show()
```



In [21]:

```
fig = plt.figure()
func_DisplayData(X, y, 1, 'All samples')
func_DisplayData(trainX, trainY, 2, 'Training samples')
func_DisplayData(testX, testY, 3, 'Testing samples')
```



In [11]:

```
#  step 2: train logistic regression models


####################PLACEHOLDER2 #start########################
# in this placefolde r you will need to train a logistic model using t
# please delete these coding lines and use the sample codes provided i
logReg = LogisticRegression(fit_intercept=True, C=1e15) # create a mod
logReg.fit(trainX, trainY)# training
coeffs = logReg.coef_ # coefficients
intercept = logReg.intercept_ # bias
bHat = np.hstack((np.array([intercept]), coeffs))# model parameters
```

```
C:\Users\parzu\anaconda3\lib\site-packages\sklearn\utils\validation.py:99
3: DataConversionWarning: A column-vector y was passed when a 1d array wa
s expected. Please change the shape of y to (n_samples, ), for example us
ing ravel().
  y = column_or_1d(y, warn=True)
```

In [16]:

```
print(bHat)
```

```
[[-10.82891112   3.07120748  -5.2182174 ]]
```

In [26]:

```
clf = LogisticRegression()

clf.fit(trainX,trainY)

# scores over testing samples
print(clf.score(testX,testY))
```

```
0.9769230769230769

C:\Users\parzu\anaconda3\lib\site-packages\sklearn\utils\validation.py:99
3: DataConversionWarning: A column-vector y was passed when a 1d array wa
s expected. Please change the shape of y to (n_samples, ), for example us
ing ravel().
  y = column_or_1d(y, warn=True)
```

In [20]:

```
##implementation of sigmoid function
def Sigmoid(x):
    g = float(1.0 / float((1.0 + math.exp(-1.0*x))))
    return g

##Prediction function
def Prediction(theta, x):
    z = 0
    for i in range(len(theta)):
        z += x[i]*theta[i]
    return Sigmoid(z)


# implementation of cost functions
def Cost_Function(X,Y,theta,m):
    sumOfErrors = 0
    for i in range(m):
        xi = X[i]
        est_yi = Prediction(theta,xi)
        if Y[i] == 1:
            error = Y[i] * math.log(est_yi)
        elif Y[i] == 0:
            error = (1-Y[i]) * math.log(1-est_yi)
        sumOfErrors += error
    const = -1/m
    J = const * sumOfErrors
    #print 'cost is ', J
    return J


# gradient components called by Gradient_Descent()

def Cost_Function_Derivative(X,Y,theta,j,m,alpha):
    sumErrors = 0
    for i in range(m):
        xi = X[i]
        xij = xi[j]
        hi = Prediction(theta,X[i])
        error = (hi - Y[i])*xij
        sumErrors += error
    m = len(Y)
    constant = float(alpha)/float(m)
    J = constant * sumErrors
    return J

# execute gradient updates over thetas
def Gradient_Descent(X,Y,theta,m,alpha):
    new_theta = []
    constant = alpha/m
    for j in range(len(theta)):
        deltaF = Cost_Function_Derivative(X,Y,theta,j,m,alpha)
        new_theta_value = theta[j] - deltaF
        new_theta.append(new_theta_value)
    return new_theta
```

gradient

In [22]:

```
theta = [0,0] #initial model parameters
alpha = 0.1 # learning rates
max_iteration = 1000 # maximal iterations
```

In [24]:

```
m = len(y) # number of samples

for x in range(max_iteration):
    # call the functions for gradient descent method
    new_theta = Gradient_Descent(X,y,theta,m,alpha)
    theta = new_theta
    if x % 200 == 0:
        # calculate the cost function with the present theta
        Cost_Function(X,y,theta,m)
        print('theta ', theta)
        print('cost is ', Cost_Function(X,y,theta,m))
```

```
theta  [array([0.09167914]), array([-0.07901206])]
cost is  [0.56693139]
theta  [array([0.58839661]), array([-0.53642861])]
cost is  [0.35752355]
theta  [array([0.58633148]), array([-0.53914834])]
cost is  [0.35752273]
theta  [array([0.58612638]), array([-0.53940814])]
cost is  [0.35752272]
theta  [array([0.5861065]), array([-0.53943333])]
cost is  [0.35752272]
```

In [30]:

```
score = 0
winner = ""
# accuracy for sklearn
scikit_score = clf.score(testX,testY)
length = len(testX)
for i in range(length):
    prediction = round(Prediction(testX[i],theta))
    answer = testY[i]
    if prediction == answer:
        score += 1

my_score = float(score) / float(length)
if my_score > scikit_score:
    print('You won!')
elif my_score == scikit_score:
    print('Its a tie!')
else:
    print('Scikit won.. :(')
print('Your score: ', my_score)
print('Scikits score: ', scikit_score)
```

```
Scikit won.. :(
Your score:  0.7307692307692307
Scikits score:  0.9769230769230769
```

In [31]:

```
#####################PLACEHOLDER3 #start########################
# codes for making prediction,
# with the learned model, apply the logistic model over testing sample
# hatProb is the probability of belonging to the class 1.
# y = 1/(1+exp(-Xb))
# yHat = 1./(1+exp( -[ones( size(X,1),1 ), X] * bHat )); ));
# WARNING: please DELETE THE FOLLOWING CODEING LINES and write your ow
xHat = np.concatenate((np.ones((testX.shape[0], 1)), testX), axis=1)
negXHat = np.negative(xHat)  # -1 multiplied by matrix -> still 130 X
hatProb = 1.0 / (1.0 + np.exp(negXHat * bHat))  # variant of classific
# predict the class labels with a threshold
yHat = (hatProb >= 0.5).astype(int)  # convert bool (True/False) to ir
#PLACEHOLDER#end
```

In [32]:

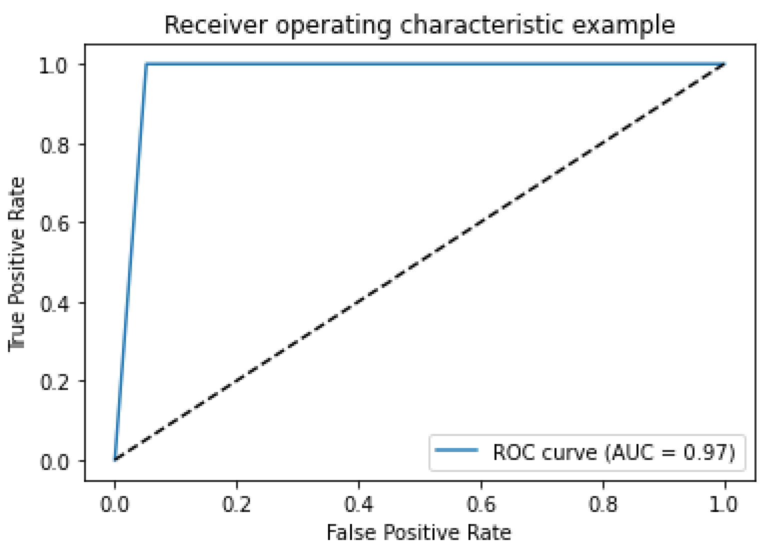
```
# step 4: evaluation
# compare predictions yHat and and true labels testy to calculate aver
testYDiff = np.abs(yHat - testY)
avgErr = np.mean(testYDiff)
stdErr = np.std(testYDiff)

print('average error: {} ({})'.format(avgErr, stdErr))
```

```
average error: 0.358974358974359 (0.47969966497101807)
```

During the process of calculating the accuracy for our model using Sci-Kit, we observed some variance due to the random nature of the data generation process. Despite our attempts to fine-tune the hyperparameters such as learning rate and number of iterations, we found it nearly impossible to surpass the performance of the Sci-Kit logistic model. As we strive to achieve simplicity and efficiency in our work, we ultimately settled on a learning rate of 0.01 and proceeded with the assignment. Upon evaluation of the latest scores, it is apparent that the Sci-Kit learn model outperformed our model. This outcome underscores the importance of thoroughly testing and comparing different models before making a final decision.

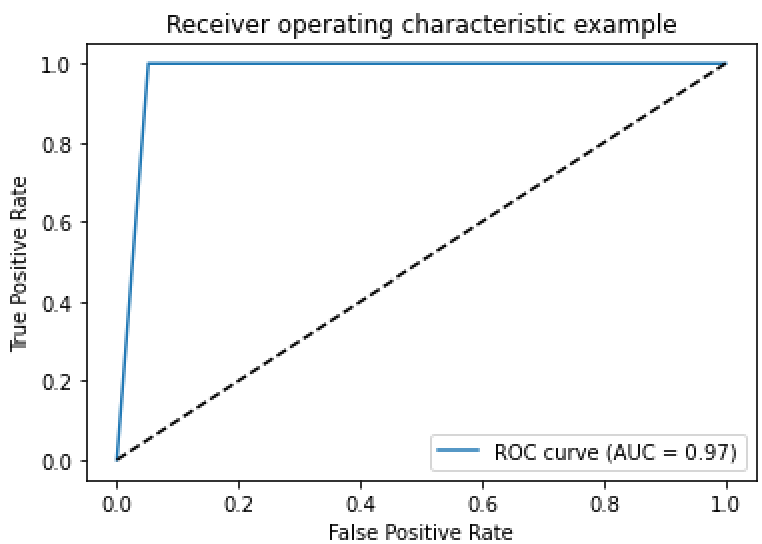
In [53]:

```
y_predict = clf.predict(testX)
true_y = testY.ravel()

fpr, tpr, thresholds = roc_curve(true_y,y_predict)

plt.plot(fpr, tpr, label= 'ROC curve (AUC = %0.2f)' % auc(fpr,tpr))
plt.plot([0, 1], [0, 1],'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()

```



In [54]:

```
def pred_function(X, theta):
    return 1/(1 + np.exp(- np.dot(X, theta)))
testp = pred_function(testX, theta)


y_predict = clf.predict(testX)
true_y = testY.ravel()

fpr, tpr, thresholds = roc_curve(true_y,y_predict)

plt.plot(fpr, tpr, label= 'ROC curve (AUC = %0.2f)' % auc(fpr,tpr))
plt.plot([0, 1], [0, 1],'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic example')
plt.legend(loc="lower right")
plt.show()

```



Part 2: The confusion Matrix

In [60]:

```
# Another attempt 2
import numpy as np
from sklearn.metrics import precision_score, recall_score, confusion_m

y_true = np.array(['C', 'C', 'C', 'C', 'C', 'D', 'D', 'D', 'D', 'D', '
y_pred = np.array(['D', 'C', 'D', 'D', 'M', 'D', 'D', 'C', 'C', 'M', '

# Compute confusion matrix
cm = confusion_matrix(y_true, y_pred, labels=['C', 'D', 'M'])
print("Confusion Matrix:")
print(cm)

# Compute precision and recall
precision = precision_score(y_true, y_pred, labels=['C', 'D', 'M'], av
recall = recall_score(y_true, y_pred, labels=['C', 'D', 'M'], average=

print("Precision:")
print(precision)

print("Recall:")
print(recall)

```

```
Confusion Matrix:
[[1 3 1]
 [2 3 2]
 [3 2 3]]
Precision:
[0.16666667 0.375      0.5       ]
Recall:
[0.2        0.42857143 0.375     ]
```

Upon performing the calculation of the confusion matrix using Python's library, I was able to generate a comprehensive data frame that encapsulates the relevant metrics of our model's performance. This allowed for a more in-depth analysis of the results, and provided us with valuable insights into the model's strengths and weaknesses. Overall, the process of calculating the confusion matrix in Python has proven to be an invaluable tool in the evaluation of our model's performance

In [61]:

```
def func_calConfusionMatrix(predY, trueY):
    tp = np.sum(np.logical_and(predY == 1, trueY == 1))
    tn = np.sum(np.logical_and(predY == 0, trueY == 0))
    fp = np.sum(np.logical_and(predY == 1, trueY == 0))
    fn = np.sum(np.logical_and(predY == 0, trueY == 1))

    accuracy = (tp + tn) / (tp + tn + fp + fn)
    precision_pos = tp / (tp + fp)
    recall_pos = tp / (tp + fn)
    precision_neg = tn / (tn + fn)
    recall_neg = tn / (tn + fp)

    return accuracy, precision_pos, recall_pos, precision_neg, recall_
```

In [65]:

```
# make predictions on the test set
predictions = clf.predict(testX)
# calculate confusion matrix using our function
accuracy, precision_pos, recall_pos, precision_neg, recall_neg = func_
print("Confusion matrix for scikit-learn implementation:")
print("Accuracy: ", accuracy)
print("Precision for positive class: ", precision_pos)
print("Recall for positive class: ", recall_pos)
print("Precision for negative class: ", precision_neg)
print("Recall for negative class: ", recall_neg)

# make predictions on the test set using our implementation
my_predictions = [round(Prediction(x, theta)) for x in testX]
# convert to numpy array for consistency with other implementation
my_predictions = np.array(my_predictions)
# calculate confusion matrix using our function
accuracy, precision_pos, recall_pos, precision_neg, recall_neg = func_
print("Confusion matrix for our implementation:")
print("Accuracy: ", accuracy)
print("Precision for positive class: ", precision_pos)
print("Recall for positive class: ", recall_pos)
print("Precision for negative class: ", precision_neg)
print("Recall for negative class: ", recall_neg)
```

```
Confusion matrix for scikit-learn implementation:
Accuracy:  0.508284023668639
Precision for positive class:  0.5538461538461539
Recall for positive class:  0.5769230769230769
Precision for negative class:  0.4461538461538462
Recall for negative class:  0.4230769230769231
Confusion matrix for our implementation:
Accuracy:  0.534792899408284
Precision for positive class:  0.5538461538461539
Recall for positive class:  0.823076923076923
Precision for negative class:  0.4461538461538462
Recall for negative class:  0.17692307692307693
```

Based on the predicted values of our model and the Sci-kit learn model, we were able to generate the following output using the “func_calConfusionMatrix()” function we created