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 03: Logistic Regression

Thus far, the problems we've encountered have been *regression* problems, in which the target $y \in \mathbb{R}$.

Today we'll start experimenting with *classification* problems, beginning with *binary* classification problems, in which the target $y \in \{ 0, 1 \}$.

Background

The simplest approach to classification, applicable when the input feature vector $\mathbf{x} \in \mathbb{R}^n$, is a simple generalization of what we do in linear regression. Recall that in linear regression, we assume that the target is drawn from a Gaussian distribution whose mean is a linear function of $\mathbf{x}$:
$$ y \sim {\cal N}(\theta^\top \mathbf{x}, \sigma^2) $$

In logistic regression, similarly, we'll assume that the target is drawn from a Bernoulli distribution with parameter p being the probability of class 1:
$$ y \sim \text{Bernoulli}(p) $$

That's fine, but how do we model the parameter p? How is it related to $\mathbf{x}$?

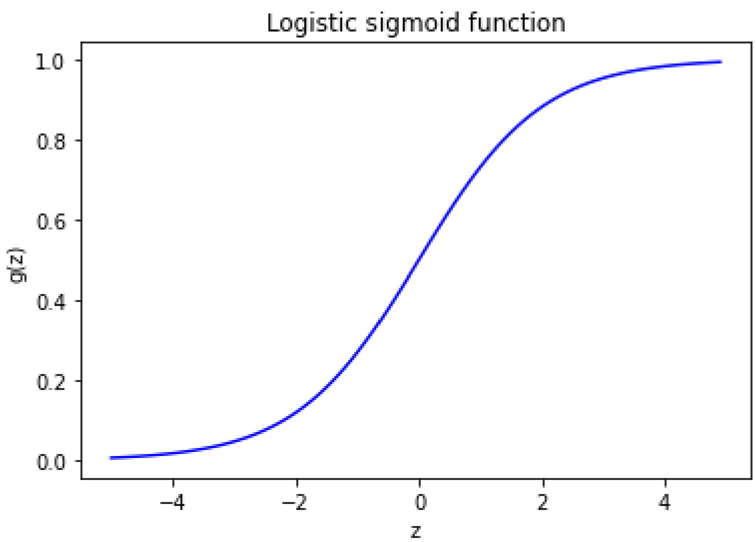
In linear regression, we assume that the mean of the Gaussian is $\theta^\top \mathbf{x}$, i.e., a linear function of $\mathbf{x}$.

In logistic regression, we'll assume that p is a "squashed" linear function of $\mathbf{x}$, i.e.,
$$ p = \text{sigmoid}(\theta^\top \mathbf{x}) = g(\theta^\top \mathbf{x}) = \frac{1}{1+e^{-\theta^\top \mathbf{x}}}. $$

Later, when we introduce generalized linear models, we'll see why p should take this form. For now, though, we can simply note that the selection makes sense. Since p is a discrete probability, p is bounded by $0 \le p \le 1$. The sigmoid function $g(\cdot)$ conveniently obeys these bounds:

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt

def f(z):
    return 1 / (1 + np.exp( -z ))

z = np.arange(-5, 5, 0.1)
plt.plot(z, f(z), 'b-')
plt.xlabel('z')
plt.ylabel('g(z)')
plt.title('Logistic sigmoid function')
plt.show()
```



We see that the sigmoid approaches 0 as its input approaches $-\infty$ and approaches 1 as its input approaches $+\infty$. If its input is 0, its value is 0.5.

Again, this choice of function may seem strange at this point, but bear with it! We'll derive this function from a more general principle, the generalized linear model, later.

OK then, we now understand that for logistic regression, the assumptions are:

1. The *data* are pairs $(\textbf{x}, y) \in \mathbb{R}^n \times \{ 0, 1 \}$.
2. The *hypothesis function* is $h_\theta(\textbf{x}) = \frac{1}{1+e^{-\theta^\top \mathbf{x}}}$.

What else do we need... ? A cost function and an algorithm for minimizing that cost function!

Cost function for logistic regression

You can refer to the lecture notes to see the derivation, but for this lab, let's just skip to the chase. With the hypothesis $h_\theta(\mathbf{x})$ chosen as above, the log likelihood function $\ell(\theta)$ can be derived as $$ \ell(\theta) = \log {\cal L}(\theta) = \sum_{i=1}^{m}y^{(i)}\log(h_{\theta}(\mathbf{x}^{(i)})) + (1 - y^{(i)})\log(1 - (h_{\theta}(\mathbf{x}^{(i)})) .$$

Negating the log likelihood function to obtain a loss function, we have
$$ J(\theta) = - \sum_{i=1}^m y^{(i)}\log h_\theta(\mathbf{x}^{(i)}) + (1-y^{(i)})\log(1-h_\theta(\textbf{x}^{(i)})) .$$

There is no closed-form solution to this problem like there is in linear regression, so we have to use gradient descent to find θ minimizing $J(\theta)$. Luckily, the function *is* convex in θ so there is just a single global minimum, and gradient descent is guaranteed to get us there eventually if we take the right step size.

The *stochastic* gradient of J, for a single observed pair $(\mathbf{x}, y)$, turns out to be (see lecture notes)
$$\nabla_J(\theta) = (h_\theta(\mathbf{x}) - y)\mathbf{x} .$$

Give some thought as to whether following this gradient to increase the loss J would make a worse classifier, and vice versa!

Finally, we obtain the update rule for the jth iteration selecting training pattern i:
$$ \theta^{(j+1)} \leftarrow \theta^{(j)} + \alpha(y^{(i)} - h_\theta(\textbf{x}^{(i)}))\textbf{x}^{(i)} .$$

Note that we can perform *batch gradient descent* simply by summing the single-pair gradient over the entire training set before taking a step, or *mini-batch gradient descent* by summing over a small subset of the data.

Example dataset 1: student admissions data

This example is from Andrew Ng's machine learning course on Coursera.

The data contain students' scores for two standardized tests and an admission decision (0 or 1).

In [3]:

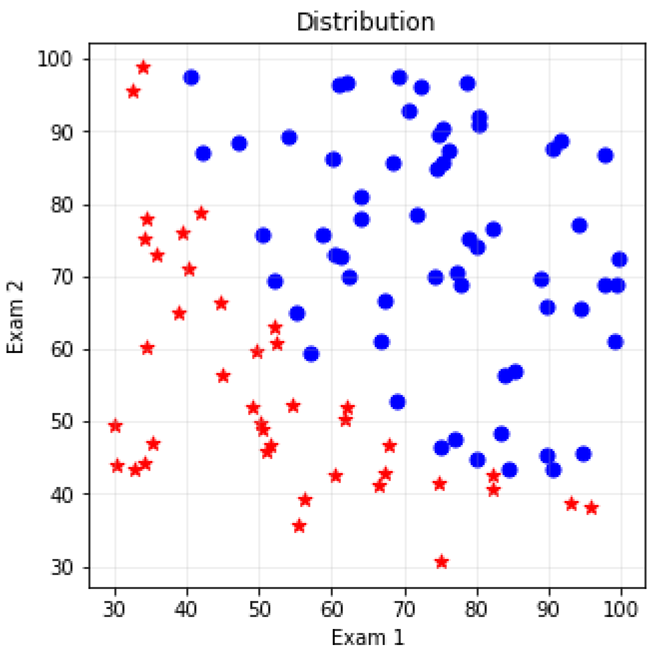
```
# Load student admissions data. The data file does not contain headers,
# so we use hard coded indices for exam 1, exam2, and the admission decision.

data = np.loadtxt('ex2data1.txt',delimiter = ',')
exam1_data = data[:,0]
exam2_data = data[:,1]
X = np.array([exam1_data, exam2_data]).T
y = data[:,2]

# Output some sample data

print('Exam scores', X[0:5,:])
print('-----------------------------')
print('Admission decision', y[0:5])
```

```
Exam scores [[34.62365962 78.02469282]
 [30.28671077 43.89499752]
 [35.84740877 72.90219803]
 [60.18259939 86.3085521 ]
 [79.03273605 75.34437644]]
-----------------------------
Admission decision [0. 0. 0. 1. 1.]
```

Let's plot the data...

In [4]:

```
# Plot the data

idx_0 = np.where(y == 0)
idx_1 = np.where(y == 1)

fig1 = plt.figure(figsize=(5, 5))
ax = plt.axes()
ax.set_aspect(aspect = 'equal', adjustable = 'box')
plt.title('Distribution')
plt.xlabel('Exam 1')
plt.ylabel('Exam 2')
plt.grid(axis='both', alpha=.25)
ax.scatter(exam1_data[idx_0], exam2_data[idx_0], s=50, c='r', marker='*', label='Not Admit
ted')
ax.scatter(exam1_data[idx_1], exam2_data[idx_1], s=50, c='b', marker='o', label='Admitted'
)
plt.show()
```



Let's see if we can find good values for θ without normalizing the data. We will definitely want to split the data into train and test, however...

In [5]:

```
import random

# As usual, we fix the seed to eliminate random differences between different runs

random.seed(12)

# Partion data into training and test datasets

m, n = X.shape
XX = np.insert(X, 0, 1, axis=1)
y = y.reshape(m, 1)
idx = np.arange(0, m)
random.shuffle(idx)
percent_train = .6
m_train = int(m * percent_train)
train_idx = idx[0:m_train]
test_idx = idx[m_train:]
X_train = XX[train_idx,:];
X_test = XX[test_idx,:];

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

Important functions needed later

Let's put all of our important functions here...

In [6]:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def h(X, theta):
    return sigmoid(X @ theta)

def grad_j(X, y, y_pred):
    return X.T @ (y - y_pred) / X.shape[0]

def j(theta, X, y):
    y_pred = h(X, theta)
    error = (-y * np.log(y_pred)) - ((1 - y) * np.log(1 - y_pred))
    cost = sum(error) / X.shape[0]
    grad = grad_j(X, y, y_pred)
    return cost[0], grad
```

Initialize theta

In any iterative algorithm, we need an initial guess. Here we'll just use zeros for all parameters.

In [7]:

```
# Initialize our parameters, and use them to make some predictions

theta_initial = np.zeros((n+1, 1))

print('Initial theta:', theta_initial)
print('Initial predictions:', h(XX, theta_initial)[0:5,:])
print('Targets:', y[0:5,:])
```

```
Initial theta: [[0.]
 [0.]
 [0.]]
Initial predictions: [[0.5]
 [0.5]
 [0.5]
 [0.5]
 [0.5]]
Targets: [[0.]
 [0.]
 [0.]
 [1.]
 [1.]]
```

Training function

Here's a function to do batch training for `num_iters` iterations.

In [8]:

```
def train(X, y, theta_initial, alpha, num_iters):
    theta = theta_initial
    j_history = []
    for i in range(num_iters):
        cost, grad = j(theta, X, y)
        theta = theta + alpha * grad
        j_history.append(cost)
    return theta, j_history
```

Do the training

Here we run the training function for a million batches!

In [9]:

```
# Train for 1000000 iterations on full training set

alpha = .0005
num_iters = 1000000
theta, j_history = train(X_train, y_train, theta_initial, alpha, num_iters)

print("Theta optimized:", theta)
print("Cost with optimized theta:", j_history[-1])
```

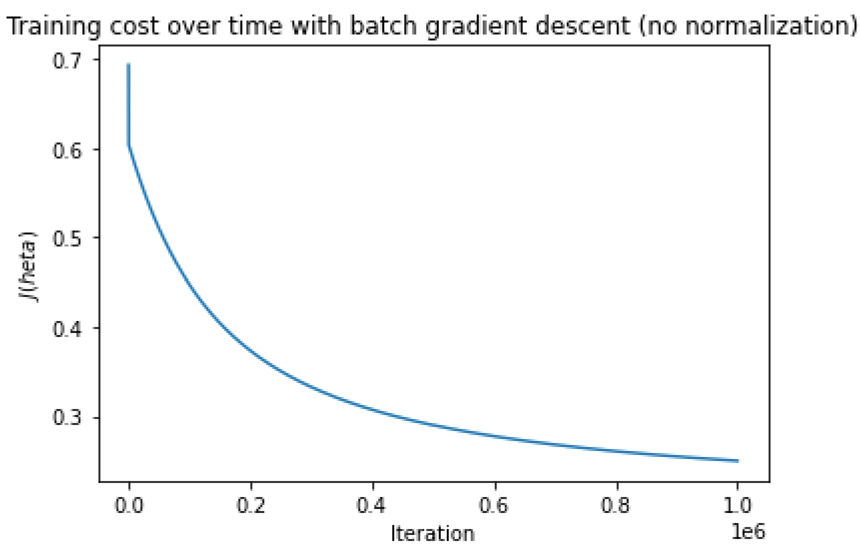
```
Theta optimized: [[-11.29380461]
 [  0.10678604]
 [  0.07994591]]
Cost with optimized theta: 0.24972975869900035
```

Plot the loss curve

Next let's plot the loss curve (loss as a function of iteration).

In [10]:

```
plt.plot(j_history)
plt.xlabel("Iteration")
plt.ylabel("$J(\theta)$")
plt.title("Training cost over time with batch gradient descent (no normalization)")
plt.show()
```



In-lab exercise from Example 1 (Total 35 points)

That took a long time, right?

We'll see if we can do better. We will try the following:

1. Try increasing the learning rate α and starting with a better initial θ. How much does it help?
    - Try at least 2 learning rate α with 2 difference θ (4 experiments)
    - Do not forget to plot the loss curve to compare your results
2. Better yet, try *normalizing the data* and see if the training converges better. How did it go?
    - Be sure to plot loss curves to compare the results with unnormalized and normalized data.
3. Discuss the effects of normalization, learning rate, and initial θ in your report.

Do this work in the following steps.

Exercise 1.1 (5 points)

Fill in two different values for α and θ.

Use variable names `alpha1` , `alpha2` , `theta_initial1` , and `theta_initial2` .

In [11]:

```
# grade task: change 'None' value to number(s) or function
# YOUR CODE HERE
#raise NotImplementedError()
# declare your alphas
alpha1 = 0.0001
alpha2 = 0.005

# initialize thetas as you want
theta_initial1 = np.ones((n+1, 1))
theta_initial2 = np.random.rand(n+1,1)

# define your num iterations
num_iters = 1000000
```

In [12]:

```
alpha_list = [alpha1, alpha2]
print('alpha 1:', alpha1)
print('alpha 2:', alpha2)

theta_initial_list = [theta_initial1, theta_initial2]
print('theta 1:', theta_initial_list[0])
print('theta 2:', theta_initial_list[1])

print('Use num iterations:', num_iters)

# Test function: Do not remove
assert alpha_list[0] is not None and alpha_list[1] is not None, "Alpha has not been fille
d"
chk1 = isinstance(alpha_list[0], (int, float))
chk2 = isinstance(alpha_list[1], (int, float))
assert chk1 and chk2, "Alpha must be number"
assert theta_initial_list[0] is not None and theta_initial_list[1] is not None, "initializ
ed theta has not been filled"
chk1 = isinstance(theta_initial_list[0], (list,np.ndarray))
chk2 = isinstance(theta_initial_list[1], (list,np.ndarray))
assert chk1 and chk2, "Theta must be list"
chk1 = ((n+1, 1) == theta_initial_list[0].shape)
chk2 = ((n+1, 1) == theta_initial_list[1].shape)
assert chk1 and chk2, "Theta size are incorrect"
assert num_iters is not None and isinstance(num_iters, int), "num_iters must be integer"
print("success!")
# End Test function
```

```
alpha 1: 0.0001
alpha 2: 0.005
theta 1: [[1.]
 [1.]
 [1.]]
theta 2: [[0.36005027]
 [0.680931  ]
 [0.81861755]]
Use num iterations: 1000000
success!
```

Exercise 1.2 (5 points)

Fill in the code required to train your model on a particular α and θ:

In [13]:

```
# grade task: change 'None, None' value to number(s) or function
j_history_list = []
theta_list = []
for alpha in alpha_list:
    for theta_initial in theta_initial_list:
        # YOUR CODE HERE
        theta, j_history = train(X_train, y_train, theta_initial, alpha, num_iters)
        #raise NotImplementedError()
        theta_i = theta
        j_history_i = j_history
        # theta_i, j_history_i = None, None
        j_history_list.append(j_history_i)
        theta_list.append(theta_i)
```

```
/tmp/ipykernel_162/2314104836.py:12: RuntimeWarning: divide by zero encounter
ed in log
  error = (-y * np.log(y_pred)) - ((1 - y) * np.log(1 - y_pred))
/tmp/ipykernel_162/2314104836.py:12: RuntimeWarning: invalid value encountere
d in multiply
  error = (-y * np.log(y_pred)) - ((1 - y) * np.log(1 - y_pred))
```

In [14]:

```
# Test function: Do not remove
assert theta_list[0] is not None and j_history_list[0] is not None, "No values in theta_li
st or j_history_list"
chk1 = isinstance(theta_list[0], (list,np.ndarray))
chk2 = isinstance(j_history_list[0][0], (int, float))
assert chk1 and chk2, "Wrong type in theta_list or j_history_list"
print("success!")
# End Test function
```

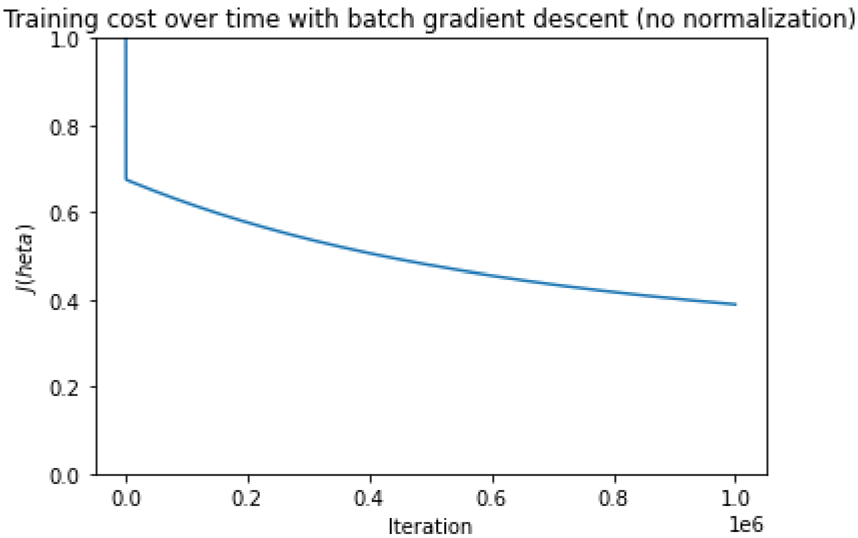
```
success!
```

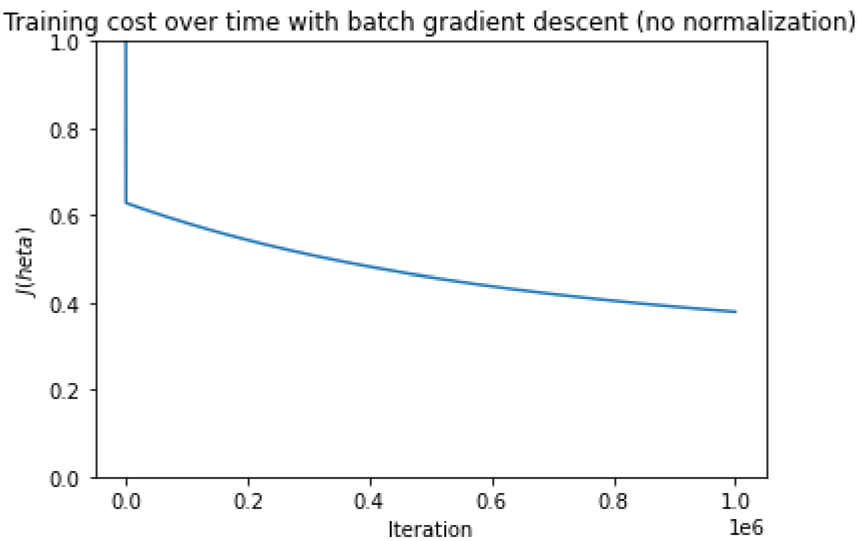
Exercise 1.3 (10 points)

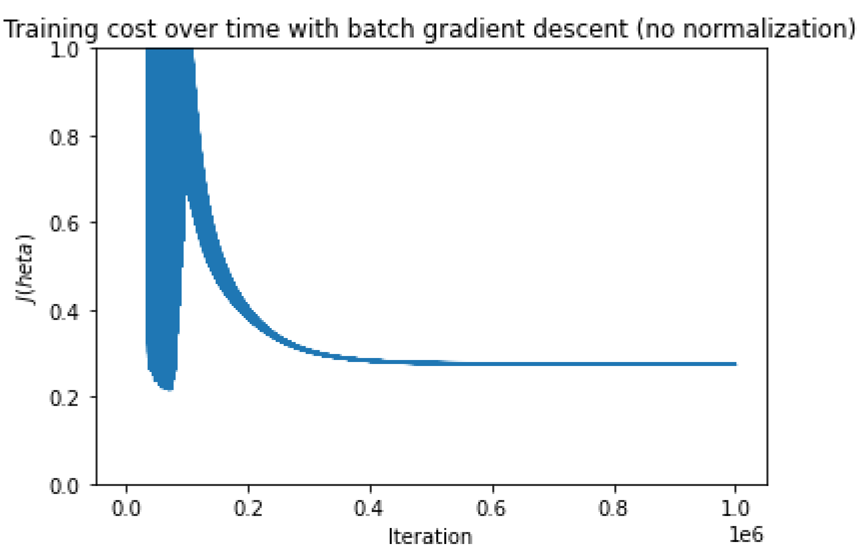
Write code to plot loss curves for each of the sequences in `j_history_list` from the previous exercise:

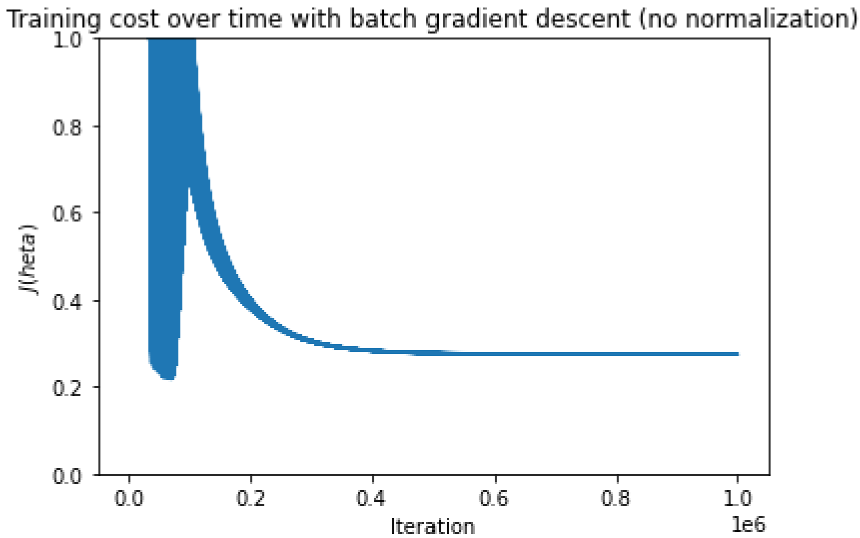
In [15]:

```
# YOUR CODE HERE

for j_history in j_history_list:
    plt.plot(j_history)
    plt.xlabel('Iteration')
    plt.ylabel("$J(\theta)$")
    plt.title("Training cost over time with batch gradient descent (no normalization)")
    plt.ylim(0, 1)
    plt.show()

#raise NotImplementedError()
```


Training cost over time with batch gradient descent (no normalization)
J(heta)
Iteration
1e6


Training cost over time with batch gradient descent (no normalization)
J(heta)
Iteration
1e6


Training cost over time with batch gradient descent (no normalization)
J(heta)
Iteration
1e6



Exercise 1.4 (10 points)

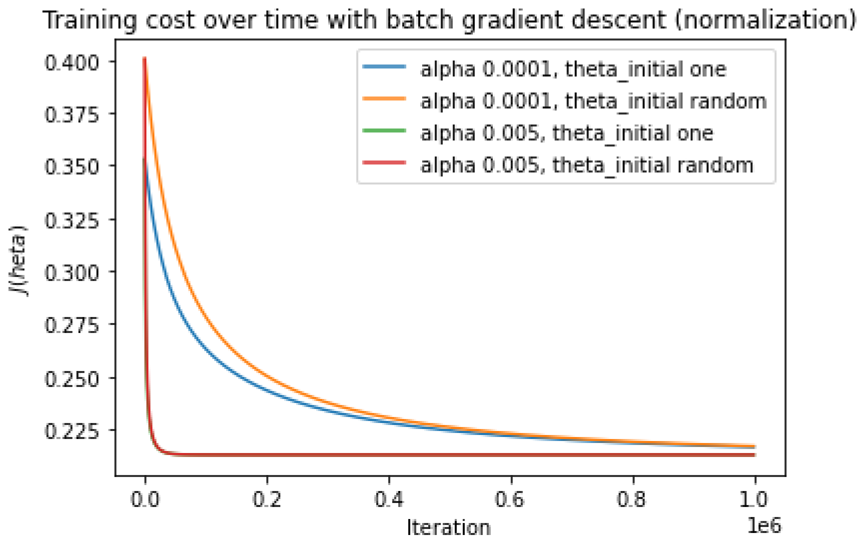
- Repeat your training, but **normalize** your data before training
- Compare the results between normalized data and unnormalized data

In [16]:

```
# code here
X, y
means = np.mean(data, axis=0)
stds = np.std(data, axis=0)
data_norm = (data - means) /stds

exam1_data = data_norm[:,0]
exam2_data = data_norm[:,1]
X = np.array([exam1_data, exam2_data]).T
y = data[:,2]

import random

# As usual, we fix the seed to eliminate random differences between different runs

random.seed(12)

# Partion data into training and test datasets

m, n = X.shape
XX = np.insert(X, 0, 1, axis=1)
y = y.reshape(m, 1)
idx = np.arange(0, m)
random.shuffle(idx)
percent_train = .6
m_train = int(m * percent_train)
train_idx = idx[0:m_train]
test_idx = idx[m_train:]
X_train = XX[train_idx,:];
X_test = XX[test_idx,:];

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

j_history_list = []
theta_list = []
for alpha in alpha_list:
    for theta_initial in theta_initial_list:
        # YOUR CODE HERE
        theta, j_history = train(X_train, y_train, theta_initial, alpha, num_iters)
        #raise NotImplementedError()
        theta_i = theta
        j_history_i = j_history
        # theta_i, j_history_i = None, None
        j_history_list.append(j_history_i)
        theta_list.append(theta_i)
```

In [17]:

```
alphal = ['0.0001','0.0001', '0.005', '0.005']
thetal = ['one', 'random', 'one', 'random']
for i in range(len(j_history_list)):
    plt.plot(j_history_list[i], label='alpha ' + str(alphal[i]) + ', theta_initial ' + str
(thetal[i]))
    plt.xlabel('Iteration')
    plt.ylabel("$J(\theta)$")
    plt.title("Training cost over time with batch gradient descent (normalization)")
    plt.legend()
plt.show()
```



Exercise 1.5 (5 points)

Discuss the effects of normalization, learning rate, and initial θ in your report.

Write your discussion here.

The logistic regression decision boundary

Note that when $\theta^\top \textbf{x} = 0$, we have $h_\theta(\textbf{x}) = 0.5$. That is, we are equally unsure as to whether $\textbf{x}$ belongs to class 0 or class 1. The contour at which $h_\theta(\textbf{x}) = 0.5$ is called the classifier's *decision boundary*.

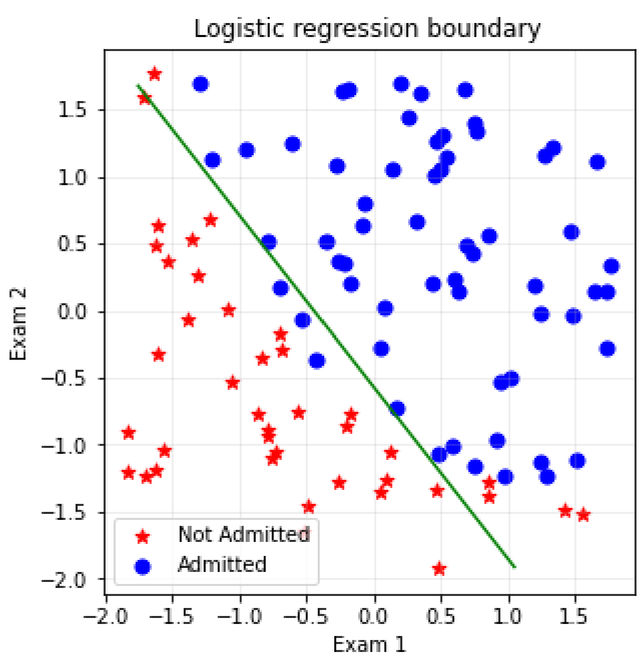
We know that in the plane, the equation $$ax+by+c=0$$ is the general form of a 2D line. In our case, we have $$\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$$ as our decision boundary, but clearly, this is just a 2D line in the plane. So when we plot x_1 against x_2, it is easy to plot the boundary line.

In [18]:

```
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 [19]:

```
fig1 = plt.figure(figsize=(5,5))
ax = plt.axes()
ax.set_aspect(aspect = 'equal', adjustable = 'box')
plt.title('Logistic regression boundary')
plt.xlabel('Exam 1')
plt.ylabel('Exam 2')
plt.grid(axis='both', alpha=.25)
ax.scatter(X[:,0][idx_0], X[:,1][idx_0], s=50, c='r', marker='*', label='Not Admitted')
ax.scatter(X[:,0][idx_1], X[:,1][idx_1], s=50, c='b', marker='o', label='Admitted')
point_1, point_2 = boundary_points(X, theta)
plt.plot([point_1[0,0], point_2[0,0]],[point_1[1,0], point_2[1,0]], 'g-')
plt.legend(loc=0)
plt.show()
```



You may have to adjust the above code to make it work with normalized data.

Test set performance

Now let's apply the learned classifier to the test data we reserved in the beginning:

In [20]:

```
def r_squared(y, y_pred):
    return 1 - np.square(y - y_pred).sum() / np.square(y - y.mean()).sum()
```

In [21]:

```
y_test_pred_soft = h(X_test, theta)
y_test_pred_hard = (y_test_pred_soft > 0.5).astype(int)

test_rsq_soft = r_squared(y_test, y_test_pred_soft)
test_rsq_hard = r_squared(y_test, y_test_pred_hard)
test_acc = (y_test_pred_hard == y_test).astype(int).sum() / y_test.shape[0]

print('Got test set soft R^2 %0.4f, hard R^2 %0.4f, accuracy %0.2f' % (test_rsq_soft, test
_rsq_hard, test_acc))
```

```
Got test set soft R^2 0.7382, hard R^2 0.6931, accuracy 0.93
```

For classification, accuracy is probably the more useful measure of goodness of fit.

Example 2: Loan prediction dataset

Let's take another example dataset and see what we can do with it.

This dataset is from Kaggle (https://www.kaggle.com/altruistdelhite04/loan-prediction-problem-dataset).

The data concern loan applications. It has 12 independent variables, including 5 categorical variables. The dependent variable is the decision "Yes" or "No" for extending a loan to an individual who applied.

One thing we will have to do is to clean the data, by filling in missing values and converting categorical data to reals. We will use the Python libraries pandas and sklearn to help with the data cleaning and preparation.

Read the data and take a look at it

In [22]:

```
# Import Pandas. You may need to run "pip3 install pandas" at the console if it's not alre
ady installed

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 [23]:

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

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

```
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
```

Handle missing values

We can see from the above table that the `Married` column has 3 missing values in the training dataset and 0 missing values in the test dataset. Let's take a look at the distribution over the datasets then fill in the missing values in approximately the same ratio.

You may be interested to look at the [documentation of the Pandas `fillna()` function (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html). It's great!

In [24]:

```
# 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().su
m())
```

```
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
```

Now the number of examples missing the `Married` attribute is 0.

Let's complete the data processing based on examples given and logistic regression model on training dataset. Then we'll get the model's accuracy (goodness of fit) on the test dataset.

Here is another example of filling in missing values for the `Dependents` (number of children and other dependents) attribute. We see that categorical values are all numeric except one value "3+" Let's create a new category value "4" for "3+" and ensure that all the data is numeric:

In [25]:

```
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_test,
num_1_test, num_2_test, num_3_test):
    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
```

Once missing values are filled in, you'll want to convert strings to numbers.

Finally, here's an example of replacing missing values for a numeric attribute. Typically, we would use the mean of the attribute over the training set.

In [26]:

```
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
```

Take-home exercise (65 points)

Using the data from Example 2 above, finish the data cleaning and preparation. Build a logistic regression model based on the cleaned dataset and report the accuracy on the test and training sets.

- Set up $\mathbf{x}$ and y data (10 points)
- Train a logistic regression model and return the values of θ and J you obtained. Find the best α you can; you may find it best to normalize before training. (30 points)
- Using the best model parameters θ you can find, run on the test set and get the model's accuracy. (10 points)
- Summarize what you did to find the best results in this take home exercise. (15 points)

To turn in

Turn in this Jupyter notebook with your solutions to he exercises and your experiment reports, both for the in-lab exercise and the take-home exercise. Be sure you've discussed what you learned in terms of normalization and data cleaning and the results you obtained.

In [27]:

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

```
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 [28]:

```
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)

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().su
m())
print('Missing values for test data:\n------------------------\n', data_test.isnull().sum
())
```

```
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 [29]:

```
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().su
m())
print('Missing values for test data:\n------------------------\n', data_test.isnull().sum
())
```

```
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 [30]:

```
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['Loa
n_Amount_Term'].isnull().sum())
data_test['Loan_Amount_Term'].fillna(LoanAT_mean, inplace = True, limit = data_test['Loan_
Amount_Term'].isnull().sum())


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

```
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 [31]:

```
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().su
m())
print('Missing values for test data:\n------------------------\n', data_test.isnull().sum
())
```

```
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
```

First we will drop the 'Loan_ID' columns as it will not help with learning of the model.

In [32]:

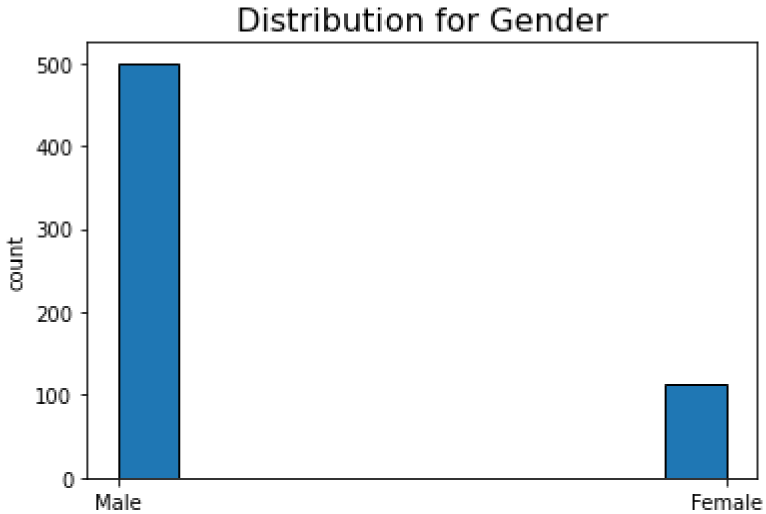
```
data_train.drop(columns=['Loan_ID'], inplace = True)
data_test.drop(columns=['Loan_ID'], inplace = True)
```

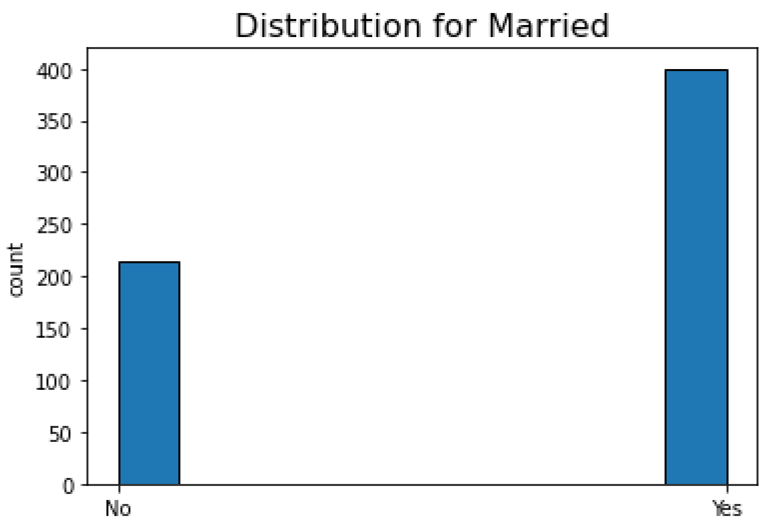
In [33]:

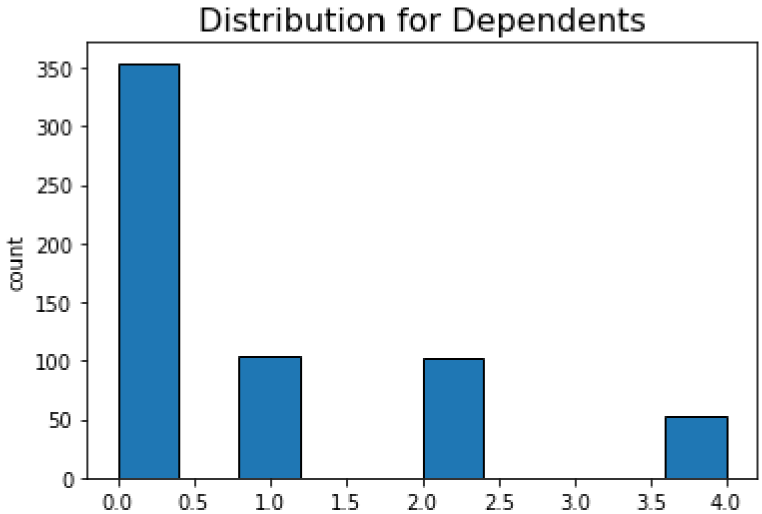
```
data_train['Dependents'] = data_train['Dependents'].astype(int)
data_test['Dependents'] = data_test['Dependents'].astype(int)
```

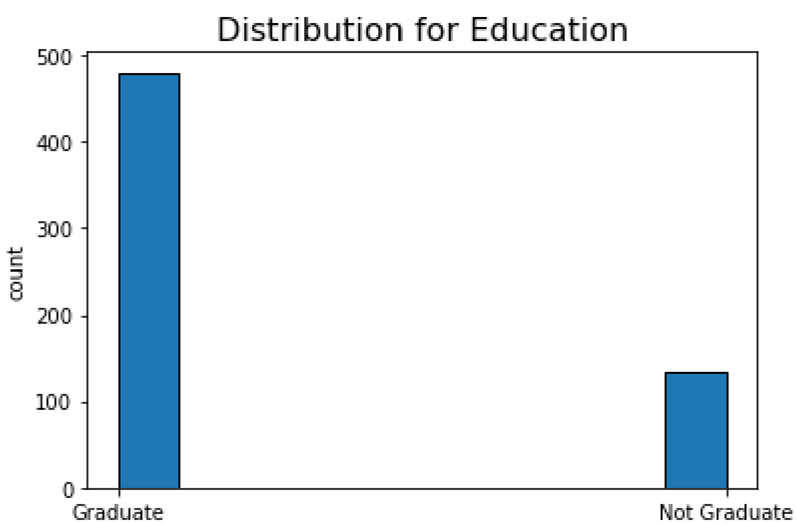
In [34]:

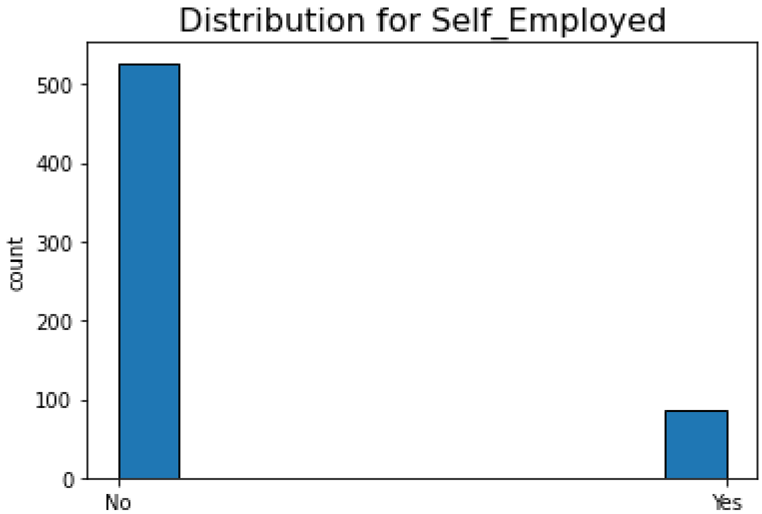
```
column_names = list(data_train.columns)

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)
```

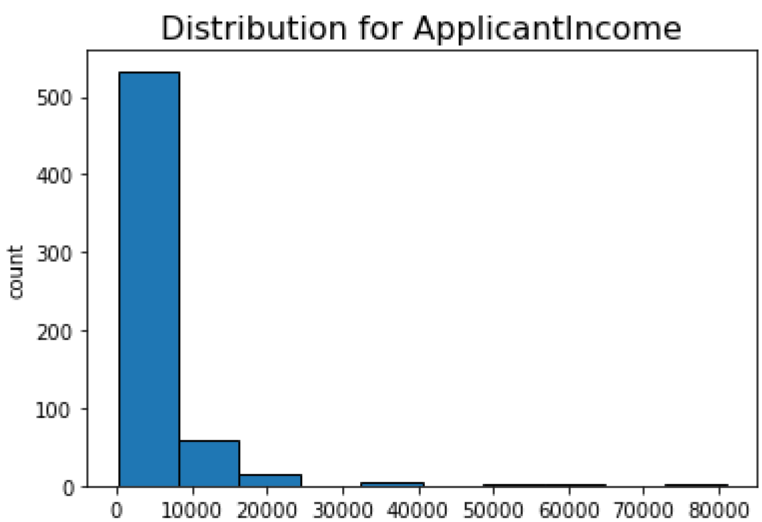

Distribution for Gender
count
500
400
300
200
100
0
Male
Female

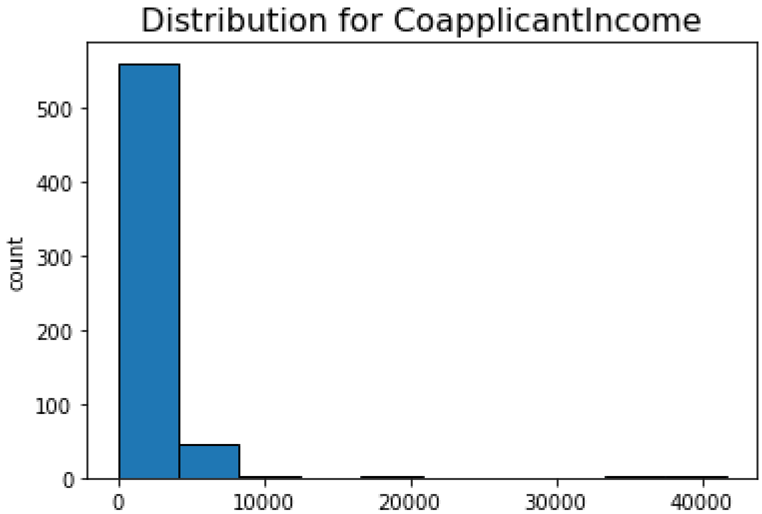

Distribution for Married
count
400
350
300
250
200
150
100
50
0
No
Yes

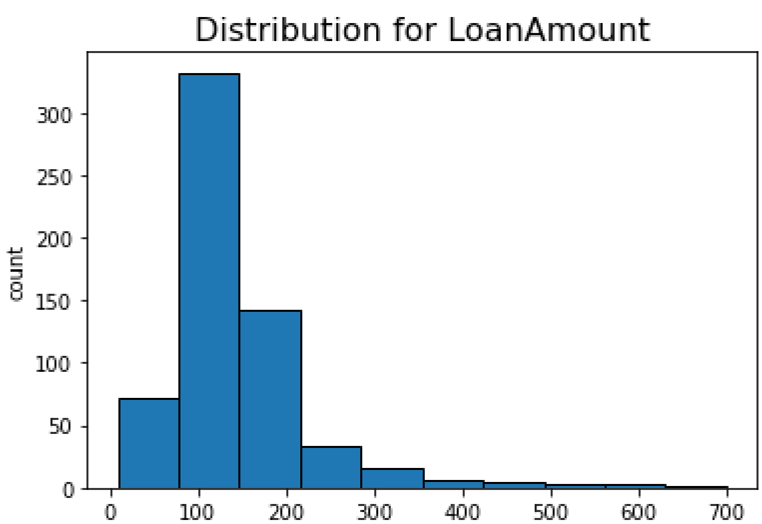

Distribution for Dependents
count
350
300
250
200
150
100
50
0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

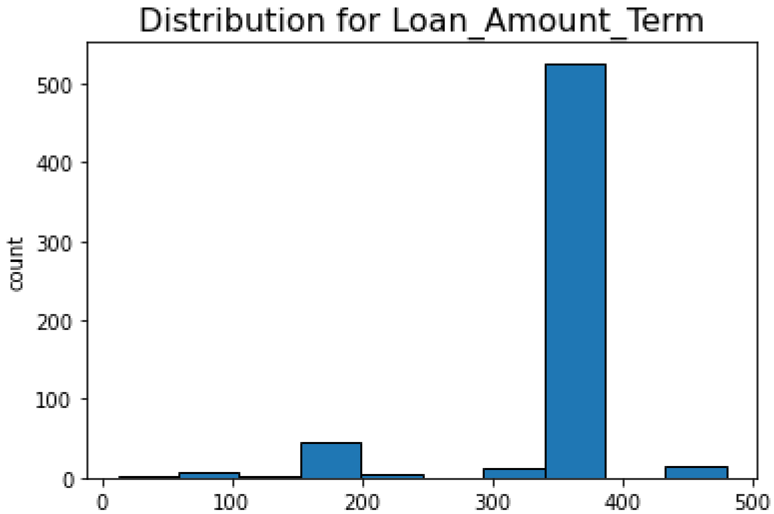

Distribution for Education
count
500
400
300
200
100
0
Graduate
Not Graduate

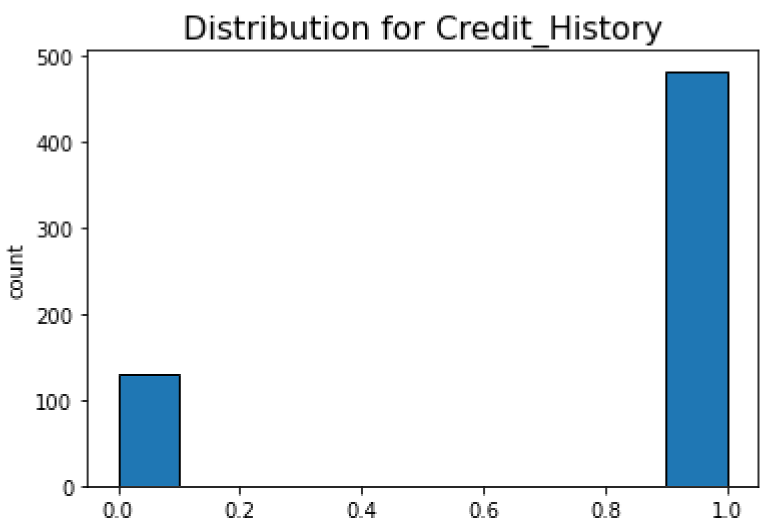

Distribution for Self_Employed
count
0
100
200
300
400
500
No
Yes

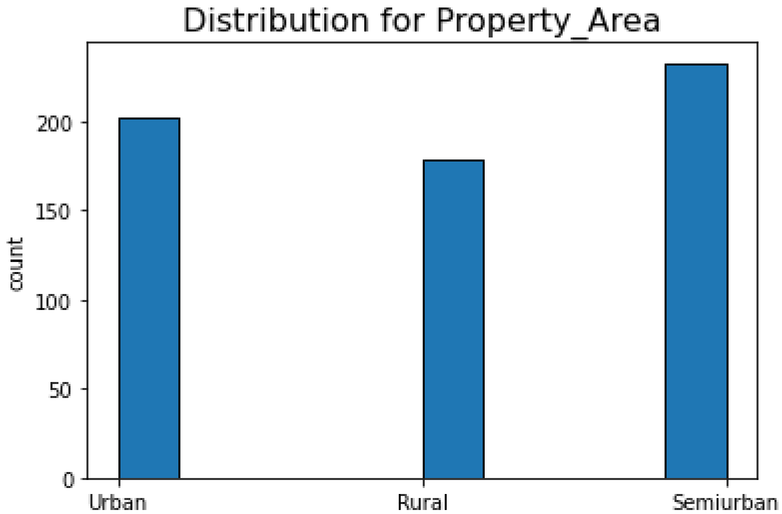

Distribution for ApplicantIncome
count
0
100
200
300
400
500
0
10000
20000
30000
40000
50000
60000
70000
80000

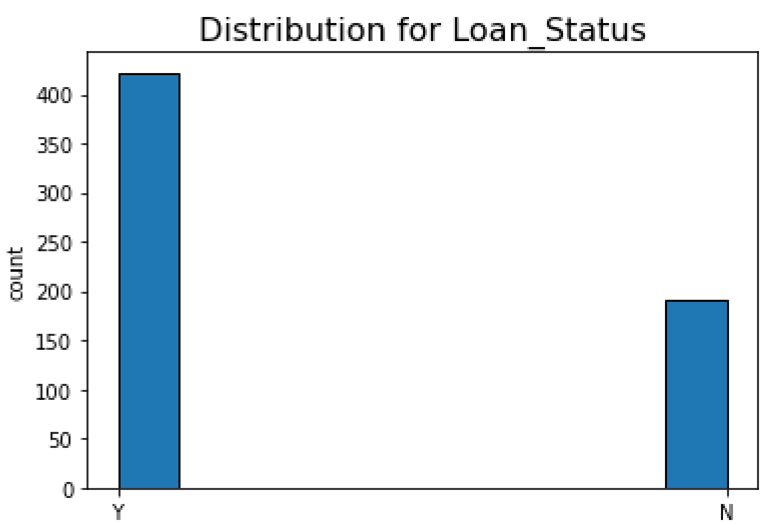

Distribution for CoapplicantIncome
count
0
100
200
300
400
500
0
10000
20000
30000
40000


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


Distribution for Credit_History
count
0
100
200
300
400
500
0.0
0.2
0.4
0.6
0.8
1.0


Distribution for Property_Area
count
0
50
100
150
200
Urban
Rural
Semiurban


Distribution for Loan_Status
count
0
50
100
150
200
250
300
350
400
Y
N

In [35]:

```
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 [36]:

```
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 [37]:

```
train_edu = pd.Categorical(list(data_train['Education']), categories = data_train['Educati
on'].unique())
test_edu = pd.Categorical(list(data_test['Education']), categories = data_test['Education'
].unique())

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 [38]:

```
train_se = pd.Categorical(list(data_train['Self_Employed']), categories=data_train['Self_E
mployed'].unique())
test_se = pd.Categorical(list(data_test['Self_Employed']), categories=data_test['Self_Empl
oyed'].unique())

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 [39]:

```
train_pa = pd.Categorical(list(data_train['Property_Area']), categories=data_train['Proper
ty_Area'].unique())
test_pa = pd.Categorical(list(data_test['Property_Area']), categories=data_test['Property_
Area'].unique())

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 [40]:

```
train_ls = pd.Categorical(list(data_train['Loan_Status']), categories=data_train['Loan_Sta
tus'].unique())

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

In [41]:

```
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 [42]:

```
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 [43]:

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

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

In [44]:

```
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 [45]:

```
y= data_train['Loan_Status']

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

In [46]:

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

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

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

In [47]:

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

```
(614, 12)
```

In [48]:

```
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 [49]:

```
percent_train = 0.8
X_train, X_test, y_train, y_test = train_test_split(X, y, percent_train = percent_train, r
andom_seed=1000)
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 [50]:

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def h_theta(X, theta):
    return sigmoid(X.dot(theta))

def grad_j(X, y, y_pred):
    return X.T @(y - y_pred)/ X.shape[0]

def j(theta, X, y):
    y_pred = h_theta(X, theta)
    error = (-y * np.log(y_pred)) - ((1 - y) * np.log(1 - y_pred))
    cost = sum(error) / X.shape[0]
    grad = grad_j(X, y, y_pred)
    return cost[0], grad

def train(X, y, theta_initial, alpha, num_iters):
    theta = theta_initial
    j_history = []
    for i in range(num_iters):
        cost, grad = j(theta, X, y)
        theta = theta + alpha * grad
        j_history.append(cost)
    return theta, j_history

def r_squared(y, y_pred):
    return 1 - np.square(y - y_pred).sum() / np.square(y - y.mean()).sum()
```

In [51]:

```
alpha_list = [0.0009, 0.0005, 0.0001, 0.009, 0.005, 0.001, 0.09, 0.05, 0.01]
theta_initial = np.zeros((X_train.shape[1], 1))
num_iters = 100000

j_history_list = []
theta_list = []
for alpha in alpha_list:
    theta, j_history = train(X_train, y_train, theta_initial, alpha, num_iters)
    theta_i = theta
    j_history_i = j_history
    j_history_list.append(j_history_i)
    theta_list.append(theta_i)
    print("alpha : ", alpha)
    print("Theta_initial : ", theta_initial)
    print("Theta optimized : ", theta)
    print("Cost with optimized theta: ", j_history[-1])
    print("="*30)
    print('='*30)
```

```
alpha :  0.0009
Theta_initial :  [[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Theta optimized :  [[-0.91067134]
 [-0.02298181]
 [-0.2260846 ]
 [ 0.08744181]
 [ 0.16361142]
 [ 0.05051641]
 [-0.00342504]
 [ 0.13213894]
 [ 0.0461376 ]
 [ 0.05972628]
 [-0.95474671]
 [-0.2103654 ]]
Cost with optimized theta:  0.5083372035955813
==============================
==============================
alpha :  0.0005
Theta_initial :  [[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Theta optimized :  [[-0.90971485]
 [-0.02155805]
 [-0.22365765]
 [ 0.08596138]
 [ 0.16356107]
 [ 0.05063116]
 [-0.00293081]
 [ 0.13199162]
 [ 0.04574625]
 [ 0.05945524]
 [-0.954185  ]
 [-0.20997237]]
Cost with optimized theta:  0.5083377792405659
==============================
==============================
alpha :  0.0001
```

```
Theta_initial :  [[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Theta optimized :  [[-0.74610516]
 [ 0.00977211]
 [-0.13805941]
 [ 0.05039023]
 [ 0.1398901 ]
 [ 0.05069512]
 [-0.00156322]
 [ 0.1028685 ]
 [ 0.03927622]
 [ 0.03453535]
 [-0.81683748]
 [-0.16734947]]
Cost with optimized theta:  0.5125382873001048
==============================
==============================
alpha :  0.009
Theta_initial :  [[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Theta optimized :  [[-0.91068977]
 [-0.02302702]
 [-0.22616074]
 [ 0.08748206]
 [ 0.16360724]
 [ 0.05051176]
 [-0.00347998]
 [ 0.13211717]
 [ 0.04619396]
 [ 0.05971734]
 [-0.95475721]
 [-0.21036812]]
Cost with optimized theta:  0.5083372029732048
==============================
==============================
alpha :  0.005
Theta_initial :  [[0.]
```

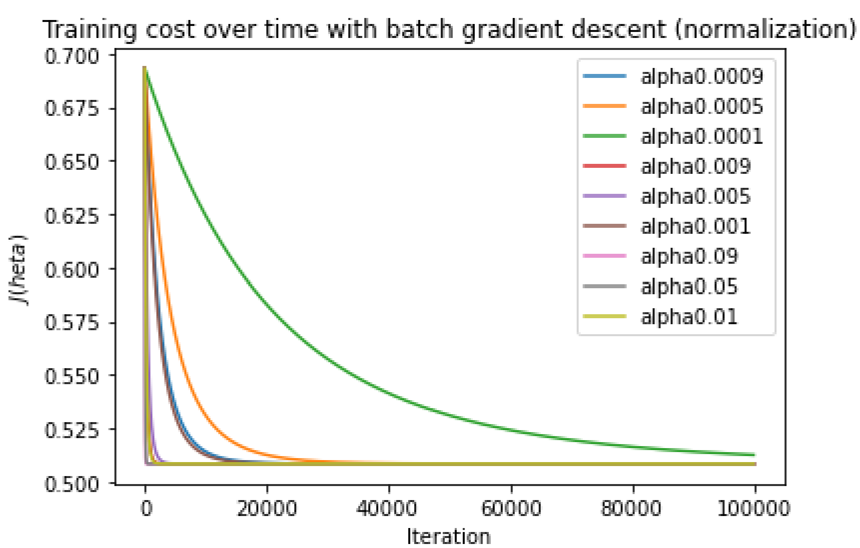
```
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Theta optimized :  [[-0.91068977]
 [-0.02302702]
 [-0.22616074]
 [ 0.08748206]
 [ 0.16360724]
 [ 0.05051176]
 [-0.00347998]
 [ 0.13211717]
 [ 0.04619396]
 [ 0.05971734]
 [-0.95475721]
 [-0.21036812]]
Cost with optimized theta:  0.5083372029732048
==============================
==============================
alpha :  0.001
Theta_initial :  [[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Theta optimized :  [[-0.91068228]
 [-0.0230078 ]
 [-0.2261278 ]
 [ 0.08746579]
 [ 0.1636092 ]
 [ 0.05051395]
 [-0.00344996]
 [ 0.13213039]
 [ 0.04616215]
 [ 0.05972335]
 [-0.95475311]
 [-0.21036762]]
Cost with optimized theta:  0.5083372031115233
==============================
==============================
alpha :  0.09
Theta_initial :  [[0.]
 [0.]
```

```
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Theta optimized :  [[-0.91068977]
 [-0.02302702]
 [-0.22616074]
 [ 0.08748206]
 [ 0.16360724]
 [ 0.05051176]
 [-0.00347998]
 [ 0.13211717]
 [ 0.04619396]
 [ 0.05971734]
 [-0.95475721]
 [-0.21036812]]
Cost with optimized theta:  0.5083372029732048
==============================
==============================
alpha :  0.05
Theta_initial :  [[0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Theta optimized :  [[-0.91068977]
 [-0.02302702]
 [-0.22616074]
 [ 0.08748206]
 [ 0.16360724]
 [ 0.05051176]
 [-0.00347998]
 [ 0.13211717]
 [ 0.04619396]
 [ 0.05971734]
 [-0.95475721]
 [-0.21036812]]
Cost with optimized theta:  0.5083372029732048
==============================
==============================
alpha :  0.01
Theta_initial :  [[0.]
 [0.]
 [0.]
```

```
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
Theta optimized :  [[-0.91068977]
 [-0.02302702]
 [-0.22616074]
 [ 0.08748206]
 [ 0.16360724]
 [ 0.05051176]
 [-0.00347998]
 [ 0.13211717]
 [ 0.04619396]
 [ 0.05971734]
 [-0.95475721]
 [-0.21036812]]
Cost with optimized theta:  0.5083372029732047
==============================
==============================
```

In [52]:

```
for i in range(len(j_history_list)):
    plt.plot(j_history_list[i], label='alpha' + str(alpha_list[i]))
    plt.xlabel('Iteration')
    plt.ylabel("$J(\theta)$")
    plt.title("Training cost over time with batch gradient descent (normalization)")
    plt.legend()
plt.show()
```



In [53]:

```
y_test_pred = h_theta(X_test, theta)
y_pred = np.round(y_test_pred)
print(y_pred.T)
```

```
[[1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0.
  1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.
  1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.
  0. 1. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.
  1. 1. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.
  0. 0. 0.]]
```

In [54]:

```
from sklearn.metrics import classification_report
print("======= Classification report ========")
print(classification_report(y_test, y_pred))
```

```
======= Classification report ========
              precision    recall  f1-score   support

           0       0.75      0.88      0.81        85
           1       0.57      0.34      0.43        38

    accuracy                           0.72       123
   macro avg       0.66      0.61      0.62       123
weighted avg       0.69      0.72      0.69       123
```

In [55]:

```
y_test_pred_soft = h_theta(X_test, theta)
y_test_pred_hard = (y_test_pred_soft > 0.5).astype(int)

test_rsq_soft = r_squared(y_test, y_test_pred_soft)
test_rsq_hard = r_squared(y_test, y_test_pred_hard)
test_acc = (y_test_pred_hard == y_test).astype(int).sum() / y_test.shape[0]

print('Got test set soft R^2 %0.4f, hard R^2 %0.4f, accuracy %0.2f' % (test_rsq_soft, test
_rsq_hard, test_acc))
```

```
Got test set soft R^2 0.0846, hard R^2 -0.3328, accuracy 0.72
```

In [56]:

```
y_train_pred_soft = h_theta(X_train, theta)
y_train_pred_hard = (y_train_pred_soft > 0.5).astype(int)

train_rsq_soft = r_squared(y_train, y_train_pred_soft)
train_rsq_hard = r_squared(y_train, y_train_pred_hard)
train_acc = (y_train_pred_hard == y_train).astype(int).sum() / y_train.shape[0]

print('Got train set soft R^2 %0.4f, hard R^2 %0.4f, accuracy %0.2f' % (train_rsq_soft, tr
ain_rsq_hard, train_acc))
```

```
Got train set soft R^2 0.2352, hard R^2 0.0066, accuracy 0.79
```

Summary

First, all the null values in the training and test sets are filled with each unique values according to the ratio in the category columns. For numeric columns, the null values are filled with mean vales of each variable. After filling all null values, "loan_Id" column is dropped because it has no useful to the model and then each columns is ploted to see the distribution. Next, each categorical values are converted into float (0.0, 1.0, etc.). Then the training data is splitted into X and y ("Loan_status"). Then the data X was normailized and inserted 1 in the first column. The data X and y was splitted into training set and test set with training set ratio 80%. The training set was trained with various alpha values and among these values 0.01 has the lowest cost values but the cost difference between 0.01 and both 0.05, 0.09 is only 0.0000000000000001. The training set has accuracy 0.79 and the test set has accuracy 0.72.

In [57]:

```
data_test.shape
```

Out[57]:

```
(367, 11)
```

In [58]:

```
X_T = data_test
X_T = data_Norm(X_T)
X_T = np.array(X_T)

X_T = np.insert(X_T, 0, 1, axis=1)

y_T = h_theta(X_T, theta)
y_T_pred = (y_T > 0.5).astype(int)
print(y_T_pred.T)
```

```
[[0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 1 0 1 0 0
  0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 1 0 0 1 0 1 0
  0 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1
  0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 1 1 0 1 0 0 0 0 1 1 0 0 1 0 1
  0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0
  0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 1 0 1 0 0 0 0 1 0
  0 0 0 1 0 0 0 1 0 0 1 0 0 1 1 0 1 0 0 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 1 0
  0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0
  0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 1 0
  1 0 0 0 1 0 0]]
```

In [ ]:

```

```