Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel$\rightarrow$Restart) and then **run all cells** (in the menubar, select Cell$\rightarrow$Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
NAME = "Aung Zar Lin"
ID = "121956"
```

Lab 05: Optimization Using Newton's Method

In this lab, we'll explore an alternative to gradient descent for nonlinear optimization problems: Newton's method.

Newton's method in one dimension

Consider the problem of finding the *roots* $\textbf{x}$ of a nonlinear function $f: \mathbb{R}^N \rightarrow \mathbb{R}$. A root of f is a point $\mathbf{x}$ that satisfies $f(\mathbf{x}) = 0$.

In one dimension, Newton's method for finding zeroes works as follows:

1. Pick an initial guess x_0
2. Let $x_{i+1} = x_i + \frac{f(x_i)}{f'(x_i)}$
3. If not converged, go to #2.

Convergence occurs when $|f(x_i)| < \epsilon_1$ or when $|f(x_{i+1})-f(x_i)| < \epsilon_2$.

Let's see how this works in practice.

Example 1: Root finding for a cubic polynomial

Let's begin by using Newton's method to find roots of a simple cubic polynomial $$ f(x) = x^3 + x^2 .$$

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
```

Here's a function to evaluate a polynomial created with Numpy's `poly1d` function at a particular point x:

In [3]:
```
def fx(x, p):
    f_x = np.polyval(p, x)
```

```
    return f_x
```

And here's some code to create the polynomial $x^3 + x^2$, get its derivative, and evalute the derivative at 200 points along the x axis:;

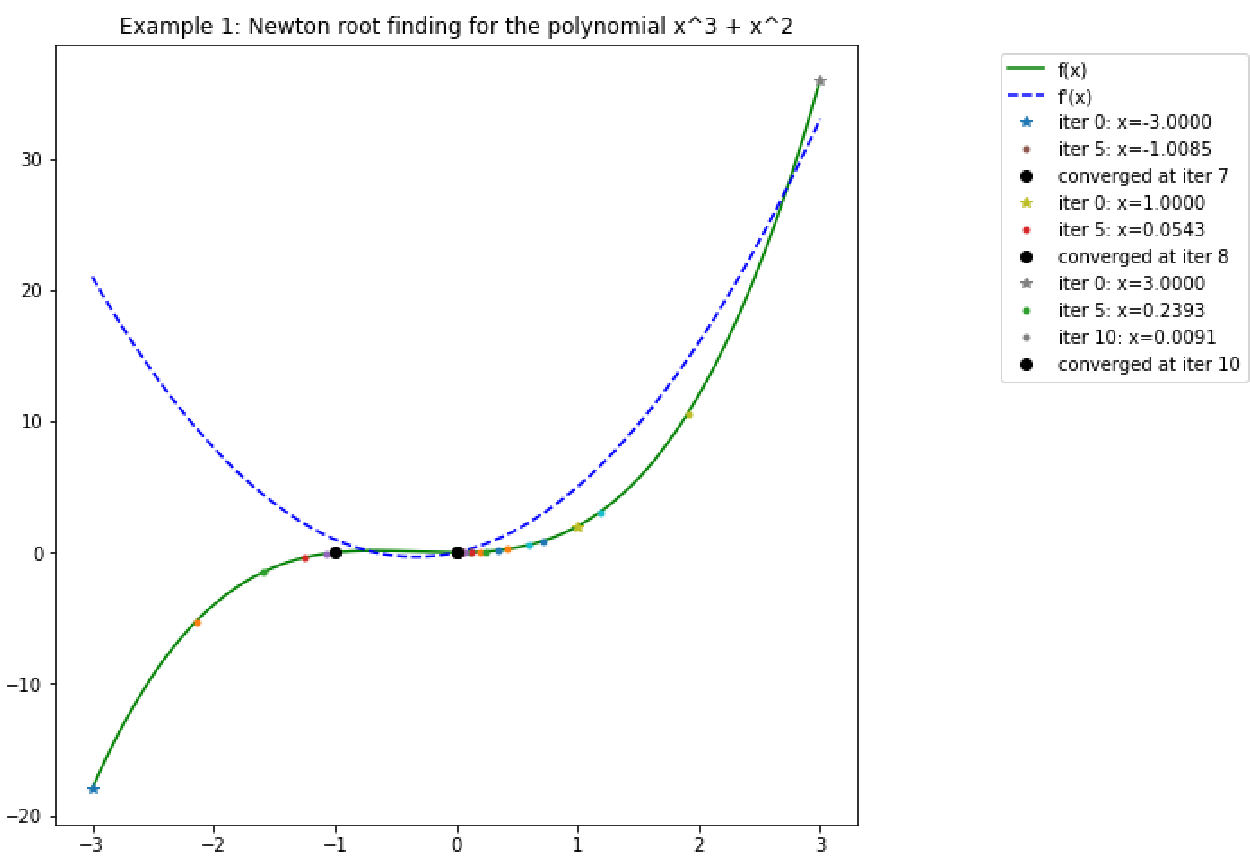
In [4]:
```
# Create the polynomial f(x) = x^3 + x^2
p = np.poly1d([1, 1, 0, 0]) # [1 * x^3, 1 * x^2, 0 * x^1, 0 * 1]

# Get f'(x) (the derivative of f(x) in polynomial form)
# We know it's 2x^2 + 2x, which is [3, 2, 0] in poly1d form
p_d = np.polyder(p)

print('f(x):')
print('----------')
print(p)
print('----------')
print("f'(x):")
print('----------')
print(p_d)
print('----------')

# Get 200 points along the x axis between -3 and 3
n = 200
x = np.linspace(-3, 3, n)

# Get values for f(x) and f'(x) in order to graph them later
y = fx(x, p)
y_d = fx(x,p_d)
```

```
f(x):
----------
   3     2
1 x + 1 x
----------
f'(x):
----------
   2
3 x + 2 x
----------
```

Next, let's try three possible guesses for x_0: -3, 1, and 3, and in each case, run Newton's root finding method from that initial guess.

In [5]:
```
# Initial guesses
x0_arr = [-3.0, 1.0, 3.0]

# Parameters for Newton: number of iterations,
# threshold for identifying a point as a zero
max_iters = 30
threshold = 0.0001

# Set up plot
fig1 = plt.figure(figsize=(8,8))
ax = plt.axes()
plt.plot(x, y, 'g-', label='f(x)')
plt.plot(x, y_d, 'b--', label="f'(x)")

roots = []
```

```
for x0 in x0_arr:
    i = 0
    xi = x0
    fxi = fx(xi, p)
    # Plot initial data point
    plt.plot(xi, fxi, '*', label=("iter 0: x=%.4f" % x0))
    while i < max_iters:
        # x_i+1 = x_i - f(x_i)/f'(x_i)
        xi = xi - fx(xi, p) / fx(xi, p_d)
        fxi = fx(xi, p)
        # Plot (xi, fxi) and add a legend entry every 5 iterations
        if (i+1) % 5 == 0:
            plt.plot(xi, fxi, '.', label=("iter %d: x=%.4f" % (i+1, xi)))
        else:
            plt.plot(xi, fxi, '.')
        # Check if |f(x)| < threshold
        if np.abs(fxi) < threshold:
            roots.append(xi)
            break
        i = i + 1
    plt.plot(xi, fx(xi, p), 'ko', label=("converged at iter %d" % (i+1)))

plt.legend(bbox_to_anchor=(1.5, 1.0), loc='upper right')
plt.title('Example 1: Newton root finding for the polynomial x^3 + x^2')

plt.show()
```



Example 2: Root finding for the sine function

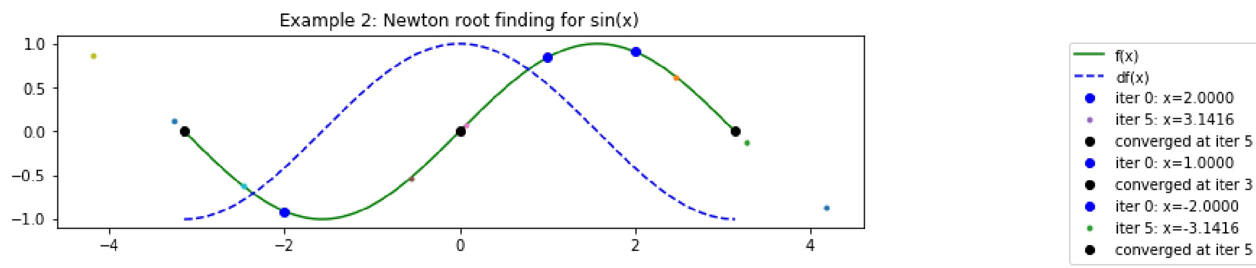
Next, consider the function $f(x) = \sin(x)$:

In [6]:
```
def fx_sin(x):
    return np.sin(x)

def fx_dsin(x):
    return np.cos(x)
```

Let's get 200 points in the range $[-\pi, \pi]$ for plotting:

In [7]:
```
# Get f(x)=sin(x) and f'(x) at 200 points for plotting
n = 200
x = np.linspace(-np.pi, np.pi, n)
y = fx_sin(x)
y_d = fx_dsin(x)
```

In [8]:
```
# Initial guesses
x0_arr = [2.0, 1.0, -2.0]

# Parameters for Newton: number of iterations,
# threshold for identifying a point as a zero
max_iters = 30
threshold = 0.0001

# Set up plot
fig1 = plt.figure(figsize=(10,10))
ax = plt.axes()
ax.set_aspect(aspect='equal', adjustable='box')
plt.plot(x, y, 'g-', label='f(x)')
plt.plot(x, y_d, 'b--', label='df(x)')

roots = []
for x0 in x0_arr:
    i = 0;
    xi = x0
    fxi = fx_sin(xi)
    # Plot initial data point
    plt.plot(xi, fxi, 'bo', label=("iter 0: x=%.4f" % x0))
    while i < max_iters:
        # x_i+1 = x_i - f(x_i)/f'(x_i)
        xi = xi - fx_sin(xi) / fx_dsin(xi)
        fxi = fx_sin(xi)
        # Plot (xi, fxi) and add a legend entry every 5 iterations
        if (i+1) % 5 == 0:
            plt.plot(xi, fxi, '.', label=("iter %d: x=%.4f" % (i+1, xi)))
        else:
            plt.plot(xi, fxi, '.')
        # Check if |f(x)| < threshold
        if np.abs(fxi) < threshold:
            roots.append(xi)
            break
        i = i + 1
    plt.plot(xi, fx_sin(xi), 'ko', label=("converged at iter %d" % (i+1)))

plt.legend(bbox_to_anchor=(1.5, 1.0), loc='upper right')
plt.title('Example 2: Newton root finding for sin(x)')

plt.show()

print('Roots: %f, %f, %f' % (roots[0], roots[1], roots[2]))
```




```
Roots: 3.141593, -0.000096, -3.141593
```

Notice that we get some extreme values of x for some cases. For example, when $x_0 = -2$, where the slope is pretty close to 0, the next iteration gives a value less than -4.

Newton's method for optimization

Now, consider the problem of minimizing a scalar function $J : \mathbb{R}^n \mapsto \mathbb{R}$. We would like to find $$ \theta^* = \text{argmin}_\theta J(\theta) $$ We already know gradient descent: $$ \theta^{(i+1)} \leftarrow \theta^{(i)} - \alpha \nabla_J(\theta^{(i)}).$$ But Newton's method gives us a potentially faster way to find θ^* as a zero of the system of equations $$\nabla_J(\theta^*) = \mathbf{0}.$$

In one dimension, to find the zero of $f'(x)$, obviously, we would apply Newton's method to $f'(x)$, obtaining the iteration $$x_{i+1} = x_i - f'(x_i) / f''(x_i).$$ The multivariate extension of Newton's optimization method is $$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathtt{H}^{-1}_f(\mathbf{x}_i) \nabla_f(\mathbf{x}_i),$$ where $\mathtt{H}_f(\mathbf{x})$ is the *Hessian* of f evaluated at $\textbf{x}$: $$\mathtt{H}_f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 x_n} \\ \frac{\partial^2 f}{\partial x_2 x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n x_1} & \frac{\partial^2 f}{\partial x_n x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

This means, for the minimization of $J(\theta)$, we would obtain the update rule $$\theta^{(i+1)} \leftarrow \theta^{(i)} - \mathtt{H}^{-1}_J(\theta^{(i)}) \nabla_J(\theta^{(i)}).$$

Application to logistic regression

Let's create some difficult sample data as follows:

Class 1: Two features x_1 and x_2 jointly distributed as a two-dimensional spherical Gaussian with parameters

$$\mu = \begin{bmatrix} x_{1c} \\ x_{2c} \end{bmatrix}, \Sigma = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_1^2 \end{bmatrix}.$$
Class 2: Two features x_1 and x_2 in which the data are generated by first sampling an angle θ according to a uniform distribution, sampling a distance d according to a one-dimensional Gaussian with a mean of $(3\sigma_1)^2$ and a variance of $(\frac{1}{2}\sigma_1)^2$,

then outputting the point $$\textbf{x} = \begin{bmatrix} x_{1c} + d \cos\theta \\ x_{2c} + d \sin\theta \end{bmatrix} .$$

Generate 100 samples for each of the classes, guided by the following exercises.

Exercise 1.1 (5 points)

Generate data for class 1 with 100 samples:

$$\mu = \begin{bmatrix} x_{1c} \\ x_{2c} \end{bmatrix}, \Sigma = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_1^2 \end{bmatrix}.$$

▶ **Hint:**

In [9]:

```
mu_1 = np.array([1.0, 2.0])
sigma_1 = 1
num_sample = 100

cov_mat = np.array([[sigma_1, 0], [0, sigma_1]])
X1 = np.random.multivariate_normal(mu_1, cov_mat, num_sample)

# YOUR CODE HERE
#raise NotImplementedError()
```

In [10]:

```
print(X1[:5])

# Test function: Do not remove
assert X1.shape == (100, 2), 'Size of X1 is incorrect'
assert cov_mat.shape == (2, 2), 'Size of x_test is incorrect'
count = 0
for i in range(2):
    for j in range(2):
        if i==j and cov_mat[i,j] != 0:
            if cov_mat[i,j] == sigma_1:
                count += 1
        else:
            if cov_mat[i,j] == 0:
                count += 1
assert count == 4, 'cov_mat data is incorrect'

print("success!")
# End Test function
```

```
[[0.05972066 2.02507983]
 [0.22351787 1.35675528]
 [0.51964659 5.5349679 ]
 [1.18262603 3.3564972 ]
 [0.89393368 2.9793407 ]]
success!
```

Expected result (or something similar):\ [[-0.48508229 2.65415886]\ [1.17230227 1.61743589]\ [-0.61932146 3.53986541]\ [0.70583088 1.45944356]\ [-0.93561505 0.2042285]]

Exercise 1.2 (5 points)

Generate data for class 2 with 100 samples:

$$\textbf{x} = \begin{bmatrix} x_{1c} + d \cos\theta \\ x_{2c} + d \sin\theta \end{bmatrix}$$
where θ is sampled uniformly from $[0, 2\pi]$ and d is sampled from a one-dimensional Gaussian with a mean of $(3\sigma_1)^2$ and a variance of $(\frac{1}{2}\sigma_1)^2$.

▶ **Hint:**

In [11]:
```
# 1. Create sample angle from 0 to 2pi with 100 samples
angle = np.random.uniform(low=0, high=2*np.pi, size = num_sample)
# 2. Create sample with normal distribution of d with mean and variance
d =  np.random.normal(np.square(3*sigma_1), np.square(sigma_1*.5), size =num_sample)
# 3 Create X2
X2 = np.array([X1[:,0] + d*np.cos(angle), X1[:,1] + d*np.sin(angle)]).T

# YOUR CODE HERE
#raise NotImplementedError()
```

In [12]:
```
print('angle:',angle[:5])
print('d:', d[:5])
print('X2:', X2[:5])

# Test function: Do not remove
assert angle.shape == (100,) or angle.shape == (100,1) or angle.shape == 100, 'Size of
assert d.shape == (100,) or d.shape == (100,1) or d.shape == 100, 'Size of d is incorre
assert X2.shape == (100,2), 'Size of X2 is incorrect'
assert angle.min() >= 0 and angle.max() <= 2*np.pi, 'angle generate incorrect'
assert d.min() >= 8 and d.max() <= 10, 'd generate incorrect'
assert X2[:,0].min() >= -13 and X2[:,0].max() <= 13, 'X2 generate incorrect'
assert X2[:,1].min() >= -10 and X2[:,1].max() <= 13.5, 'X2 generate incorrect'

print("success!")
# End Test function
```

```
angle: [1.97074405 1.598663   4.14390958 2.78534555 1.95595168]
d: [8.66434792 8.86771789 9.20158249 8.87126579 8.79867849]
X2: [[-3.31391817 10.0056491 ]
 [-0.02356393 10.22103027]
 [-4.43403666 -2.2193949 ]
 [-7.13163319  6.45043489]
 [-2.41175725 11.13342866]]
success!
```

Expected result (or something similar):\ angle: [4.77258271 3.19733552 0.71226709 2.11244845 6.06280915]\ d: [9.13908279 8.84218552 9.24427852 8.74831667 8.85727588]\ X2: [[0.064701 -6.46837219]\ [-7.65614929 1.12480234]\ [6.37750805 9.58147629]\ [-3.80438416 8.95550952]\ [7.70745021 -1.73194274]]

Exercise 1.3 (5 points)

Combine X1 and X2 into single dataset

In [13]:
```
# 1. concatenate X1, X2 together
X = np.concatenate([X1, X2], axis=0)
# 2. Create y with class 1 as 0 and class 2 as 1
y1 = np.zeros((num_sample,1))
y2 = np.ones((num_sample,1))
```

```
y = np.concatenate([y1, y2], axis=0)
# YOUR CODE HERE
#raise NotImplementedError()
```

In [14]:

```
print("shape of X:", X.shape)
print("shape of y:", y.shape)

# Test function: Do not remove
assert X.shape == (200, 2), 'Size of X is incorrect'
assert y.shape == (200,) or y.shape == (200,1) or y.shape == 200, 'Size of y is incorre
assert y.min() == 0 and y.max() == 1, 'class type setup is incorrect'

print("success!")
# End Test function
```

```
shape of X: (200, 2)
shape of y: (200, 1)
success!
```

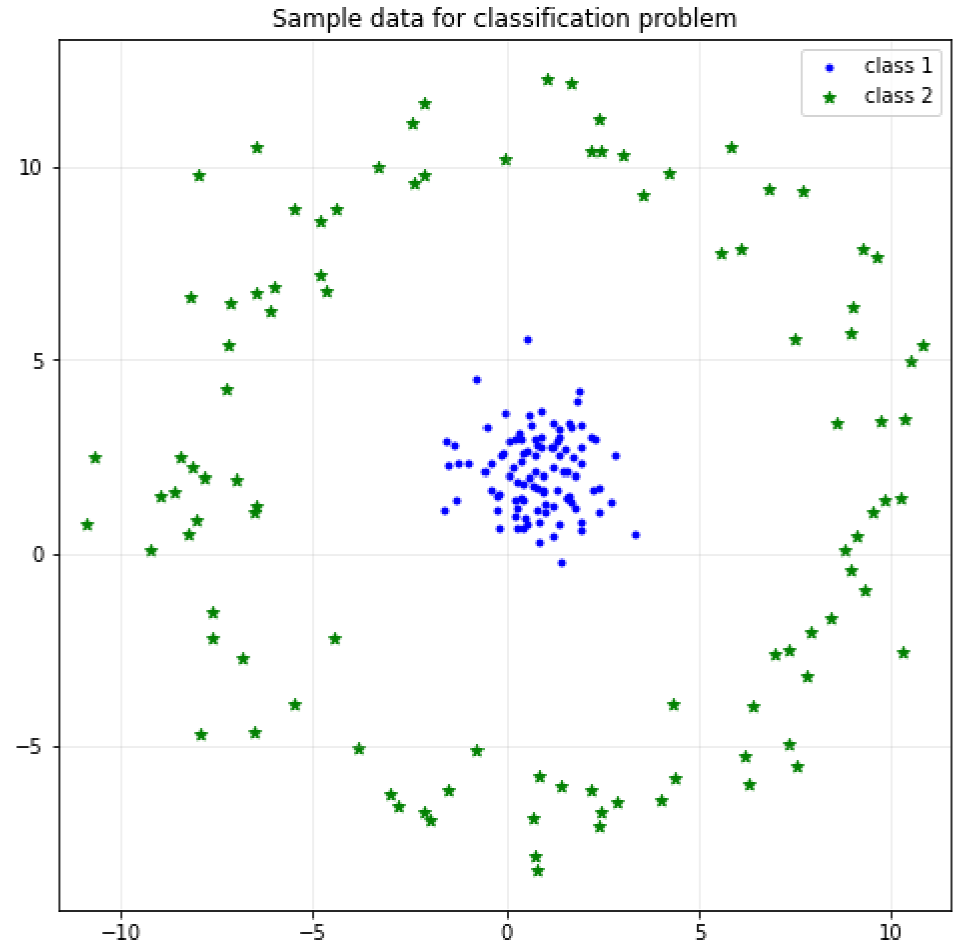
Expect result (or looked alike):\ shape of X: (200, 2)\ shape of y: (200, 1)

Exercise 1.4 (5 points)

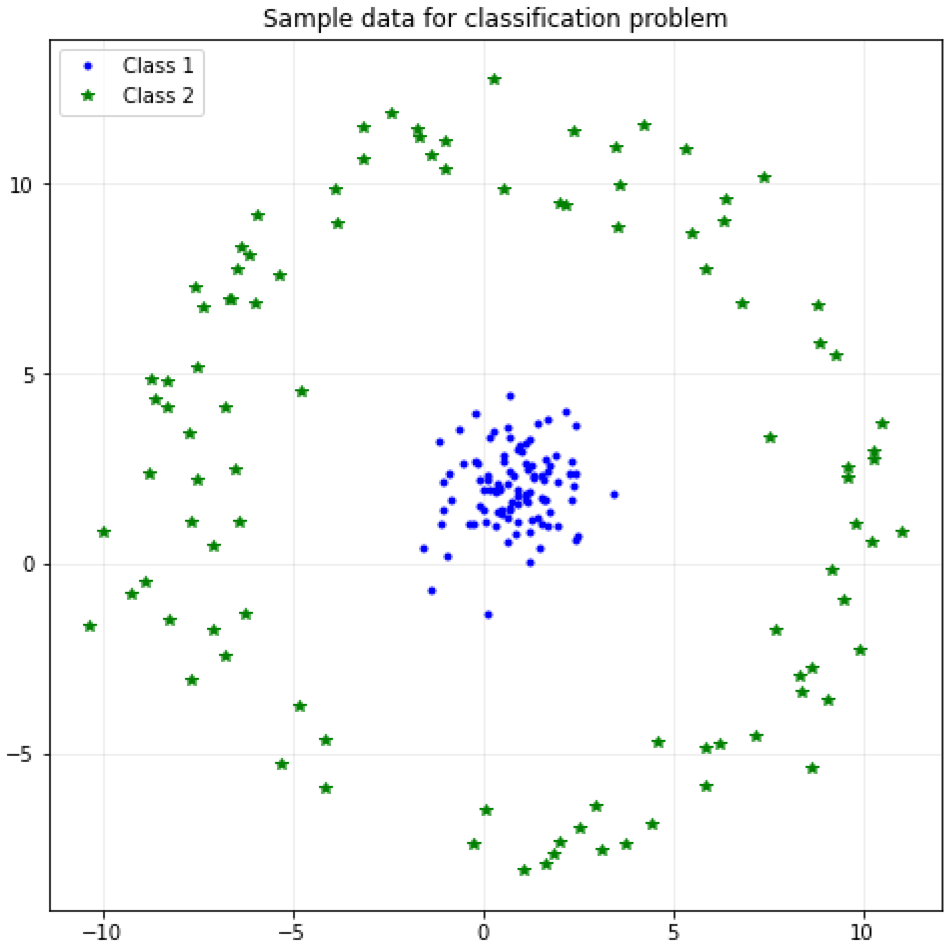
Plot the graph between class1 and class2 with **difference color and point style**.

In [15]:

```
fig1 = plt.figure(figsize=(8,8))
ax = plt.axes()
plt.title('Sample data for classification problem')
plt.grid(axis='both', alpha=.25)
# plot graph here
# YOUR CODE HERE
plt.scatter(X1[:,0], X1[:,1], c='b', marker='.', label='class 1')
plt.scatter(X2[:,0], X2[:,1], c='g', marker='*', label='class 2')
plt.legend()
#raise NotImplementedError()
# end plot graph
plt.axis('equal')
plt.show()
```



Expect result (or looked alike):



Exercise 1.5 (5 points)

Split data into training and test datasets with 80% of training set and 20% of test set

In [16]:
```
train_size = 0.8

m, n = X.shape
idx = np.arange(m)
np.random.shuffle(idx)

train_m = int(m * train_size)

idx_train = idx[:train_m]
idx_test = idx[train_m:]

X_train = X[idx_train]
X_test = X[idx_test]
y_train = y[idx_train]
y_test = y[idx_test]

# YOUR CODE HERE
#raise NotImplementedError()
```

In [17]:
```
print('idx_train:', idx_train[:10])
print("train size, X:", X_train.shape, ", y:", y_train.shape)
print("test size,  X:", X_test.shape,  ", y:", y_test.shape)
```

```
# Test function: Do not remove
assert X_train.shape == (160, 2), 'Size of X_train is incorrect'
assert y_train.shape == (160,) or y_train.shape == (160,1) or y.shape == 160, 'Size of
assert X_test.shape == (40, 2), 'Size of X_test is incorrect'
assert y_test.shape == (40,) or y_test.shape == (40,1) or y.shape == 40, 'Size of y_tes

print("success!")
# End Test function
```

```
idx_train: [119  45  14  57  47 145 158 136 159  33]
train size, X: (160, 2) , y: (160, 1)
test size,  X: (40, 2) , y: (40, 1)
success!
```

Expected reult (or something similar):\ idx_train: [78 61 28 166 80 143 6 76 98 133]\ train size, X: (160, 2) , y: (160, 1) \ test size, X: (40, 2) , y: (40, 1)

Exercise 1.6 (5 points)

Write a function to normalize your \$\mathtt{X}\$ data

Practice yourself (No grade, but has extra score 3 points)

Try to use Jupyter notebook's LaTeX equation capabilities to write the normalization equations for your dataset.

YOUR ANSWER HERE

\$\$Z=\frac{X-\mu}{\sigma}\$\$
where \$\mu\$ = mean and \$\sigma\$ = standard deviation

In [18]:

```
def normalization(X):
    """
    Take in numpy array of X values and return normalize X values,
    the mean and standard deviation of each feature
    """
    means = np.mean(X, axis=0)
    stds = np.std(X, axis=0)
    X_norm = (X - means) / stds
    # YOUR CODE HERE
    #raise NotImplementedError()

    return X_norm
```

In [19]:

```
XX = normalization(X)

X_train_norm = XX[idx_train]
X_test_norm = XX[idx_test]

# Add 1 at the first column of training dataset (for bias) and use it when training
X_design_train = np.insert(X_train_norm,0,1,axis=1)
X_design_test = np.insert(X_test_norm,0,1,axis=1)

m,n = X_design_train.shape

print(X_train_norm.shape)
```

```python
print(X_design_train.shape)
print(X_test_norm.shape)
print(X_design_test.shape)

# Test function: Do not remove
assert XX[:,0].min() >= -2.5 and XX[:,0].max() <= 2.5, 'Does the XX is normalized?'
assert XX[:,1].min() >= -2.5 and XX[:,1].max() <= 2.5, 'Does the XX is normalized?'

print("success!")
# End Test function
```

```
(160, 2)
(160, 3)
(40, 2)
(40, 3)
success!
```

Exercise 1.7 (10 points)

define class for logistic regression: batch gradient descent

The class includes:

- **Sigmoid** function $$sigmoid(z) = \frac{1}{1+e^{-z}}$$
- **Softmax** function $$softmax(z) = \frac{e^{z_i}}{\sum_n{e^z}}$$
- **Hyperthesis (h)** function $$\hat{y} = h(X;\theta) = softmax(\theta . X)$$
- **Gradient (Negative likelihood)** function $$gradient = - X . \frac{y-\hat{y}}{n}$$
- **Cost** function $$cost = \frac{\sum{((-y\log{\hat{y}}) - ((1-y)\log{(1 - \hat{y})}))}}{n}$$
- **Gradient ascent** function
- **Prediction** function
- **Get accuracy** funciton

In [20]:

```python
class Logistic_BGD:
    def __init__(self):
        pass

    def sigmoid(self,z):
        s = 1 / (1 + np.exp(-z))
        # YOUR CODE HERE
        #raise NotImplementedError()
        return s

    def softmax(self, z):
        sm = np.exp(z)/np.sum(np.exp(z))
        # YOUR CODE HERE
        #raise NotImplementedError()
        return sm

    def h(self,X, theta):
        hf = self.sigmoid(X @ theta)
        # YOUR CODE HERE
        #raise NotImplementedError()
        return hf

    def gradient(self, X, y, y_pred):
        grad = X.T.dot(y_pred - y) / X.shape[0]
```

```
        # YOUR CODE HERE
        #raise NotImplementedError()
        return grad

    def costFunc(self, theta, X, y):
        y_pred = self.h(X, theta)
        error = (y * np.log(y_pred)) + ((1 - y) * np.log(1 - y_pred))
        cost = -sum(error) / X.shape[0]
        grad = self.gradient(X, y, y_pred)
        # YOUR CODE HERE
        #raise NotImplementedError()
        return cost, grad

    def gradientAscent(self, X, y, theta, alpha, num_iters):
        m = len(y)
        J_history = []
        theta_history = []
        for i in range(num_iters):
            # 1. calculate cost, grad function
            cost, grad = self.costFunc(theta, X, y)
            # 2. update new theta
            #theta = None
            theta = theta - alpha * grad
            # YOUR CODE HERE
            #raise NotImplementedError()


            J_history.append(cost)
            theta_history.append(theta)
        J_min_index = np.argmin(J_history)
        print("Minimum at iteration:",J_min_index)
        return theta_history[J_min_index] , J_history

    def predict(self,X, theta):
        labels=[]
        # 1. take y_predict from hyperthesis function
        y_pred = self.h(X, theta)
        # 2. classify y_predict that what it should be class1 or class2
        for i in range(len(y_pred)):
            if y_pred[i] >= 0.5:
                labels.append(1)
            else:
                labels.append(0)
        # 3. append the output from prediction
        # YOUR CODE HERE
        #raise NotImplementedError()

        labels=np.asarray(labels)
        return labels

    def getAccuracy(self,X,y,theta):
        y_pred = self.predict(X, theta)
        y_pred = y_pred.reshape(y.shape)
        percent_correct = 100*np.sum(y == y_pred).astype(int) / y.shape[0]
        # YOUR CODE HERE
        #raise NotImplementedError()
        return percent_correct
```

In [21]:

```
# Test function: Do not remove
```

```
lbgd = Logistic_BGD()
test_x = np.array([[1,2,3,4,5]]).T
out_x1 = lbgd.sigmoid(test_x)
out_x2 = lbgd.sigmoid(test_x.T)
print('out_x1', out_x1.T)
assert np.array_equal(np.round(out_x1.T, 5), np.round([[0.73105858, 0.88079708, 0.95257
assert np.array_equal(np.round(out_x2, 5), np.round([[0.73105858, 0.88079708, 0.9525741
out_x1 = lbgd.softmax(out_x1)
out_x2 = lbgd.softmax(out_x2)
print('out_x1', out_x1.T)
assert np.array_equal(np.round(out_x1.T, 5), np.round([[0.16681682, 0.19376282, 0.20818
assert np.array_equal(np.round(out_x2, 5), np.round([[0.16681682, 0.19376282, 0.2081818
test_t = np.array([[0.3, 0.2]]).T
test_x = np.array([[1,2,3,4,5, 6], [2, 9, 4, 3, 1, 0]]).T
test_y = np.array([[0,1,0,1,0,1]]).T
test_y_p = lbgd.h(test_x, test_t)
print('test_y_p', test_y_p.T)
assert np.array_equal(np.round(test_y_p.T, 5), np.round([[0.66818777, 0.9168273, 0.8455
test_g = lbgd.gradient(test_x, test_y, test_y_p)
print('test_g', test_g.T)
assert np.array_equal(np.round(test_g.T, 5), np.round([[0.9746016, 0.73165696]], 5)), "
test_c, test_g = lbgd.costFunc(test_t, test_x, test_y)
print('test_c', test_c.T)
assert np.round(test_c, 5) == np.round(0.87192491, 5), "costFunc function is incorrect"
test_t_out , test_j = lbgd.gradientAscent(test_x, test_y, test_t, 0.001, 3)
print('test_t_out', test_t_out.T)
print('test_j', test_j)
assert np.array_equal(np.round(test_t_out.T, 5), np.round([[0.29708373, 0.19781153]], 5
assert np.round(test_j[2], 5) == np.round(0.86896665, 5), "gradientAscent function is i
test_l = lbgd.predict(test_x, test_t)
print('test_l', test_l)
assert np.array_equal(np.round(test_l, 1), np.round([1,1,1,1,1,1], 1)), "gradientAscent
test_a = lbgd.getAccuracy(test_x,test_y,test_t)
print('test_a', test_a)
assert np.round(test_a, 1) == 50.0, "getAccuracy function is incorrect"

print("success!")
# End Test function
```

```
out_x1 [[0.73105858 0.88079708 0.95257413 0.98201379 0.99330715]]
out_x1 [[0.16681682 0.19376282 0.20818183 0.21440174 0.21683678]]
test_y_p [[0.66818777 0.9168273  0.84553473 0.85814894 0.84553473 0.85814894]]
test_g [[0.9746016  0.73165696]]
test_c [0.87192491]
Minimum at iteration: 2
test_t_out [[0.29708373 0.19781153]]
test_j [array([0.87192491]), array([0.87044176]), array([0.86896665])]
test_l [1 1 1 1 1 1]
test_a 50.0
success!
```

Expected result:\ out_x1 [[0.73105858 0.88079708 0.95257413 0.98201379 0.99330715]]\ out_x1 [[0.16681682 0.19376282 0.20818183 0.21440174 0.21683678]]\ test_y_p [[0.66818777 0.9168273 0.84553473 0.85814894 0.84553473 0.85814894]]\ test_g [[0.9746016 0.73165696]]\ test_c [0.87192491]\ Minimum at iteration: 2\ test_t_out [[0.29708373 0.19781153]]\ test_j [array([0.87192491]), array([0.87044176]), array([0.86896665])]\ test_l [1 1 1 1 1 1]\ test_a 50.0

Exercise 1.8 (5 points)

Training the data using Logistic_BGD class.

- Input: X_design_train
- Output: y_train
- Use 50,000 iterations

Find the initial_theta yourself

In [22]:

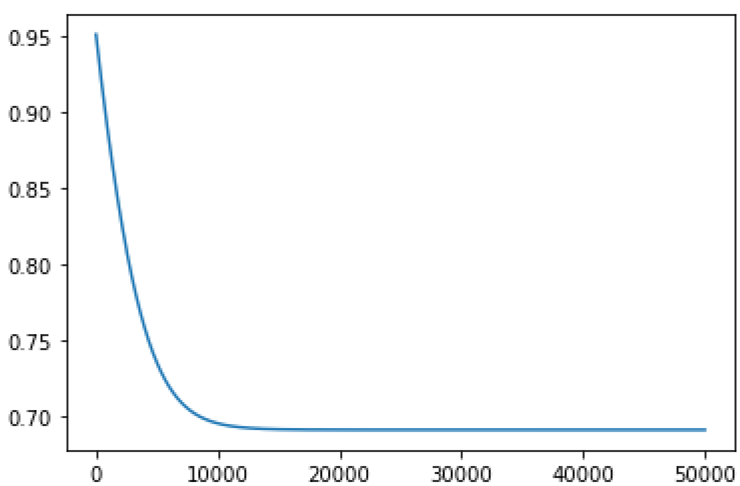
```
alpha = 0.001
iterations = 50000

BGD_model = Logistic_BGD()

initial_theta = np.ones((X_design_train.shape[1],1))
bgd_theta, bgd_cost = BGD_model.gradientAscent(X_design_train, y_train, initial_theta,

# YOUR CODE HERE
#raise NotImplementedError()
```

```
Minimum at iteration: 49999
```

In [23]:

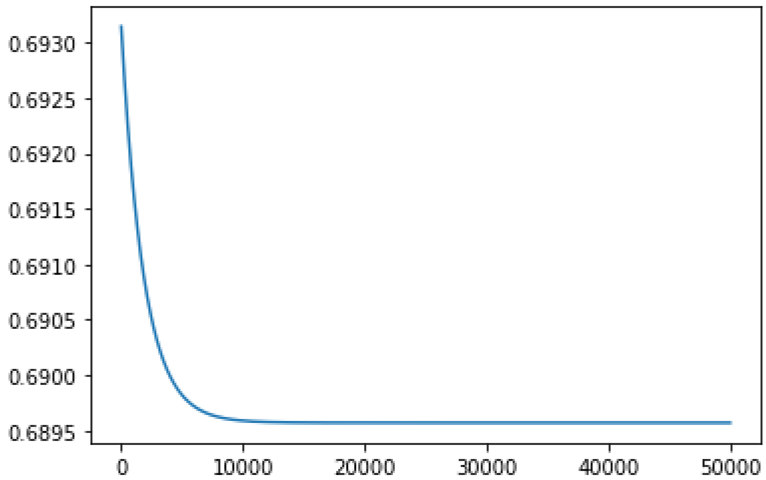
```
print(bgd_theta)
print(len(bgd_cost))

print(bgd_cost[0])
plt.plot(bgd_cost)
plt.show()

# Test function: Do not remove
assert bgd_theta.shape == (X_train.shape[1] + 1,1) or bgd_theta.shape == (X_train.shape
assert len(bgd_cost) == iterations, "cost data size is incorrect"

print("success!")
# End Test function
```

```
[[0.0961404 ]
 [0.04787377]
 [0.03641186]]
50000
[0.95098098]
```




```
success!
```

Expected result (or look alike):\ [[-0.07328673]\ [-0.13632896]\ [0.05430939]]\ 50000



In lab exercises

1. Verify that the gradient descent solution is correct. Plot the optimal decision boundary you obtain.
2. Write a new class that uses Newton's method for the optmization rather than simple gradient descent.
3. Verify that you obtain a similar solution with Newton's method. Plot the optimal decision boundary you obtain.
4. Compare the number of iterations required for gradient descent vs. Newton's method. Do you observe other issues with Newton's method such as a singular or nearly singular Hessian matrix?

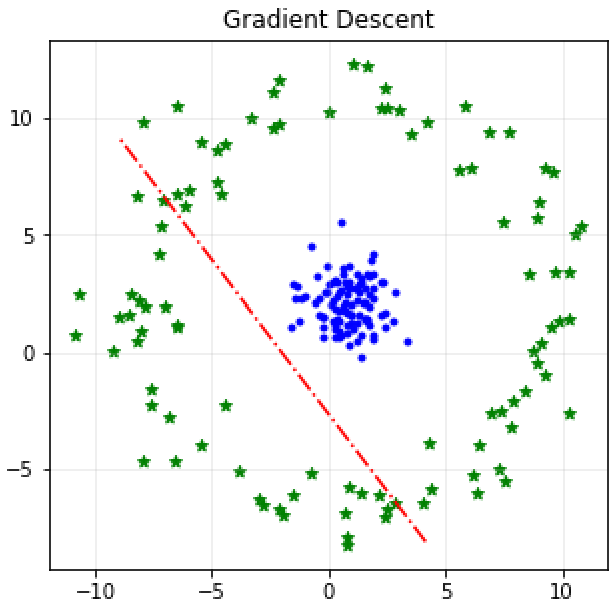
Exercise 1.9 (5 points)

Plot the optimal decision boundary of gradient ascent

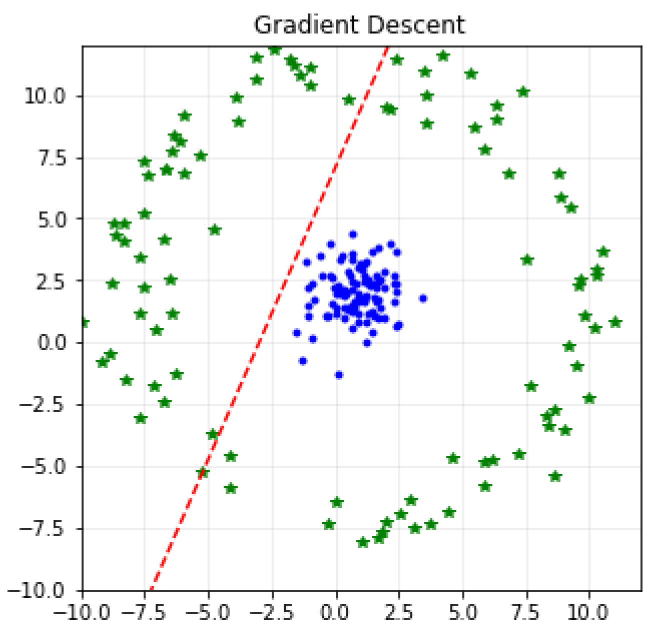
In [24]:
```
def boundary_points(X, theta):

    v_orthogonal = np.array([[theta[1,0]],[theta[2,0]]])
    v_ortho_length = np.sqrt(v_orthogonal.T @ v_orthogonal)
    dist_ortho = theta[0,0] / v_ortho_length
    v_orthogonal = v_orthogonal / v_ortho_length
    v_parallel = np.array([[-v_orthogonal[1,0]],[v_orthogonal[0,0]]])
    projections = X @ v_parallel
    proj_1 = min(projections)
    proj_2 = max(projections)
    point_1 = proj_1 * v_parallel - dist_ortho * v_orthogonal
    point_2 = proj_2 * v_parallel - dist_ortho * v_orthogonal
    return point_1, point_2
```

In [25]:
```
fig1 = plt.figure(figsize=(5,5))
```

```
ax = plt.axes()
ax.set_aspect(aspect = 'equal', adjustable = 'box')
plt.title('Gradient Descent')
plt.grid(axis='both', alpha=.25)
plt.scatter(X1[:,0], X1[:,1], c='b', marker='.', label='class 1')
plt.scatter(X2[:,0], X2[:,1], c='g', marker='*', label='class 2')

point_1, point_2 = boundary_points(X, bgd_theta)
plt.plot([point_1[0,0], point_2[0,0]], [point_1[1,0], point_2[1,0]], 'r-.')
plt.show()
```



Expected result (or look alike):\



In [26]:

```
print("Accuracy =",BGD_model.getAccuracy(X_design_test,y_test,bgd_theta))
```

```
Accuracy = 22.5
```

Exercise 2.1 (10 points)

Write Newton's method class

In [27]:

```
class Logistic_NM: #logistic regression for newton's method

    def __init__(self):
        pass

    def sigmoid(self,z):
        #s = None
        # YOUR CODE HERE
        s = 1 / (1 + np.exp(-1*z))
        #raise NotImplementedError()
        return s

    def h(self,X, theta):
        #hf = None
        # YOUR CODE HERE
        hf = self.sigmoid(X @ theta)
        #raise NotImplementedError()
        return hf

    def gradient(self, X, y, y_pred):
        #grad = None
        # YOUR CODE HERE
        m = len(y)
        grad = 1/m * np.dot(X.T,(y_pred - y))
        #raise NotImplementedError()
        return grad

    def hessian(self, X, y, theta):
        #hess_mat = None
        # YOUR CODE HERE
        y_hat = self.h(X, theta)
        X2 = X.T @ X
        Y2 = y_hat.T @ (1 - y_hat)
        value = Y2[0,0]
        hess_mat = X2 * value / X.shape[0]
        #raise NotImplementedError()
        return hess_mat

    def costFunc(self, theta, X, y):
        #cost, grad = None, None
        # YOUR CODE HERE
        m = len(y)
        y_pred = self.h(X, theta)
        error = (y * np.log(y_pred)) + ((1-y)*np.log(1-y_pred))
        cost = -1/m * np.sum(error)
        grad = self.gradient(X, y, y_pred)
        return cost, grad

    def newtonsMethod(self, X, y, theta, num_iters):
        m = len(y)
        J_history = []
        theta_history = []
        for i in range(num_iters):
            hessian_mat = np.zeros((X.shape[1], X.shape[1]))

            hmat_xi = self.hessian(X,y, theta)
            hessian_mat += hmat_xi
            cost, grad = self.costFunc(theta, X,y)
            theta = theta - np.linalg.pinv(hessian_mat) @ grad
```

```
            J_history.append(cost)
            theta_history.append(theta)
        J_min_index = np.argmin(J_history)
        return theta_history[J_min_index] , J_history



    def predict(self,X, theta):
        labels=[]
        y_pred = self.h(X, theta)
        for i in range(len(y_pred)):
            if y_pred[i] >= 0.5:
                labels.append(1)
            else:
                labels.append(0)
        # YOUR CODE HERE
        #raise NotImplementedError()

        labels=np.asarray(labels)
        return labels

    def getAccuracy(self,X,y,theta):
        #percent_correct = None
        # YOUR CODE HERE
        y_pred = self.predict(X, theta)
        y_pred = y_pred.reshape(y.shape)
        percent_correct = 100*np.sum(y == y_pred).astype(int) / y.shape[0]
        #raise NotImplementedError()
        return percent_correct
```

In [28]:

```
# Test function: Do not remove
lbgd = Logistic_NM()
test_x = np.array([[1,2,3,4,5]]).T
out_x1 = lbgd.sigmoid(test_x)
out_x2 = lbgd.sigmoid(test_x.T)
print('out_x1', out_x1.T)
assert np.array_equal(np.round(out_x1.T, 5), np.round([[0.73105858, 0.88079708, 0.95257
assert np.array_equal(np.round(out_x2, 5), np.round([[0.73105858, 0.88079708, 0.9525741
test_t = np.array([[0.3, 0.2]]).T
test_x = np.array([[1,2,3,4,5, 6], [2, 9, 4, 3, 1, 0]]).T
test_y = np.array([[0,1,0,1,0,1]]).T
test_y_p = lbgd.h(test_x, test_t)
print('test_y_p', test_y_p.T)
assert np.array_equal(np.round(test_y_p.T, 5), np.round([[0.66818777, 0.9168273, 0.8455
test_g = lbgd.gradient(test_x, test_y, test_y_p)
print('test_g', test_g.T)
assert np.array_equal(np.round(test_g.T, 5), np.round([[0.9746016, 0.73165696]], 5)), "
test_h = lbgd.hessian(test_x, test_y, test_t)
print('test_h', test_h)
assert test_h.shape == (2, 2), "hessian matrix function is incorrect"
assert np.array_equal(np.round(test_h.T, 5), np.round([[12.17334371, 6.55487738],[ 6.55
test_c, test_g = lbgd.costFunc(test_t, test_x, test_y)
print('test_c', test_c.T)
assert np.round(test_c, 5) == np.round(0.87192491, 5), "costFunc function is incorrect"
test_t_out , test_j = lbgd.newtonsMethod(test_x, test_y, test_t, 3)
print('test_t_out', test_t_out.T)
print('test_j', test_j)
assert np.array_equal(np.round(test_t_out.T, 5), np.round([[0.14765747, 0.15607017]], 5
assert np.round(test_j[2], 5) == np.round(0.7534506190845247, 5), "newtonsMethod functi
```

```
test_l = lbgd.predict(test_x, test_t)
print('test_l', test_l)
assert np.array_equal(np.round(test_l, 1), np.round([1,1,1,1,1,1], 1)), "gradientAscent
test_a = lbgd.getAccuracy(test_x,test_y,test_t)
print('test_a', test_a)
assert np.round(test_a, 1) == 50.0, "getAccuracy function is incorrect"

print("success!")
# End Test function
```

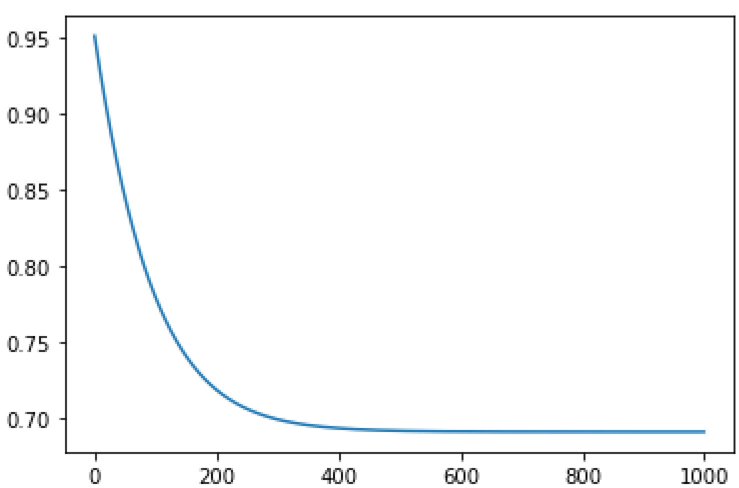
```
out_x1 [[0.73105858 0.88079708 0.95257413 0.98201379 0.99330715]]
test_y_p [[0.66818777 0.9168273  0.84553473 0.85814894 0.84553473 0.85814894]]
test_g [[0.9746016  0.73165696]]
test_h [[12.17334371  6.55487738]
 [ 6.55487738 14.84880387]]
test_c 0.8719249134773479
test_t_out [[0.14765747 0.15607017]]
test_j [0.8719249134773479, 0.7967484437157274, 0.7534506190845246]
test_l [1 1 1 1 1 1]
test_a 50.0
success!
```

Expect result: out_x1 [[0.73105858 0.88079708 0.95257413 0.98201379 0.99330715]]\ test_y_p [[0.66818777 0.9168273 0.84553473 0.85814894 0.84553473 0.85814894]]\ test_g [[0.9746016 0.73165696]]\ test_h [[12.17334371 6.55487738]\ [6.55487738 14.84880387]]\ test_c 0.8719249134773479\ Minimum at iteration: 2\ test_t_out [[0.14765747 0.15607017]]\ test_j [0.8719249134773479, 0.7967484437157274, 0.7534506190845247]\ test_l [1 1 1 1 1 1]\ test_a 50.0

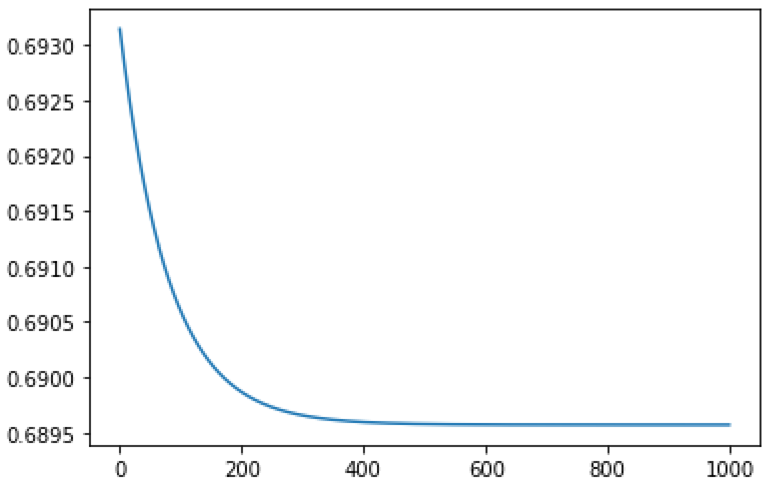
In [29]:

```
NM_model = Logistic_NM()

iterations = 1000

nm_theta, nm_cost = NM_model.newtonsMethod(X_design_train, y_train, initial_theta, iter
print("theta:",nm_theta)

print(nm_cost[0])
plt.plot(nm_cost)
plt.show()
```

```
theta: [[0.09785403]
 [0.04983972]
 [0.03835179]]
0.9509809835169094
```



Expected result (or look alike):\ Minimum at iteration: 999\ theta: [[-0.07313861]\ [-0.13605172]\ [0.05419746]]\ 0.6931471805599453

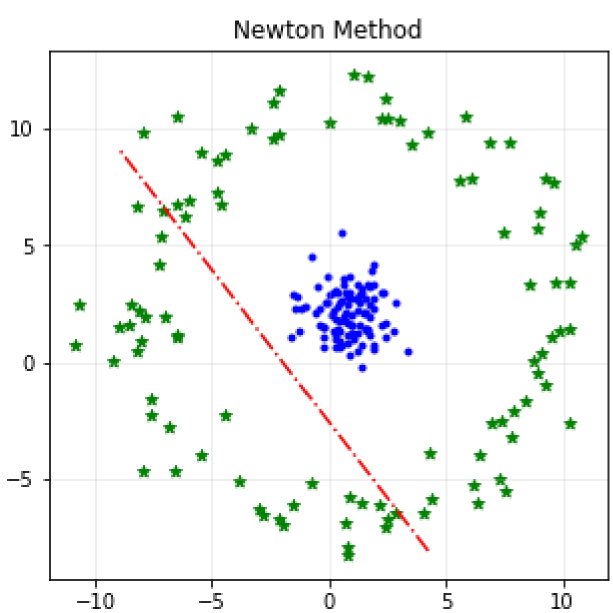


Exercise 2.2 (5 points)

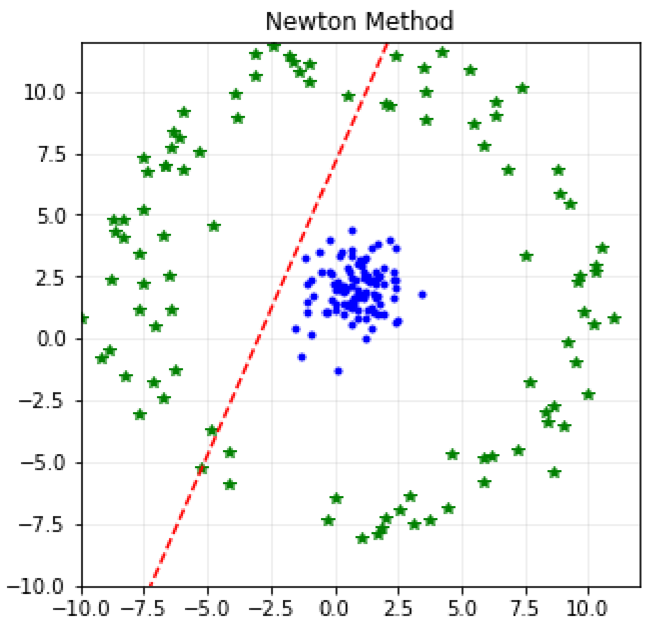
Plot the optimal decision boundary of Newton method

In [30]:

```
# YOUR CODE HERE
fig1 = plt.figure(figsize=(5,5))
ax = plt.axes()
ax.set_aspect(aspect = 'equal', adjustable = 'box')
plt.title('Newton Method')
plt.grid(axis='both', alpha=.25)
plt.scatter(X1[:,0], X1[:,1], c='b', marker='.', label='class 1')
plt.scatter(X2[:,0], X2[:,1], c='g', marker='*', label='class 2')

point_1, point_2 = boundary_points(X, nm_theta)
plt.plot([point_1[0,0], point_2[0,0]], [point_1[1,0], point_2[1,0]], 'r-.')
plt.show()
#raise NotImplementedError()
```



Expected result (or look alike):



In [31]:

```
print("Accuracy =",NM_model.getAccuracy(X_design_test,y_test,nm_theta))
```

```
Accuracy = 22.5
```

Exercise 2.3 (5 points)

Compare the number of iterations required for gradient descent vs. Newton's method. Do you observe other issues with Newton's method such as a singular or nearly singular Hessian matrix?

YOUR ANSWER HERE

Take-home exercises

1. Perform a *polar transformation* on the data above to obtain a linearly separable dataset. (5 points)
2. Verify that you obtain good classification accuracy for logistic regression with GD or Netwon's method after the polar transformation (10 points)
3. Apply Newton's method to the dataset you used for the take home exercises in Lab 03. (20 points)

polar coordinates is a mapping from a point (r,θ) in the polar coordinate plane to the correspoing point (x,y) in the Carteisan coordinate plane

$$r^2 = x^2 + y^2$$$$ x = rcos \theta$$$$ y = rsin \theta$$

In [32]:

```
df = pd.DataFrame(X, columns=['X0', 'X1'])
df['angles'] = np.arctan(df.X1 / df.X0)
df['radius'] = np.sqrt(df.X0 ** 2 + df.X1 ** 2)
df['y'] = y
```

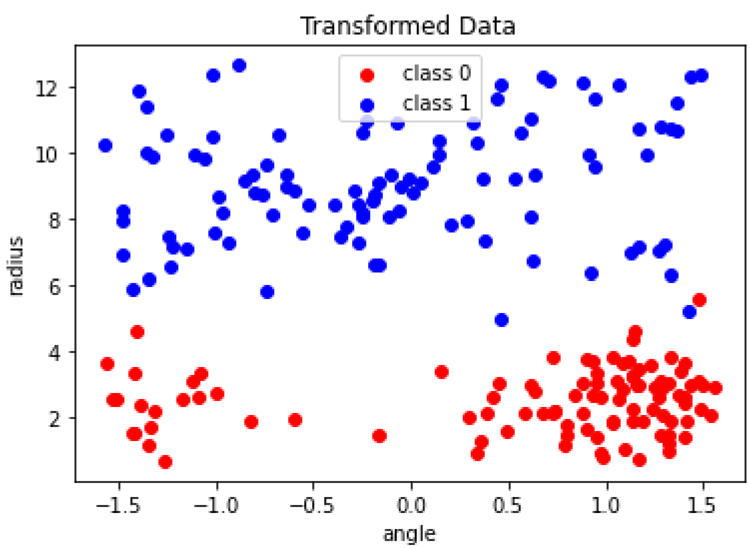
```
newX = df[['angles', 'radius']].values
newX = np.insert(newX, 0, 1, axis=1)
X_train = newX[idx_train]
X_test = newX[idx_test]
y_train = y[idx_train]
y_test = y[idx_test]
```

In [33]:

```
y0_df = df[df.y == 0]
y1_df = df[df.y == 1]

plt.title('Transformed Data')
plt.scatter(y0_df.angles, y0_df.radius, c='r', label='class 0')
plt.scatter(y1_df.angles, y1_df.radius, c='b', label='class 1')
plt.legend()
plt.xlabel('angle')
plt.ylabel('radius')
```

Out[33]:

```
Text(0, 0.5, 'radius')
```



In [34]:

```
alpha = 0.001
iterations = 50000

BGD_model = Logistic_BGD()

initial_theta = np.ones((X_train.shape[1],1))
bgd_theta_polar, bgd_cost = BGD_model.gradientAscent(X_train, y_train, initial_theta, a
print('theta: ', bgd_theta_polar)
```

```
Minimum at iteration: 49999
theta:  [[-3.61385119]
 [-0.6266392 ]
 [ 0.77783254]]
```

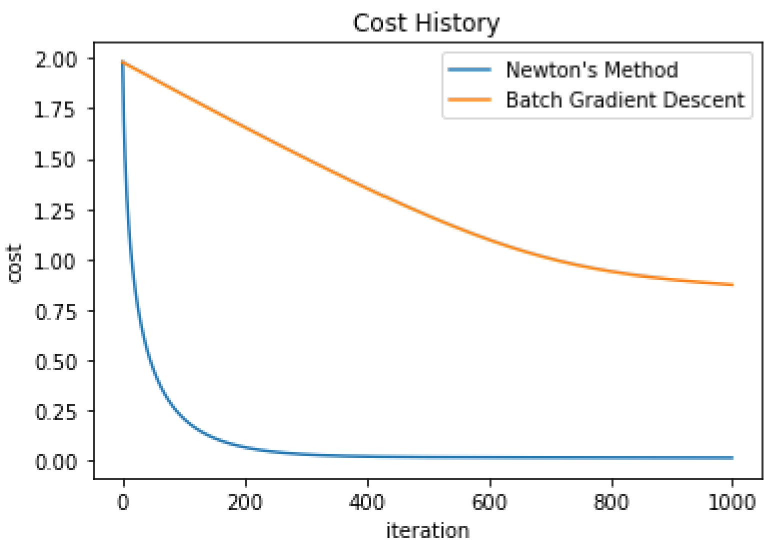
In [35]:

```
NM_model = Logistic_NM()

iterations = 1000

nm_theta_polar, nm_cost = NM_model.newtonsMethod(X_train, y_train, initial_theta, itera
print("theta:",nm_theta_polar)
```

```
theta: [[-12.18409525]
 [ -0.12961706]
 [  2.36698771]]
```

In [36]:

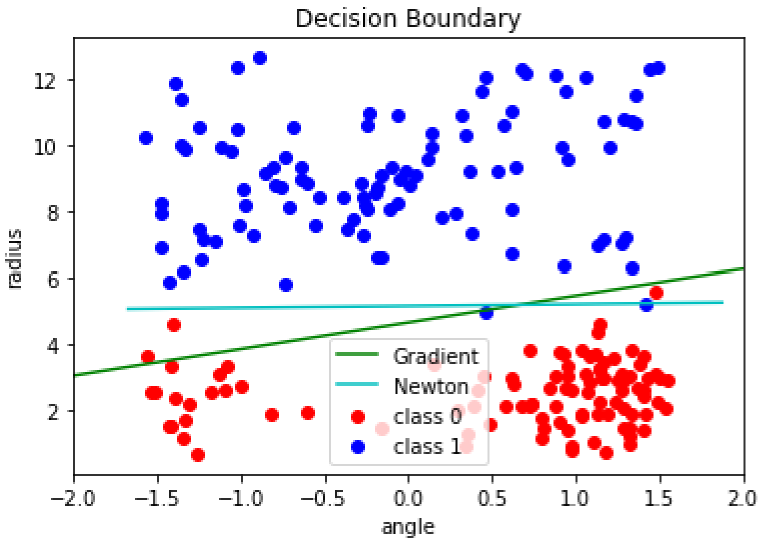
```
plt.plot(nm_cost, label='Newton\'s Method')
plt.plot(bgd_cost[:iterations], label='Batch Gradient Descent')
plt.title('Cost History')
plt.xlabel('iteration')
plt.ylabel('cost')
plt.legend()
plt.show()

print('Minimum Cost for each method from polar transformation')
print('Gradient Descent:', np.min(bgd_cost))
print('Newton\'s Method :', np.min(nm_cost))
```




```
Minimum Cost for each method from polar transformation
Gradient Descent: 0.11419681571541887
Newton's Method : 0.015572427864877987
```

In [37]:

```
y0_df = df[df.y == 0]
y1_df = df[df.y == 1]

point_1, point_2 = boundary_points(newX[:,1:], bgd_theta_polar)
point_1n, point_2n = boundary_points(newX[:,1:], nm_theta_polar)

plt.title('Decision Boundary')
plt.scatter(y0_df.angles, y0_df.radius, c='r', label='class 0')
plt.scatter(y1_df.angles, y1_df.radius, c='b', label='class 1')
plt.legend()
plt.xlabel('angle')
plt.ylabel('radius')
plt.plot([point_1[0,0], point_2[0,0]],[point_1[1,0], point_2[1,0]], 'g-', label='Gradie
plt.plot([point_1n[0,0], point_2n[0,0]],[point_1n[1,0], point_2n[1,0]], 'c-', label='Ne
plt.legend(loc='best')
plt.xlim(-2,2)
plt.show()
```



In [38]:
```
g_acc = BGD_model.getAccuracy(X_train, y_train, bgd_theta_polar)
n_acc = NM_model.getAccuracy(X_train, y_train, nm_theta_polar)

print("Train accuracy for polar transformation")
print('Gradient Accuracy : ', g_acc)
print('Newton Accuracy : ', n_acc)

g_acc = BGD_model.getAccuracy(X_test, y_test, bgd_theta_polar)
n_acc = NM_model.getAccuracy(X_test, y_test, nm_theta_polar)

print("Test accuracy for polar transformation")
print('Gradient Accuracy : ', g_acc)
print('Newton Accuracy : ', n_acc)
```

```
Train accuracy for polar transformation
Gradient Accuracy :  98.125
Newton Accuracy :  99.375
Test accuracy for polar transformation
Gradient Accuracy :  97.5
Newton Accuracy :  95.0
```

The report

Write a brief report covering your experiments (both in lab and take home) and submit the Jupyter notebook via JupyterHub at https://puffer.cs.ait.ac.th before the next lab.

In your solution, be sure to follow instructions!

In [39]:
```
# Import Pandas. You may need to run "pip3 install pandas" at the console if it's not a

import pandas as pd

# Import the data

data_train = pd.read_csv('train_LoanPrediction.csv')
data_test = pd.read_csv('test_LoanPrediction.csv')

# Start to explore the data
```

```
print('Training data shape', data_train.shape)
print('Test data shape', data_test.shape)

print('Training data:\n', data_train)
```

```
Training data shape (614, 13)
Test data shape (367, 12)
Training data:
       Loan_ID  Gender Married Dependents     Education Self_Employed  \
0    LP001002    Male      No          0      Graduate            No
1    LP001003    Male     Yes          1      Graduate            No
2    LP001005    Male     Yes          0      Graduate           Yes
3    LP001006    Male     Yes          0  Not Graduate            No
4    LP001008    Male      No          0      Graduate            No
..        ...     ...     ...        ...           ...           ...
609  LP002978  Female      No          0      Graduate            No
610  LP002979    Male     Yes         3+      Graduate            No
611  LP002983    Male     Yes          1      Graduate            No
612  LP002984    Male     Yes          2      Graduate            No
613  LP002990  Female      No          0      Graduate           Yes

     ApplicantIncome  CoapplicantIncome  LoanAmount  Loan_Amount_Term  \
0               5849                0.0         NaN             360.0
1               4583             1508.0       128.0             360.0
2               3000                0.0        66.0             360.0
3               2583             2358.0       120.0             360.0
4               6000                0.0       141.0             360.0
..               ...                ...         ...               ...
609             2900                0.0        71.0             360.0
610             4106                0.0        40.0             180.0
611             8072              240.0       253.0             360.0
612             7583                0.0       187.0             360.0
613             4583                0.0       133.0             360.0

     Credit_History Property_Area Loan_Status
0               1.0         Urban           Y
1               1.0         Rural           N
2               1.0         Urban           Y
3               1.0         Urban           Y
4               1.0         Urban           Y
..              ...           ...         ...
609             1.0         Rural           Y
610             1.0         Rural           Y
611             1.0         Urban           Y
612             1.0         Urban           Y
613             0.0     Semiurban           N

[614 rows x 13 columns]
```

In [40]:

```
# Check for missing values in the training and test data

print('Missing values for train data:\n------------------------\n', data_train.isnull()
print('Missing values for test data \n ------------------------\n', data_test.isnull().
```

```
Missing values for train data:
------------------------
 Loan_ID              0
Gender              13
Married              3
Dependents          15
```

```
Education            0
Self_Employed       32
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount          22
Loan_Amount_Term    14
Credit_History      50
Property_Area        0
Loan_Status          0
dtype: int64
Missing values for test data
 ------------------------
 Loan_ID             0
Gender              11
Married              0
Dependents          10
Education            0
Self_Employed       23
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           5
Loan_Amount_Term     6
Credit_History      29
Property_Area        0
dtype: int64
```

In [41]:

```python
# Compute ratio of each category value
# Divide the missing values based on ratio
# Fillin the missing values
# Print the values before and after filling the missing values for confirmation

print(data_train['Married'].value_counts())

married = data_train['Married'].value_counts()
print('Elements in Married variable', married.shape)
print('Married ratio ', married[0]/sum(married.values))

def fill_martial_status(data, yes_num_train, no_num_train):
    data['Married'].fillna('Yes', inplace = True, limit = yes_num_train)
    data['Married'].fillna('No', inplace = True, limit = no_num_train)

fill_martial_status(data_train, 2, 1)
print(data_train['Married'].value_counts())
print('Missing values for train data:\n------------------------\n', data_train.isnull()
```

```
Yes    398
No     213
Name: Married, dtype: int64
Elements in Married variable (2,)
Married ratio  0.6513911620294599
Yes    400
No     214
Name: Married, dtype: int64
Missing values for train data:
------------------------
 Loan_ID             0
Gender              13
Married              0
Dependents          15
Education            0
```

```
Self_Employed        32
ApplicantIncome       0
CoapplicantIncome     0
LoanAmount           22
Loan_Amount_Term     14
Credit_History       50
Property_Area         0
Loan_Status           0
dtype: int64
```

In [42]:

```
print(data_train['Dependents'].value_counts())
dependent = data_train['Dependents'].value_counts()

print('Dependent ratio 1 ', dependent['0'] / sum(dependent.values))
print('Dependent ratio 2 ', dependent['1'] / sum(dependent.values))
print('Dependent ratio 3 ', dependent['2'] / sum(dependent.values))
print('Dependent ratio 3+ ', dependent['3+'] / sum(dependent.values))

def fill_dependent_status(num_0_train, num_1_train, num_2_train, num_3_train, num_0_tes
    data_train['Dependents'].fillna('0', inplace=True, limit = num_0_train)
    data_train['Dependents'].fillna('1', inplace=True, limit = num_1_train)
    data_train['Dependents'].fillna('2', inplace=True, limit = num_2_train)
    data_train['Dependents'].fillna('3+', inplace=True, limit = num_3_train)
    data_test['Dependents'].fillna('0', inplace=True, limit = num_0_test)
    data_test['Dependents'].fillna('1', inplace=True, limit = num_1_test)
    data_test['Dependents'].fillna('2', inplace=True, limit = num_2_test)
    data_test['Dependents'].fillna('3+', inplace=True, limit = num_3_test)

fill_dependent_status(9, 2, 2, 2, 5, 2, 2, 1)

print(data_train['Dependents'].value_counts())

# Convert category value "3+" to "4"

data_train['Dependents'].replace('3+', 4, inplace = True)
data_test['Dependents'].replace('3+', 4, inplace = True)
```

```
0     345
1     102
2     101
3+     51
Name: Dependents, dtype: int64
Dependent ratio 1  0.5759599332220368
Dependent ratio 2  0.17028380634390652
Dependent ratio 3  0.1686143572621035
Dependent ratio 3+  0.08514190317195326
0     354
1     104
2     103
3+     53
Name: Dependents, dtype: int64
```

In [43]:

```
print(data_train['LoanAmount'].value_counts())

LoanAmt = data_train['LoanAmount'].value_counts()

print('mean loan amount ', np.mean(data_train["LoanAmount"]))

loan_amount_mean = np.mean(data_train["LoanAmount"])
```

```
data_train['LoanAmount'].fillna(loan_amount_mean, inplace=True, limit = 22)
data_test['LoanAmount'].fillna(loan_amount_mean, inplace=True, limit = 5)
```

```
120.0    20
110.0    17
100.0    15
187.0    12
160.0    12
         ..
570.0     1
300.0     1
376.0     1
117.0     1
311.0     1
Name: LoanAmount, Length: 203, dtype: int64
mean loan amount  146.41216216216216
```

In [44]:

```
print('Missing values for train data:\n------------------------\n', data_train.isnull()
print('Missing values for test data \n ------------------------\n', data_test.isnull().
```

```
Missing values for train data:
------------------------
 Loan_ID              0
Gender              13
Married              0
Dependents           0
Education            0
Self_Employed       32
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term    14
Credit_History      50
Property_Area        0
Loan_Status          0
dtype: int64
Missing values for test data
 ------------------------
 Loan_ID              0
Gender              11
Married              0
Dependents           0
Education            0
Self_Employed       23
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     6
Credit_History      29
Property_Area        0
dtype: int64
```

In [45]:

```
print(data_train['Gender'].value_counts())
gender = data_train['Gender'].value_counts()
print('Elements in Gender variable', gender.shape)
Male_ratio = gender[0]/sum(gender.values)
Female_ratio = gender[1]/sum(gender.values)
print('Male ratio ', Male_ratio)
print('Female ratio ', Female_ratio)
```

```python
def fill_gender_status(num_male_train, num_female_trian, num_male_test, num_female_test
    data_train['Gender'].fillna('Male', inplace = True, limit = num_male_train)
    data_train['Gender'].fillna('Female', inplace = True, limit = num_female_train)
    data_test['Gender'].fillna('Male', inplace = True, limit = num_male_test)
    data_test['Gender'].fillna('Female', inplace = True, limit = num_female_test)

num_male_train = round(Male_ratio * data_train['Gender'].isnull().sum())
num_female_train = round(Female_ratio * data_train['Gender'].isnull().sum())
num_male_test = round(Male_ratio * data_test['Gender'].isnull().sum())
num_female_test = round(Female_ratio * data_test['Gender'].isnull().sum())
fill_gender_status(num_male_train, num_female_train, num_male_test, num_female_test)

print(data_train['Gender'].value_counts())
print('Missing values for train data:\n------------------------\n', data_train.isnull()
print('Missing values for test data:\n------------------------\n', data_test.isnull().s
```

```
Male      489
Female    112
Name: Gender, dtype: int64
Elements in Gender variable (2,)
Male ratio  0.8136439267886856
Female ratio  0.18635607321131448
Male      500
Female    114
Name: Gender, dtype: int64
Missing values for train data:
------------------------
 Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed       32
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term    14
Credit_History      50
Property_Area        0
Loan_Status          0
dtype: int64
Missing values for test data:
------------------------
 Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed       23
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     6
Credit_History      29
Property_Area        0
dtype: int64
```

In [46]:

```python
print(data_train['Self_Employed'].value_counts())
```

```
S_E = data_train['Self_Employed'].value_counts()
no_ratio = S_E[0]/sum(S_E.values)
yes_ratio = S_E[1]/sum(S_E.values)
print("Elements in Self-Employed variable ", S_E.shape)
print("No ratio ", no_ratio)
print("yes ratio ", yes_ratio)

def fill_selfemployed_status(num_no_train, num_yes_train, num_no_test, num_yes_test):
    data_train['Self_Employed'].fillna('No', inplace = True, limit = num_no_train)
    data_train['Self_Employed'].fillna('Yes', inplace = True, limit = num_yes_train)
    data_test['Self_Employed'].fillna('No', inplace = True, limit = num_no_test)
    data_test['Self_Employed'].fillna('Yes', inplace = True, limit = num_yes_test)

num_no_train = round(no_ratio * data_train['Self_Employed'].isnull().sum())
num_yes_train = round(yes_ratio * data_train['Self_Employed'].isnull().sum())
num_no_test = round(no_ratio * data_test['Self_Employed'].isnull().sum())
num_yes_test = round(yes_ratio * data_test['Self_Employed'].isnull().sum())
fill_selfemployed_status(num_no_train, num_yes_train, num_no_test, num_yes_test)

print(data_train['Self_Employed'].value_counts())
print('Missing values for train data:\n------------------------\n', data_train.isnull()
print('Missing values for test data:\n------------------------\n', data_test.isnull().s
```

```
No     500
Yes     82
Name: Self_Employed, dtype: int64
Elements in Self-Employed variable  (2,)
No ratio  0.8591065292096219
yes ratio  0.140893470790378
No     527
Yes     87
Name: Self_Employed, dtype: int64
Missing values for train data:
------------------------
 Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term    14
Credit_History      50
Property_Area        0
Loan_Status          0
dtype: int64
Missing values for test data:
------------------------
 Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     6
```

```
Credit_History       29
Property_Area         0
dtype: int64
```

In [47]:

```
print(data_train['Loan_Amount_Term'].value_counts())

LoanAT = data_train['Loan_Amount_Term'].value_counts()

print('mean loan amount term ', np.mean(data_train['Loan_Amount_Term']))

LoanAT_mean = np.mean(data_train['Loan_Amount_Term'])

data_train['Loan_Amount_Term'].fillna(LoanAT_mean, inplace = True, limit = data_train['
data_test['Loan_Amount_Term'].fillna(LoanAT_mean, inplace = True, limit = data_test['Lo


print(data_train['Loan_Amount_Term'].value_counts())
print('Missing values for train data:\n------------------------\n', data_train.isnull()
print('Missing values for test data:\n------------------------\n', data_test.isnull().s
```

```
360.0    512
180.0     44
480.0     15
300.0     13
84.0       4
240.0      4
120.0      3
36.0       2
60.0       2
12.0       1
Name: Loan_Amount_Term, dtype: int64
mean loan amount term  342.0
360.0    512
180.0     44
480.0     15
342.0     14
300.0     13
84.0       4
240.0      4
120.0      3
36.0       2
60.0       2
12.0       1
Name: Loan_Amount_Term, dtype: int64
Missing values for train data:
------------------------
 Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History      50
Property_Area        0
Loan_Status          0
dtype: int64
```

```
Missing values for test data:
------------------------
 Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History      29
Property_Area        0
dtype: int64
```

In [48]:

```
print(data_train['Credit_History'].value_counts())

CH = data_train['Credit_History'].value_counts()

ratio_1 = CH[0] / sum(CH.values)
ratio_0 = CH[1] / sum(CH.values)

print('Elements is Credit_History variable ', CH.shape)
print("ratio 1.0 : ", ratio_1)
print("ratio 0.0 : ", ratio_0)

def fill_creditH_status(num_1_train, num_0_train, num_1_test, num_0_test):
    data_train['Credit_History'].fillna(1.0, inplace = True, limit = num_1_train)
    data_train['Credit_History'].fillna(0.0, inplace = True, limit = num_0_train)
    data_test['Credit_History'].fillna(1.0, inplace = True, limit = num_1_test)
    data_test['Credit_History'].fillna(0.0, inplace = True, limit = num_0_test)

num_1_train = round(ratio_1 * data_train['Credit_History'].isnull().sum())
num_0_train = round(ratio_0 * data_train['Credit_History'].isnull().sum())
num_1_test = round(ratio_1 * data_test['Credit_History'].isnull().sum())
num_0_test = round(ratio_0 * data_test['Credit_History'].isnull().sum())

fill_creditH_status(num_1_train, num_0_train, num_1_test, num_0_test)

print(data_train['Credit_History'].value_counts())
print('Missing values for train data:\n------------------------\n', data_train.isnull()
print('Missing values for test data:\n------------------------\n', data_test.isnull().s
```

```
1.0    475
0.0     89
Name: Credit_History, dtype: int64
Elements is Credit_History variable  (2,)
ratio 1.0 :  0.15780141843971632
ratio 0.0 :  0.8421985815602837
1.0    483
0.0    131
Name: Credit_History, dtype: int64
Missing values for train data:
------------------------
 Loan_ID             0
Gender              0
Married             0
Dependents          0
```

```
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History       0
Property_Area        0
Loan_Status          0
dtype: int64
Missing values for test data:
------------------------
 Loan_ID              0
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History       0
Property_Area        0
dtype: int64
```

In [49]:

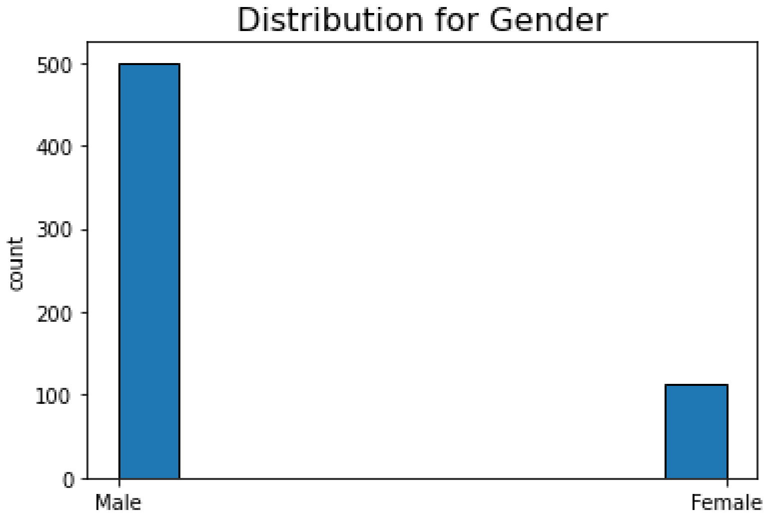
```
data_train.drop(columns=['Loan_ID'], inplace = True)
data_test.drop(columns=['Loan_ID'], inplace = True)
```

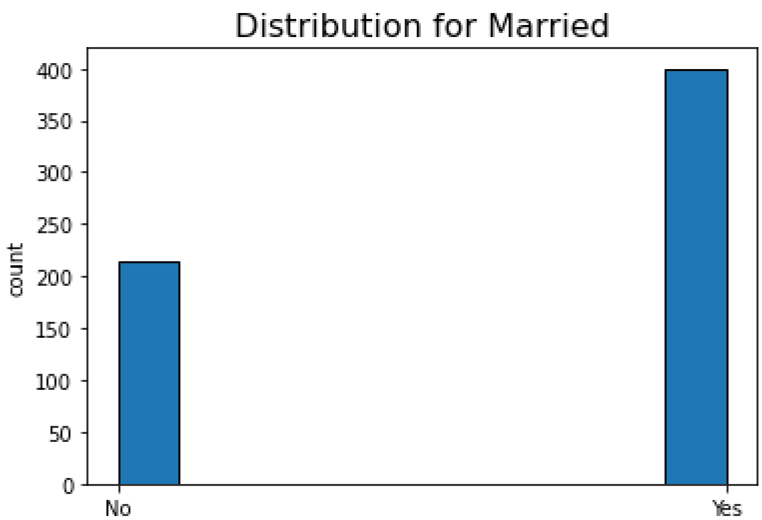
In [50]:

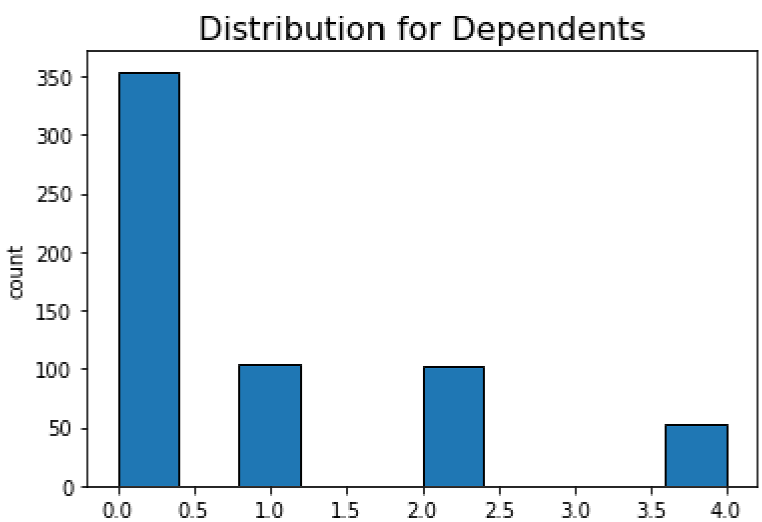
```
data_train['Dependents'] = data_train['Dependents'].astype(int)
data_test['Dependents'] = data_test['Dependents'].astype(int)
```

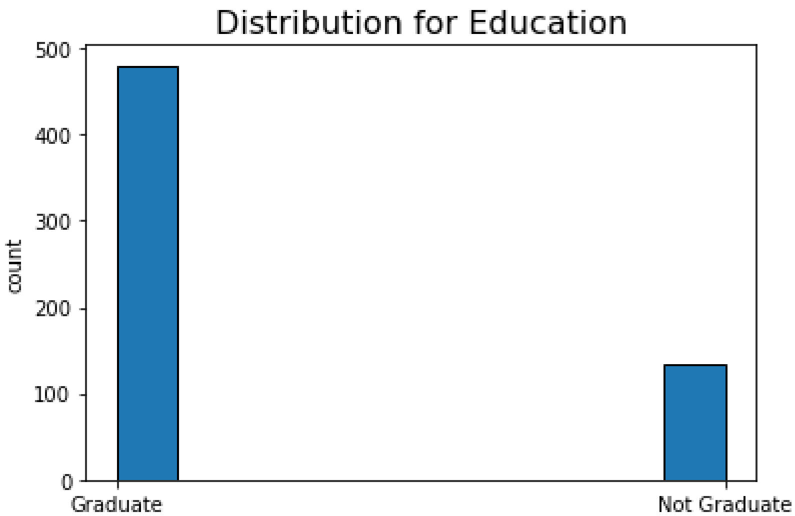
In [51]:

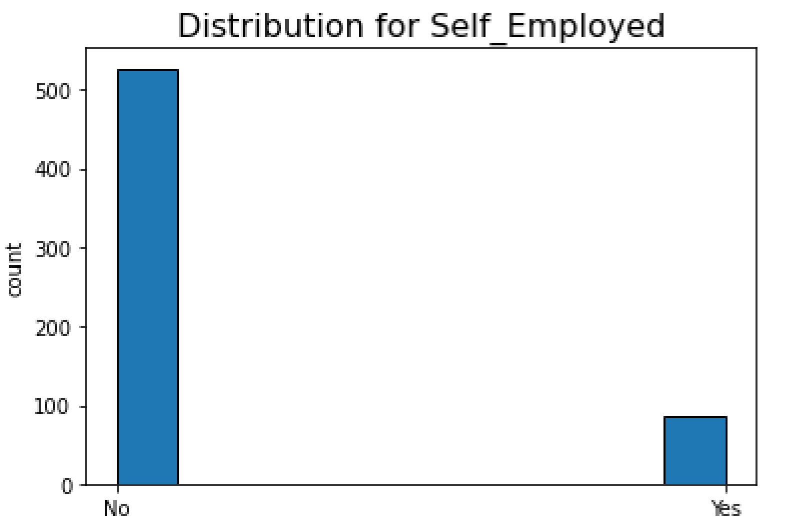
```
column_names = list(data_train.columns)

def hisplot(data, name):
    plt.hist(data[name], edgecolor='black')
    plt.title(f"Distribution for {name}", size=16)
    plt.ylabel('count')
    plt.show()

for names in column_names:
    hisplot(data_train, names)
```

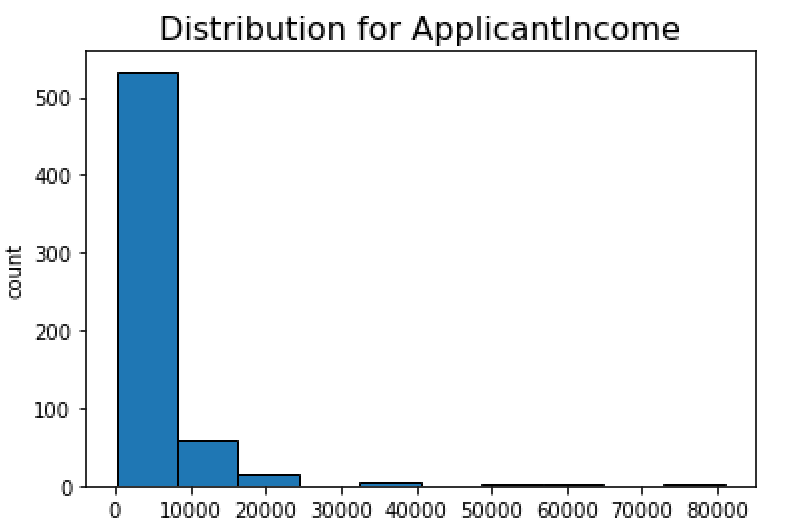

Distribution for Gender
count
500
400
300
200
100
0
Male
Female

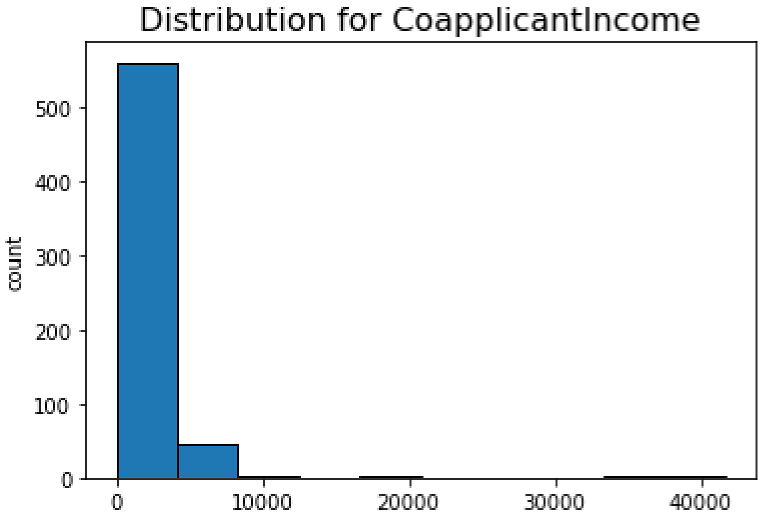

Distribution for Married
count
400
350
300
250
200
150
100
50
0
No
Yes

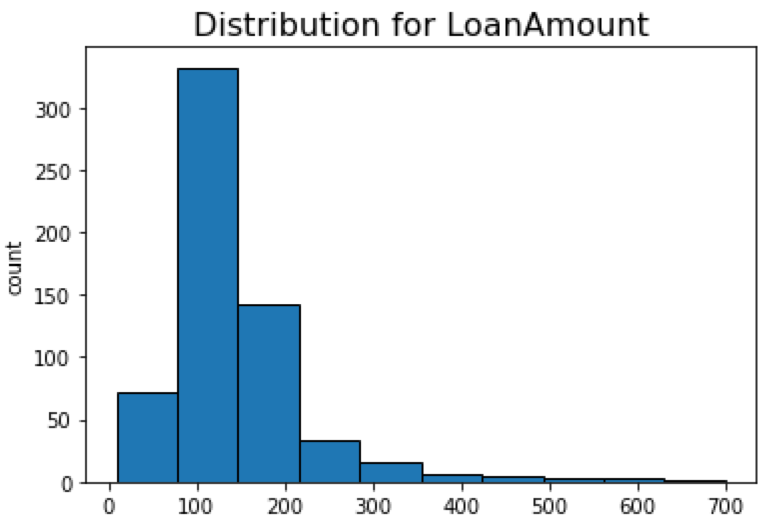

Distribution for Dependents
count
350
300
250
200
150
100
50
0
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0

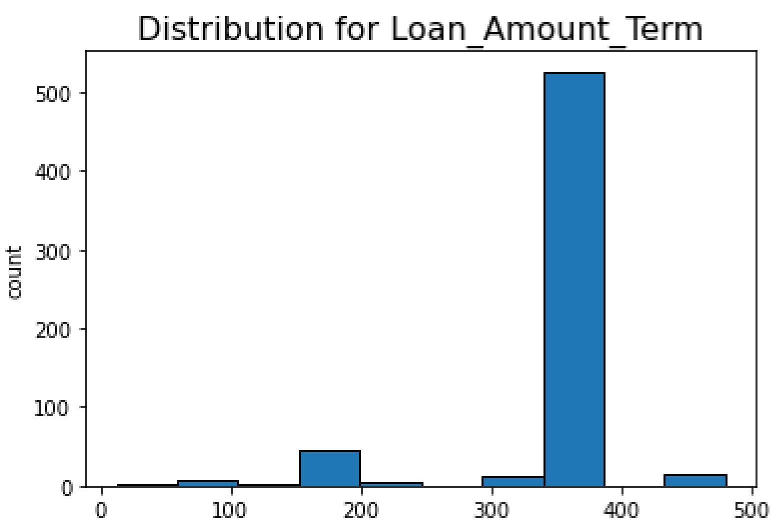

Distribution for Education
count
0
100
200
300
400
500
Graduate
Not Graduate

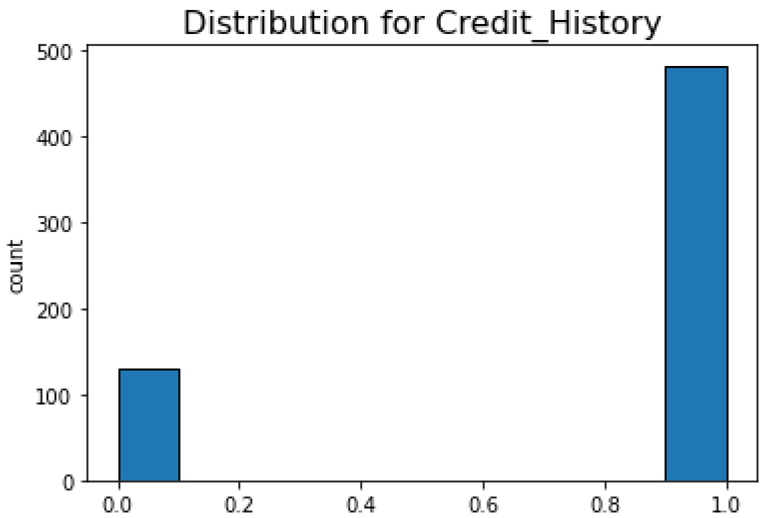

Distribution for Self_Employed
count
0
100
200
300
400
500
No
Yes

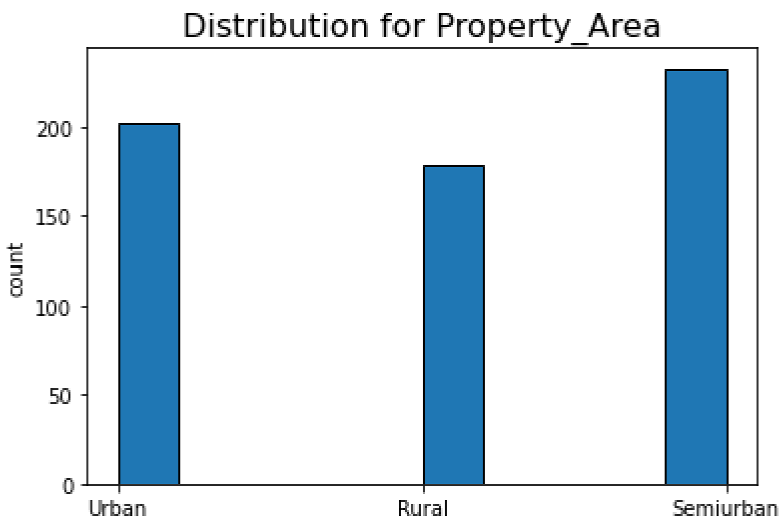

Distribution for ApplicantIncome
count
0
100
200
300
400
500
0
10000
20000
30000
40000
50000
60000
70000
80000

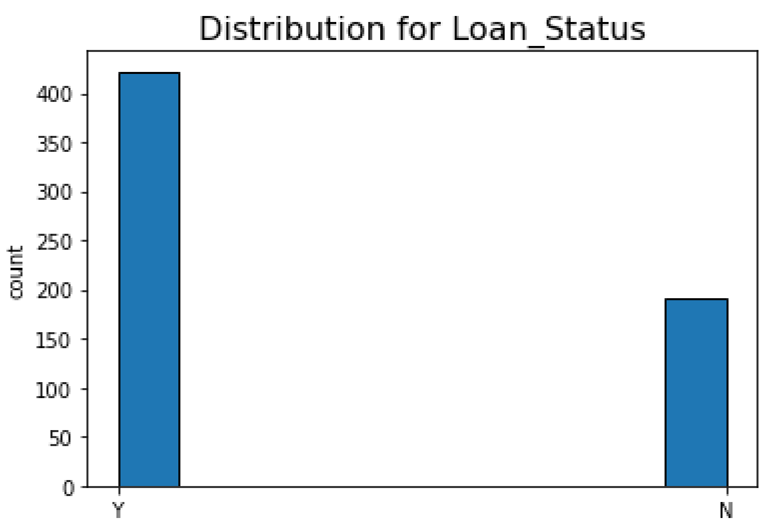

Distribution for CoapplicantIncome
count
0
100
200
300
400
500
0
10000
20000
30000
40000


Distribution for LoanAmount
count
0
50
100
150
200
250
300
0
100
200
300
400
500
600
700


Distribution for Loan_Amount_Term
count
0
100
200
300
400
500
0
100
200
300
400
500







In [52]:

```
import pandas as pd

train_gender = pd.Categorical(list(data_train['Gender']), categories=['Male', 'Female']
test_gender = pd.Categorical(list(data_test['Gender']), categories=['Male', 'Female'])

train_codes, uniques = pd.factorize(train_gender, sort=True)
data_train['Gender'] = train_codes
```

```
test_codes, uniques = pd.factorize(test_gender, sort=True)
data_test['Gender'] = test_codes
```

In [53]:
```
train_married = pd.Categorical(list(data_train['Married']), categories=['No', 'Yes'])
test_married = pd.Categorical(list(data_test['Married']), categories=['No', 'Yes'])

train_codes, uniques = pd.factorize(train_married, sort=True)
data_train['Married'] = train_codes

test_codes, uniques = pd.factorize(test_married, sort=True)
data_test['Married'] = test_codes
```

In [54]:
```
train_edu = pd.Categorical(list(data_train['Education']), categories = data_train['Educ
test_edu = pd.Categorical(list(data_test['Education']), categories = data_test['Educati

train_codes, uniques = pd.factorize(train_edu, sort=True)
data_train['Education'] = train_codes

test_codes, uniques = pd.factorize(test_edu, sort=True)
data_test['Education'] = test_codes
```

In [55]:
```
train_se = pd.Categorical(list(data_train['Self_Employed']), categories=data_train['Sel
test_se = pd.Categorical(list(data_test['Self_Employed']), categories=data_test['Self_E

train_codes, uniques = pd.factorize(train_se, sort=True)
data_train['Self_Employed'] = train_codes

test_codes, uniques = pd.factorize(test_se, sort=True)
data_test['Self_Employed'] = test_codes
```

In [56]:
```
train_pa = pd.Categorical(list(data_train['Property_Area']), categories=data_train['Pro
test_pa = pd.Categorical(list(data_test['Property_Area']), categories=data_test['Proper

train_codes, uniques = pd.factorize(train_pa, sort=True)
data_train['Property_Area'] = train_codes

test_codes, uniques = pd.factorize(test_pa, sort=True)
data_test['Property_Area'] = test_codes
```

In [57]:
```
train_ls = pd.Categorical(list(data_train['Loan_Status']), categories=data_train['Loan_

train_codes, uniques = pd.factorize(train_ls, sort=True)
data_train['Loan_Status'] = train_codes
```

In [58]:
```
data_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Gender             614 non-null    int64
```

```
 1   Married            614 non-null    int64  
 2   Dependents         614 non-null    int64  
 3   Education          614 non-null    int64  
 4   Self_Employed      614 non-null    int64  
 5   ApplicantIncome    614 non-null    int64  
 6   CoapplicantIncome  614 non-null    float64
 7   LoanAmount         614 non-null    float64
 8   Loan_Amount_Term   614 non-null    float64
 9   Credit_History     614 non-null    float64
 10  Property_Area      614 non-null    int64  
 11  Loan_Status        614 non-null    int64  
dtypes: float64(4), int64(8)
memory usage: 57.7 KB
```

In [59]:

```
data_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 367 entries, 0 to 366
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Gender             367 non-null    int64  
 1   Married            367 non-null    int64  
 2   Dependents         367 non-null    int64  
 3   Education          367 non-null    int64  
 4   Self_Employed      367 non-null    int64  
 5   ApplicantIncome    367 non-null    int64  
 6   CoapplicantIncome  367 non-null    int64  
 7   LoanAmount         367 non-null    float64
 8   Loan_Amount_Term   367 non-null    float64
 9   Credit_History     367 non-null    float64
 10  Property_Area      367 non-null    int64  
dtypes: float64(3), int64(8)
memory usage: 31.7 KB
```

In [60]:

```
print(data_train.shape)
print(data_test.shape)
```

```
(614, 12)
(367, 11)
```

In [61]:

```
def data_Norm(data):
    means = np.mean(data,axis=0)
    stds = np.std(data, axis=0)
    data_norm = (data - means) / stds
    return data_norm
```

In [62]:

```
y= data_train['Loan_Status']

X = data_train.drop(columns=['Loan_Status'], axis=1)
```

In [63]:

```
X = data_Norm(X)
X = np.array(X)
y = np.array([y]).T
```

```
print(X.shape)
print(y.shape)
```

```
(614, 11)
(614, 1)
```

In [64]:

```
X = np.insert(X, 0, 1, axis=1)
print(X.shape)
```

```
(614, 12)
```

In [65]:

```
import random
def train_test_split(X, y, percent_train, random_seed):
    idx = np.arange(0, X.shape[0])
    random.seed(random_seed)
    random.shuffle(idx)
    m = X.shape[0]
    m_train = int(m * percent_train)
    train_idx = idx[: m_train]
    test_idx = idx[m_train :]

    X_train = X[train_idx, :]
    X_test = X[test_idx, :]

    y_train = y[train_idx]
    y_test = y[test_idx]

    return X_train, X_test, y_train, y_test
```

In [66]:

```
percent_train = 0.8
X_train, X_test, y_train, y_test = train_test_split(X, y, percent_train = percent_train
print("X train shape: ", X_train.shape)
print("X test shape: ", X_test.shape)
print("Y train shape: ", y_train.shape)
print("Y test shape: ", y_test.shape)
```

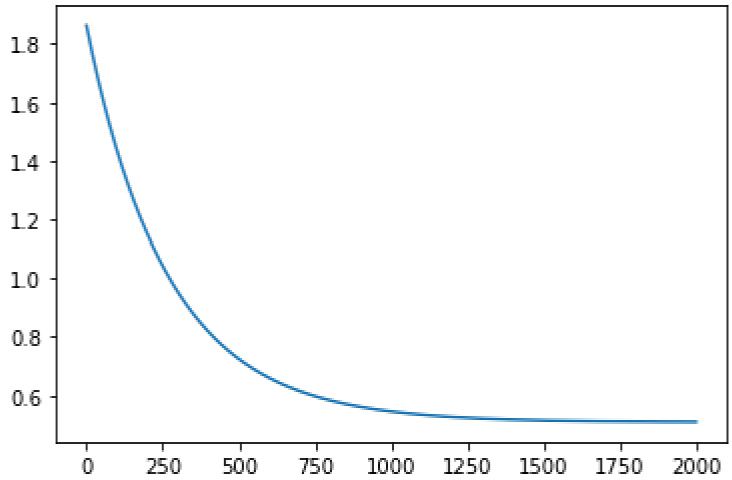
```
X train shape:  (491, 12)
X test shape:  (123, 12)
Y train shape:  (491, 1)
Y test shape:  (123, 1)
```

In [67]:

```
NM_model = Logistic_NM()

iterations = 2000
initial_theta = np.ones((X_train.shape[1],1))
nm_theta, nm_cost = NM_model.newtonsMethod(X_train, y_train, initial_theta, iterations)
print("theta:",nm_theta)

print(nm_cost[0])
plt.plot(nm_cost)
plt.show()
```

```
theta: [[-0.87974829]
 [ 0.01223725]
 [-0.21262895]
 [ 0.11319489]
 [ 0.17210148]
```

```
 [ 0.06764779]
 [ 0.04921505]
 [ 0.22121309]
 [ 0.05171198]
 [ 0.07482697]
 [-0.95384756]
 [-0.19542552]]
1.8632054110519107
```



In [68]:

```
n_acc = NM_model.getAccuracy(X_train, y_train, nm_theta)
print('Newton Accuracy for training set : ', n_acc)

n_acc = NM_model.getAccuracy(X_test, y_test, nm_theta)
print('Newton Accuracy for test set : ', n_acc)
```

```
Newton Accuracy for training set :  78.61507128309572
Newton Accuracy for test set :  72.35772357723577
```

The test set accuracy is the same as lab 3.

Summary

The training set accuracy for gradient method and newton methods are the same but the newton method required only 1000 iterations while the gradient method required 50000 iterations with 0.001 alpha.

In newton method, there is sometime occur that the Hessian matrix is singlur which means it does not have inverse matrix. So the np.linalg.inv cannot be used for that case but we can use np.linalg.pinv for the inverse.

After transforming to polar coordinate, we can the see that the accuracy is imporved.

In [ ]: