03-Logistic-Regression

September 6, 2021

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

[1]:
```python
NAME = "Nutapol Thungpao"
ID = "122148"
```

---

1 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 $y \in \{0, 1\}$.

1.1 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 $\mathbf{x}$:

$$y \sim \mathcal{N}(\theta^\top \mathbf{x}, \sigma^2)$$

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 $\mathbf{x}$?

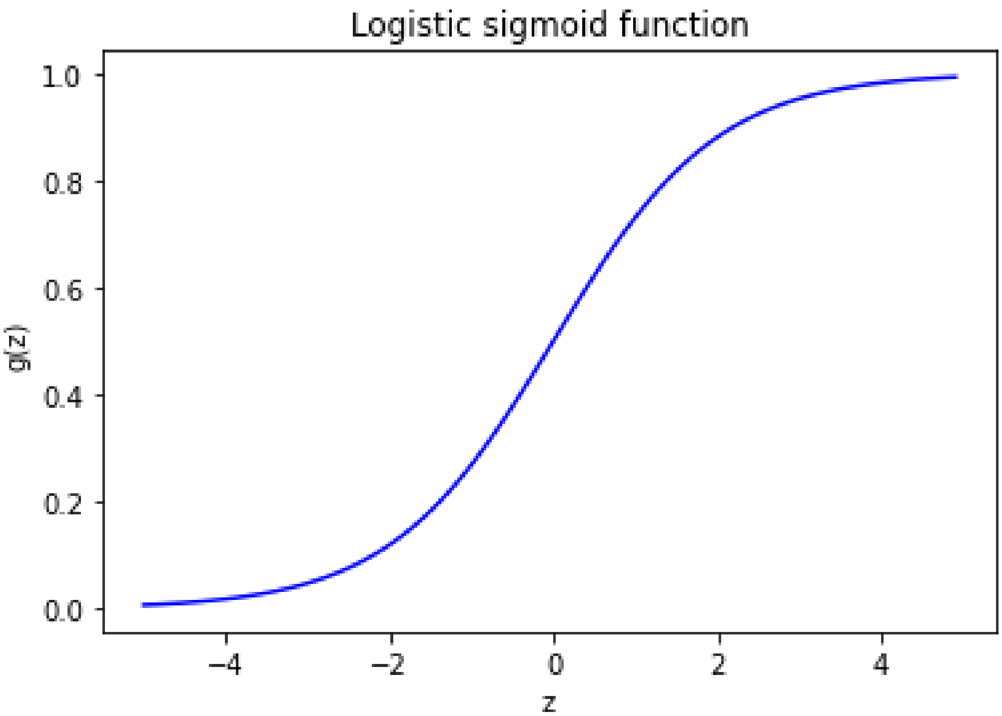
In linear regression, we assume that the mean of the Gaussian is $\theta^\top \mathbf{x}$, i.e., a linear function of $\mathbf{x}$.

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 $0 \leq p \leq 1$. The sigmoid function $g(\cdot)$ conveniently obeys these bounds:

[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 $(\mathbf{x}, y) \in \mathbb{R}^n \times \{0, 1\}$.

2. The *hypothesis function* is $h_\theta(\mathbf{x}) = \frac{1}{1+e^{-\theta^\top \mathbf{x}}}$.

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

1.2 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

$$\ell(\theta) = \log \mathcal{L}(\theta) = \sum_{i=1}^{m} y^{(i)} \log(h_\theta(\mathbf{x}^{(i)})) + (1 - y^{(i)}) \log(1 - (h_\theta(\mathbf{x}^{(i)})).$$

Negating the log likelihood function to obtain a loss function, we have

$$J(\theta) = -\sum_{i=1}^{m} y^{(i)} \log h_\theta(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(\mathbf{x}^{(i)})).$$

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:

$$\theta^{(j+1)} \leftarrow \theta^{(j)} + \alpha(y^{(i)} - h_\theta(\mathbf{x}^{(i)}))\mathbf{x}^{(i)}.$$

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.

1.3 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).

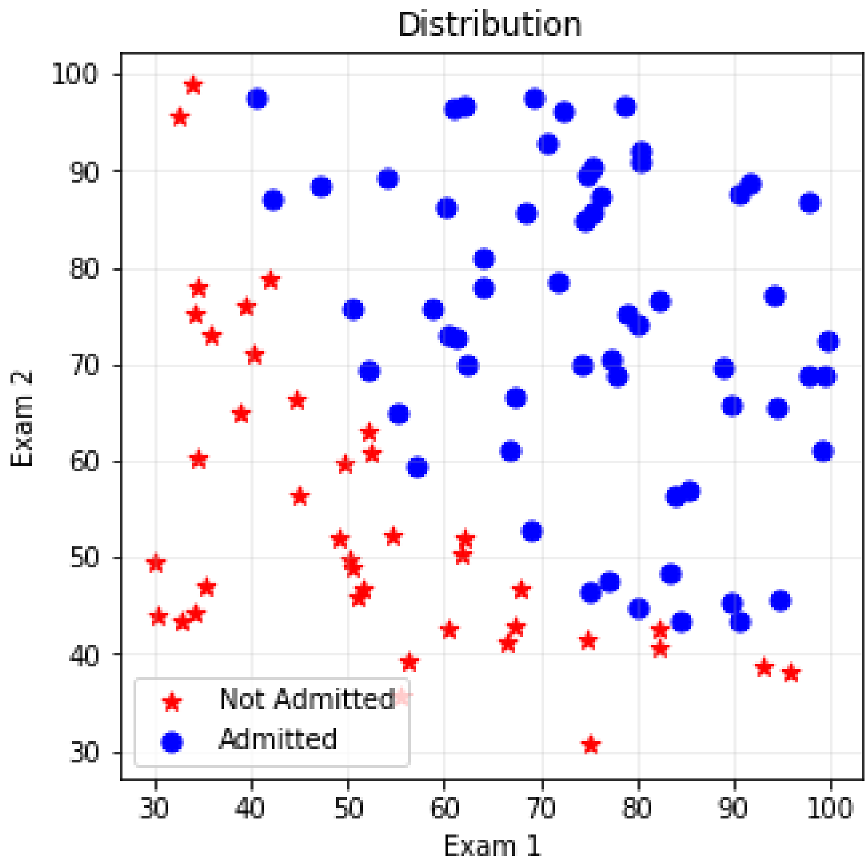
[3]:
```python
# Load student admissions data. The data file does not contain headers,
# so we use hard coded indices for exam 1, exam2, and the admission decision.

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...

[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 Admitted')
ax.scatter(exam1_data[idx_1], exam2_data[idx_1], s=50, c='b', marker='o',␣
 ↪label='Admitted')
plt.legend()
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...

[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];
```

1.3.1 Important functions needed later

Let's put all of our important functions here…

[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
```

1.3.2 Initialize theta

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

[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.]]
```

[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)
    print(i)
    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()
    return theta, j_history
```

1.3.3 Training function

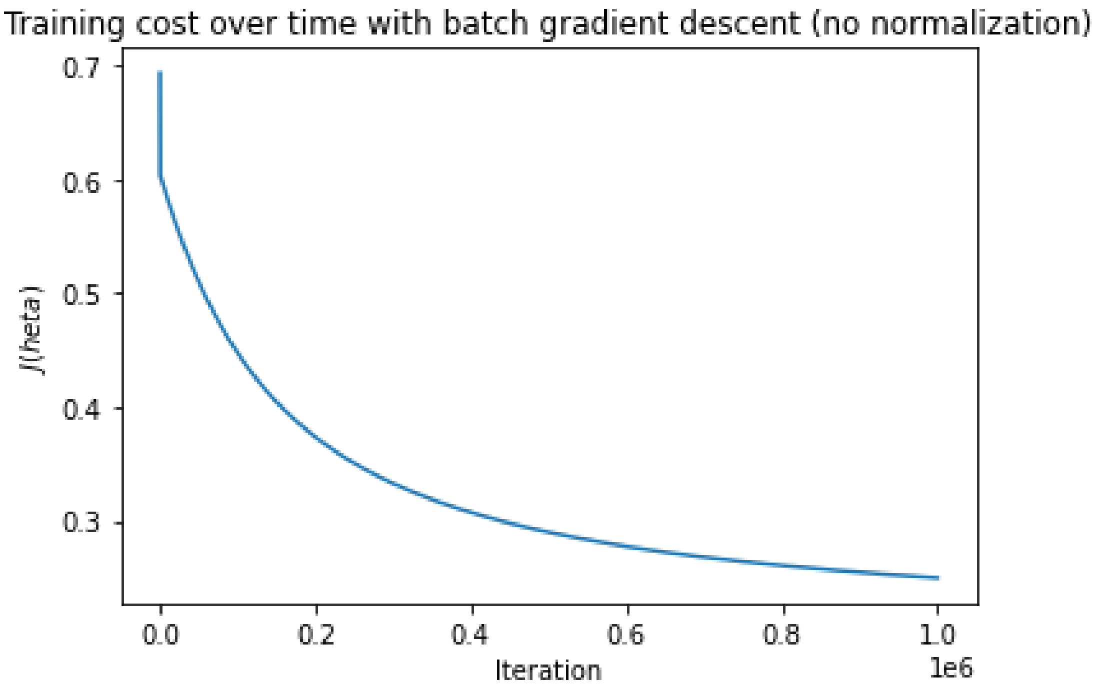
Here's a function to do batch training for `num_iters` iterations.

1.3.4 Do the training

Here we run the training function for a million batches!

[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])
```

```
999999
```

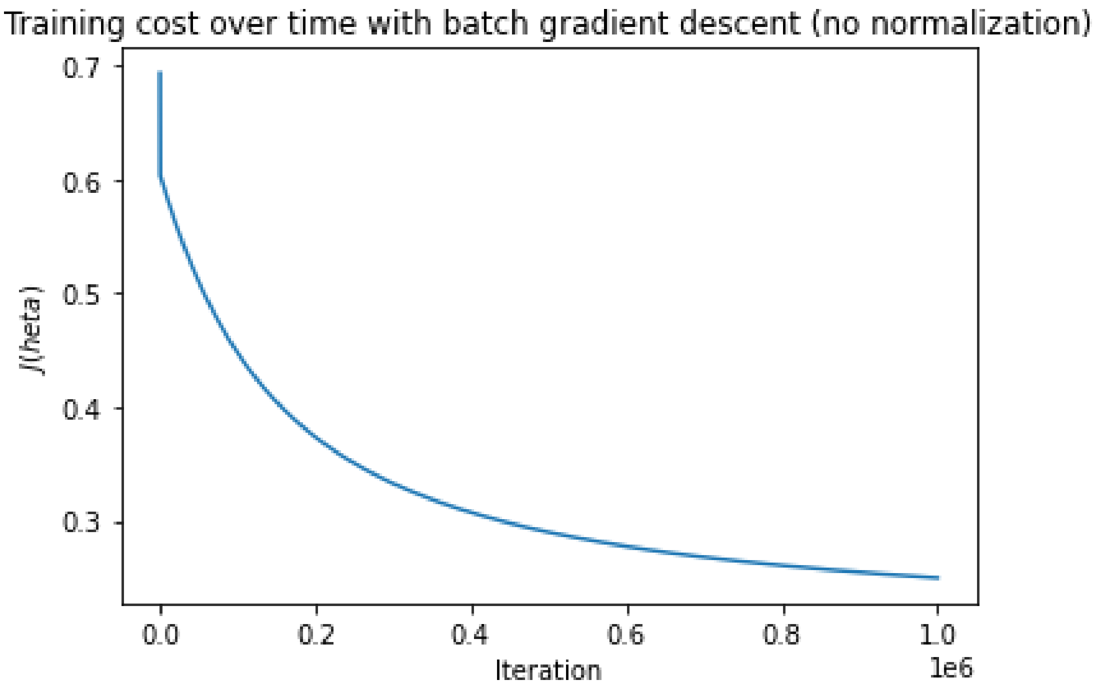



```
Theta optimized: [[-11.29380461]
 [  0.10678604]
 [  0.07994591]]
Cost with optimized theta: 0.24972975869900035
```

1.3.5 Plot the loss curve

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

[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()
```



1.3.6 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.

1.3.7 Exercise 1.1 (5 points)

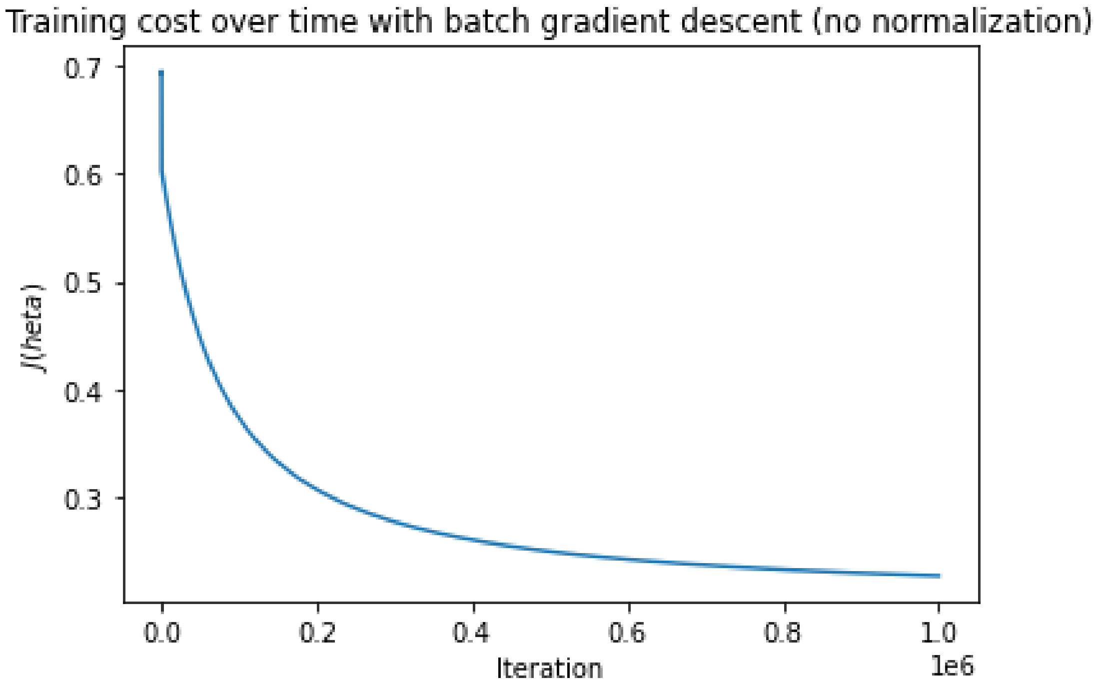
Fill in two different values for α and θ.

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

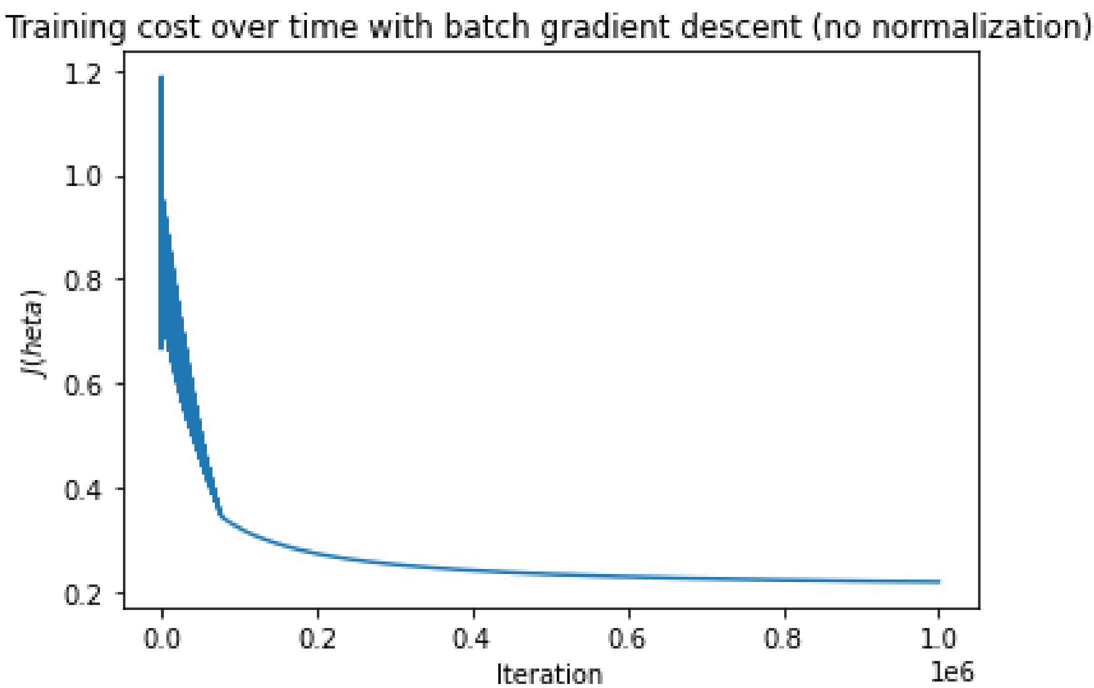
[11]:
```
# grade task: change 'None' value to number(s) or function
def trainI(X, y, theta_initial, alpha, num_iters):
    theta = theta_initial
    j_history = []
```

```
    cost_old=100000
    for i in range(num_iters):
        cost, grad = j(theta, X, y)
        theta = theta + alpha * grad
        deff=np.ans(cost_old-cost)
        if deff < 0.001:
            break
        cost_old=cost_old
        j_history.append(cost)
    print(i)
    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()
    return theta, j_history

# Train for 1000000 iterations on full training set
num_iters = 1000000


# declare your alphas
# alpha1 = None
alpha1 = .001
alpha2 = .0015
theta1, j_history1 = train(X_train, y_train, theta_initial, alpha1, num_iters)
theta2, j_history2 = train(X_train, y_train, theta_initial, alpha2, num_iters)

# alpha2 = None

# initialize thetas as you want
theta_initial1 = theta1
theta_initial2 = theta2


# define your num iterations
# num_iters = None
```

999999


Training cost over time with batch gradient descent (no normalization)
0.7
0.6
0.5
0.4
0.3
J(heta)
0.0
0.2
0.4
0.6
0.8
1.0
Iteration
1e6

999999


Training cost over time with batch gradient descent (no normalization)
1.2
1.0
0.8
0.6
0.4
0.2
J(heta)
0.0
0.2
0.4
0.6
0.8
1.0
Iteration
1e6

[12]:
```python
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 filled"
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,␣
↪"initialized 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: 0.001
alpha 2: 0.0015
theta 1: [[-14.58284092]
 [  0.13414141]
 [  0.10526915]]
theta 2: [[-16.51461854]
 [  0.15046486]
 [  0.12009075]]
Use num iterations: 1000000
success!
```

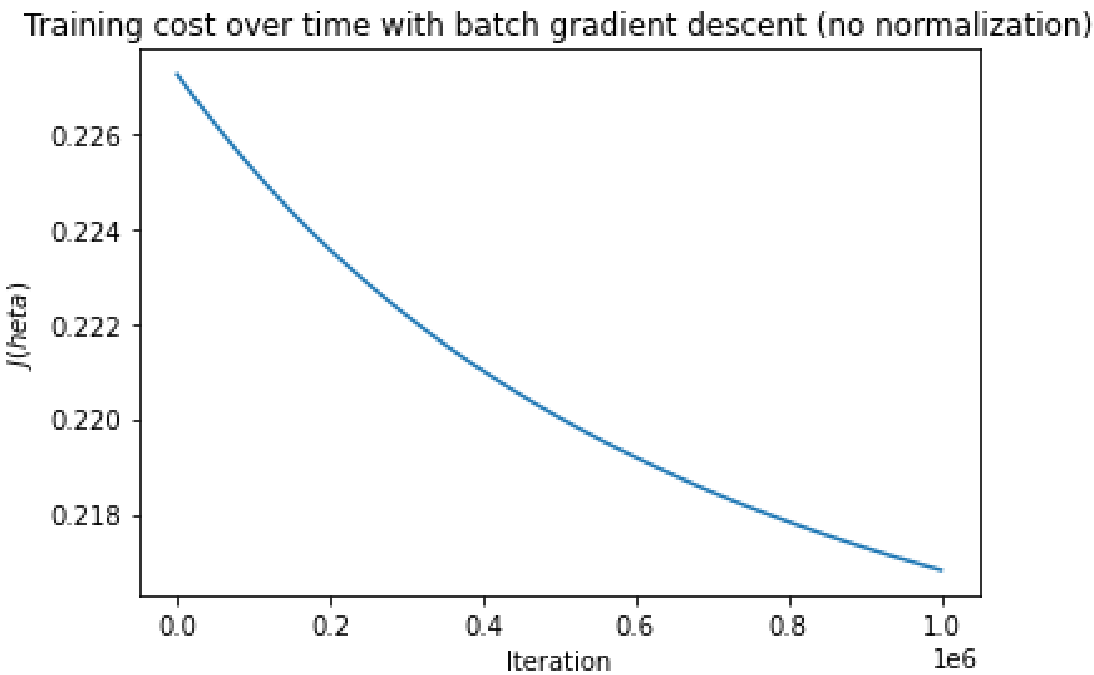
1.3.8 Exercise 1.2 (5 points)

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

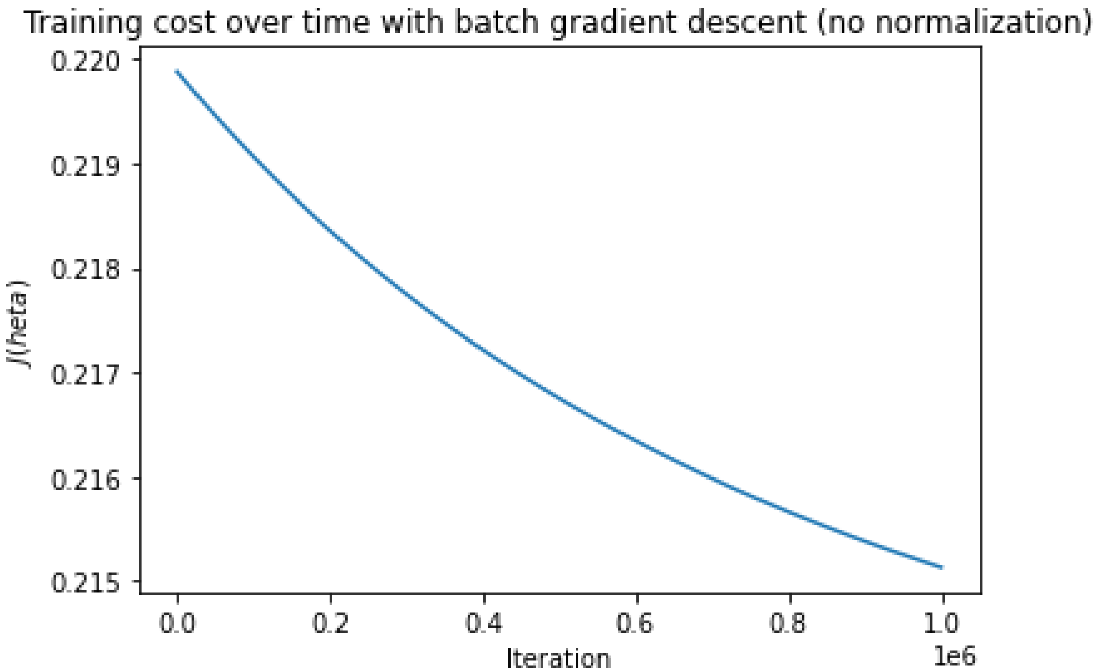
[13]:
```python
# grade task: change 'None, None' value to number(s) or function
j_history_list = []
theta_list = []
```

```
for alpha in alpha_list:
    for theta_initial in theta_initial_list:
        # YOUR CODE HERE
        theta_i, j_history_i = train(X_train, y_train, theta_initial, alpha,␣
↪num_iters)
        # theta_i, j_history_i = None, None
        j_history_list.append(j_history_i)
        theta_list.append(theta_i)
```

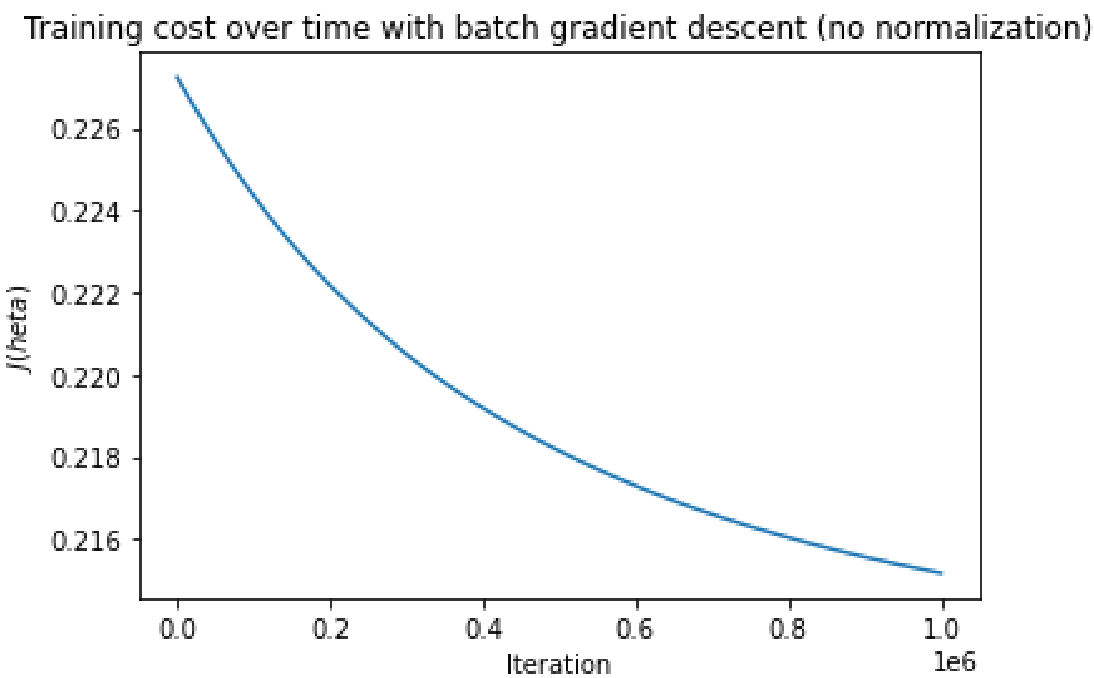
999999



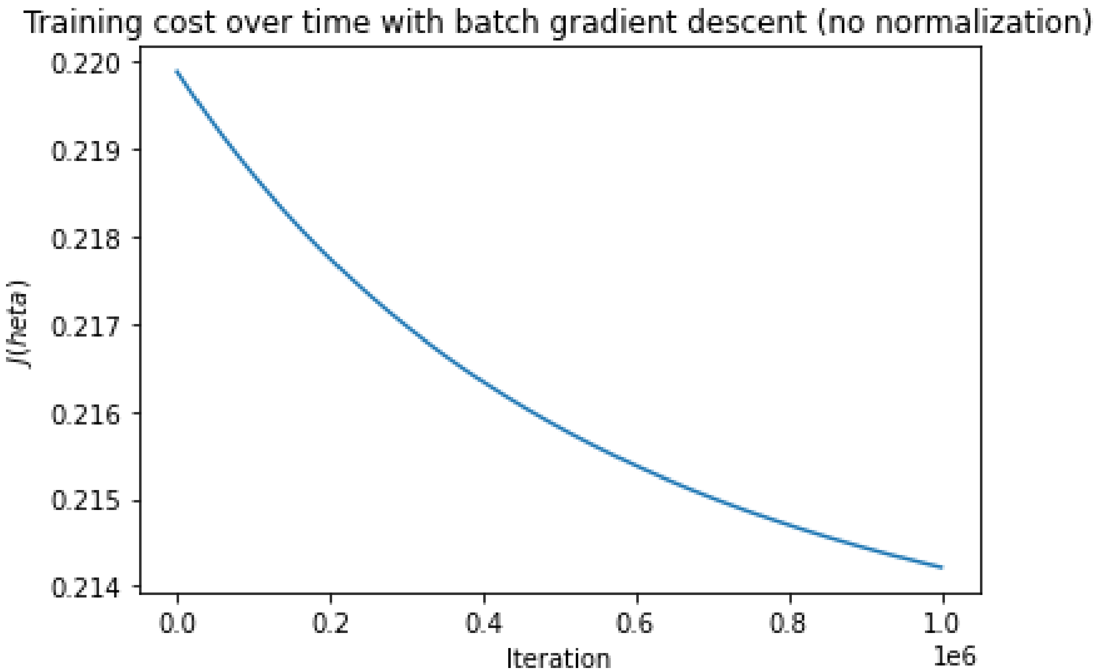
999999



999999



999999



[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_list 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!
```

1.3.9 Exercise 1.3 (10 points)

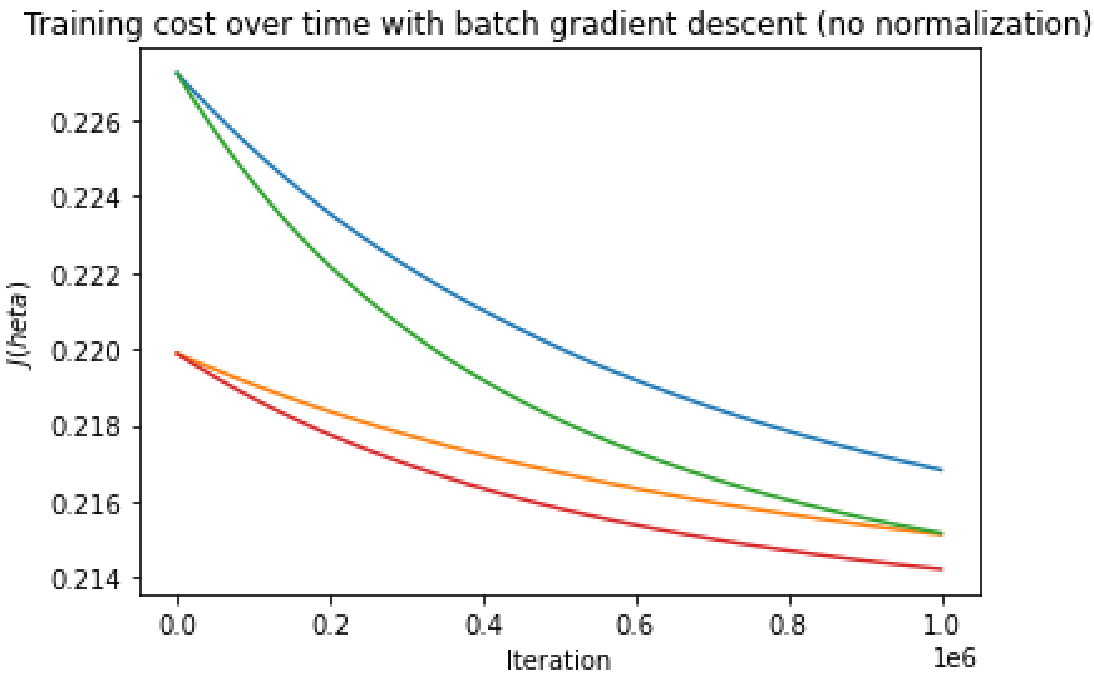
Write code to plot loss curves for each of the sequences in `j_history_list` from the previous exercise:

[15]:
```
np.array(j_history_list).shape
```

[15]:
```
(4, 1000000)
```

[16]:
```
plt.plot(j_history_list[0])
plt.plot(j_history_list[1])
plt.plot(j_history_list[2])
plt.plot(j_history_list[3])
```

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



1.3.10 Exercise 1.4 (10 points)

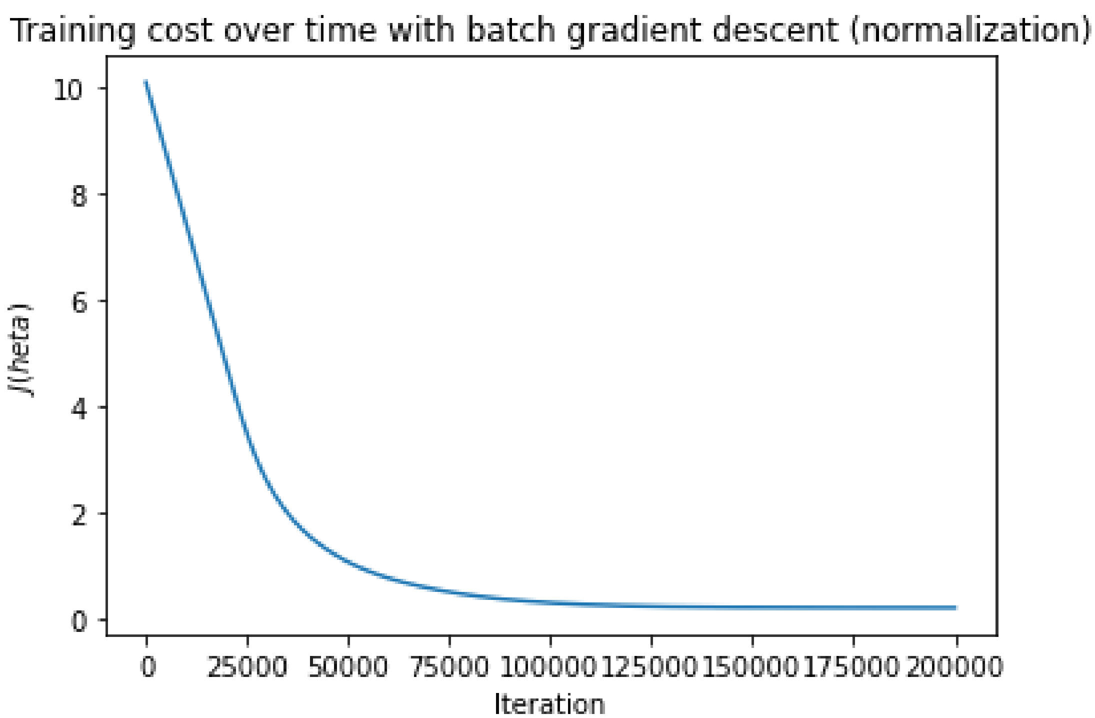
- Repeat your training, but **normalize** your data before training
- Compare the results between normalized data and unnormalized data

[17]:
```
# code here
means = np.mean(data, axis=0)
stds = np.std(data, axis=0)
data_norm = (data - means) / stds
print(data_norm.shape)

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

def train(X, y, theta_initial, alpha, num_iters):
    theta = theta_initial
    j_history = []
```

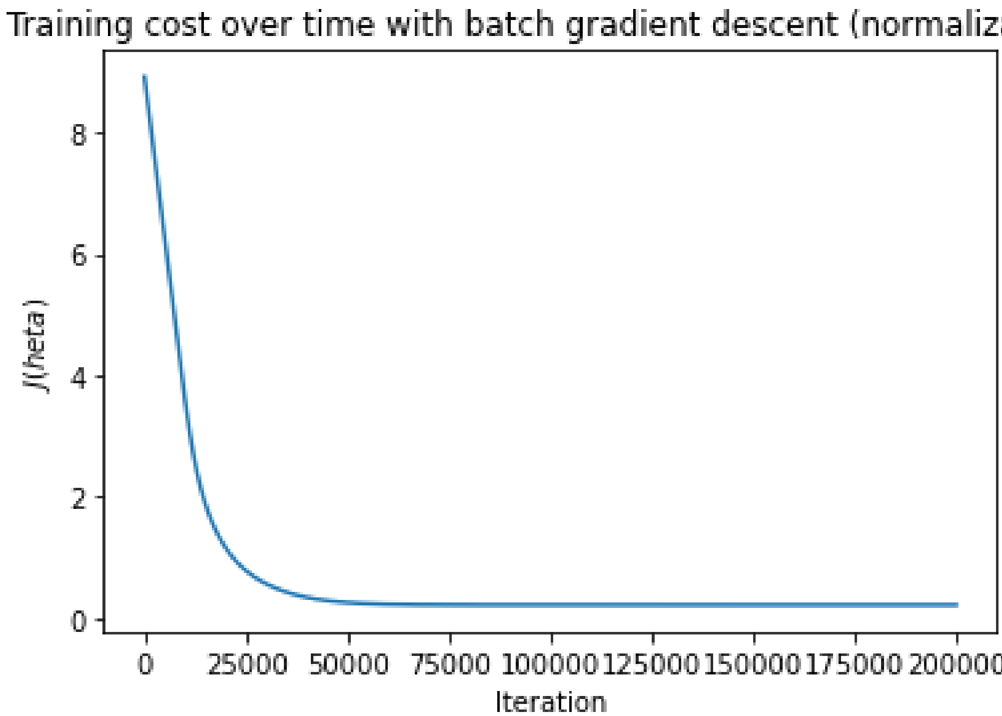
```
    for i in range(num_iters):
        cost, grad = j(theta, X, y)
        theta = theta + alpha * grad
        j_history.append(cost)
    print(i)
    plt.plot(j_history)
    plt.xlabel("Iteration")
    plt.ylabel("$J(\theta)$")
    plt.title("Training cost over time with batch gradient descent␣
 ↪(normalization)")
    plt.show()
    return theta, j_history

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];

# Train for 1000000 iterations on full training set

alpha = .0005
num_iters = 200000
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])
```

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

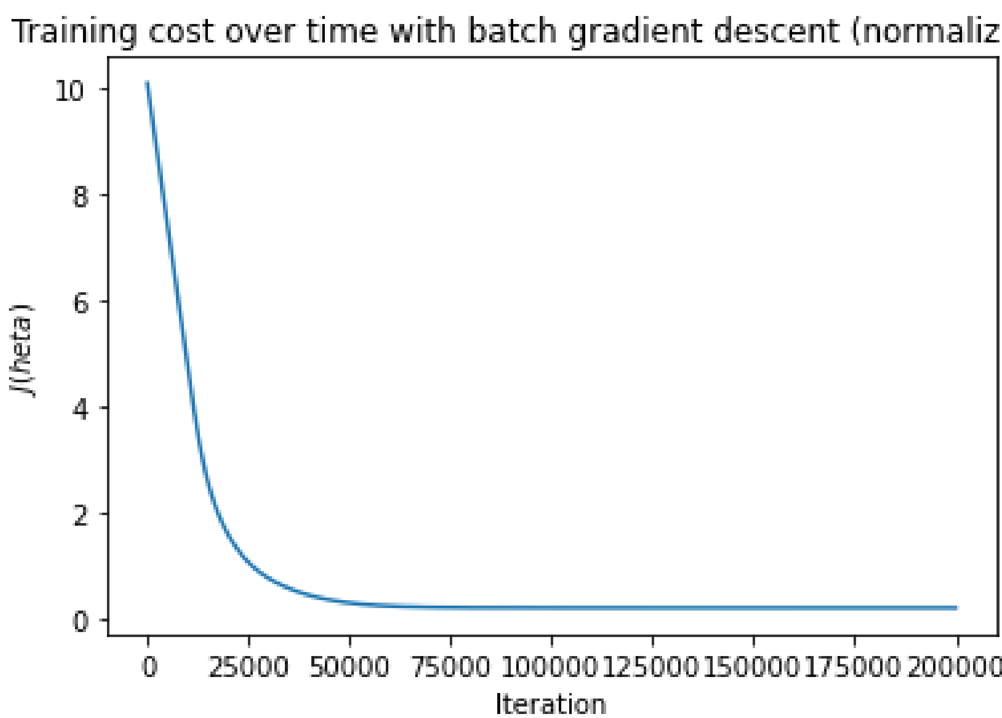
```
(100, 3)
199999
```



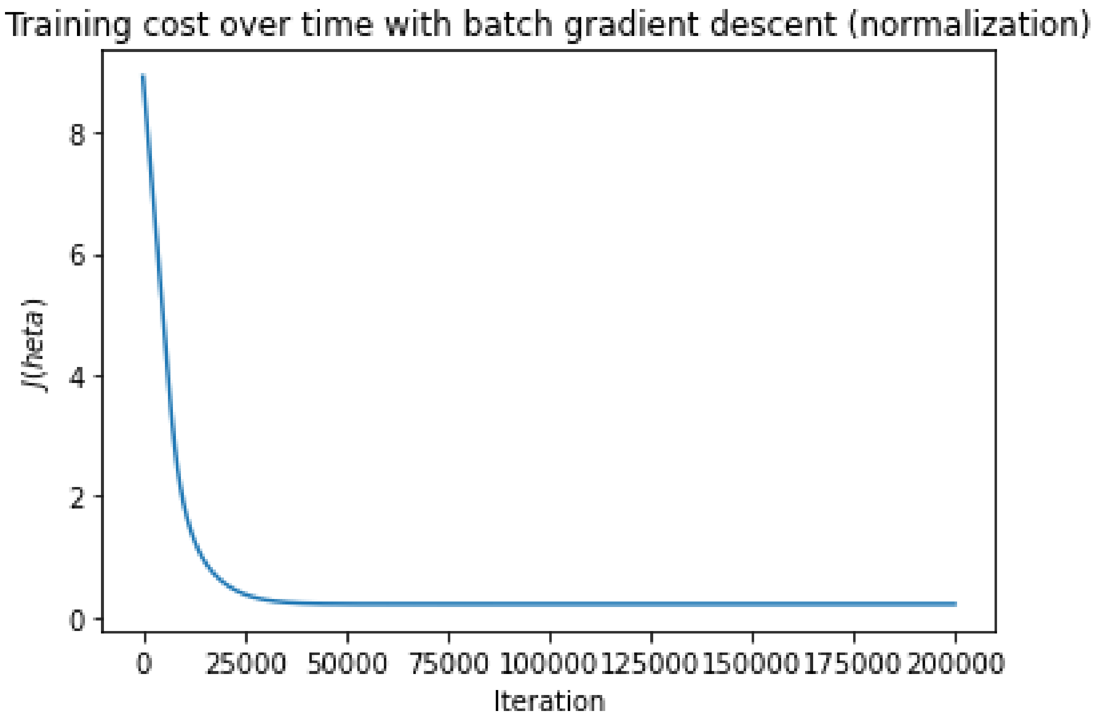

```
Theta optimized: [[1.85026199]
 [4.2887002 ]
 [3.4128576 ]]
Cost with optimized theta: 0.2142322289127974
199999
```


Training cost over time with batch gradient descent (normalization)
J(heta)
Iteration
0
2
4
6
8
0
25000
50000
75000
100000
125000
150000
175000
200000

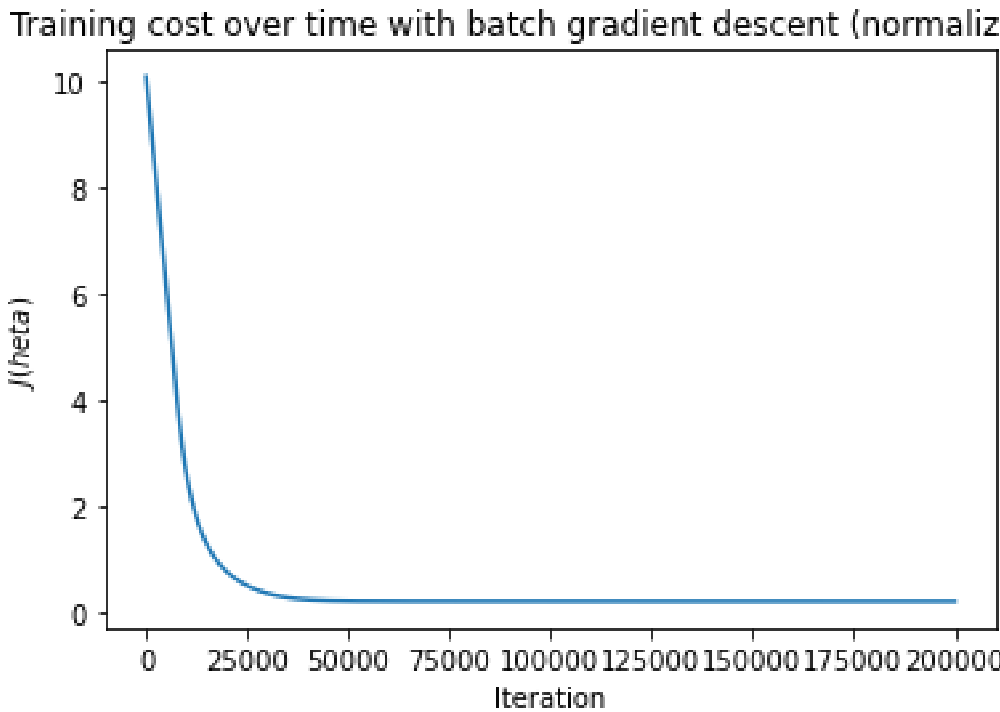
199999


Training cost over time with batch gradient descent (normalization)
J(heta)
Iteration
0
2
4
6
8
10
0
25000
50000
75000
100000
125000
150000
175000
200000

199999



199999



1.3.11 Exercise 1.5 (5 points)

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

Write your discussion here.

1.3.12 The logistic regression decision boundary

Note that when $\theta^\top \mathbf{x} = 0$, we have $h_\theta(\mathbf{x}) = 0.5$. That is, we are equally unsure as to whether $\mathbf{x}$ belongs to class 0 or class 1. The contour at which $h_\theta(\mathbf{x}) = 0.5$ 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

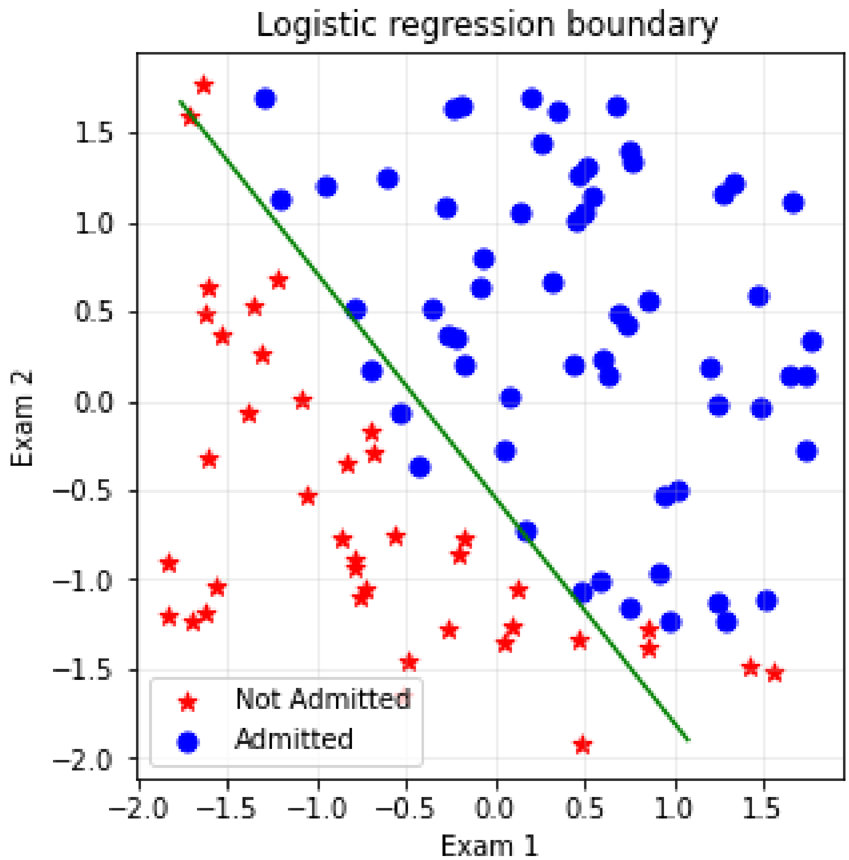
$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$$

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.

[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
```

[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.

1.3.13 Test set performance

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

[20]:
```
def r_squared(y, y_pred):
    return 1 - np.square(y - y_pred).sum() / np.square(y - y.mean()).sum()
```

[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 0.7447, hard R^2 0.6931, accuracy 0.93
```

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

1.4 Example 2: Loan prediction dataset

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

This dataset is from Kaggle.

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.

1.4.1 Read the data and take a look at it

[22]:
```
# Import Pandas. You may need to run "pip3 install pandas" at the console if␣
↪it's not already installed

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# 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]
```

[23]:
```python
# Check for missing values in the training and test data

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

1.4.2 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. It's great!

[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().sum())
```

```
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:

[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.

[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
```

1.5 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 **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)

1.6 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.

For this Dataset, It still have null information in columns Gender, Self_Employed, Loan_Amount_Term and Credit_History. Then I fill them by add same information but do not or less impact to avg of data. Affter that, I set columns drop columns of Loan_ID and Property_Area because Loan_ID it is just identified data from who and Property_Area it is name of Area that is str type that may impact that i don't know how to change to value and do not impact to datasets. Affer that, I change gender colums from Male and Female to be 1 and 2, Self_Employed colums from Yes and No to be 1 and 0 , Education colums from Graduate and Not Graduate to be 1 and 0, Married colums from Yes and No to be 1 and 0 and Loan_Status colums from Y and N to be 1 and 0.
Next, I set columns Gender, Married, Dependents, Education, Self_Employed, ApplicantIncome, CoapplicantIncome, LoanAmount, Loan_Amount_Term and Credit_History to be X Train and Loan_Status to be Y train. Affter that, I normalization data for help about calculater faster by sklearn.preprocessing.normalize function. Affter that, I train with alpha1 = .0001 alpha2 = .00005. I get best parameters = [[4.00496168e-01],[3.28510362e-01],[4.13908503e-03],[9.49288206e-05],[9.18202284e-05],[-1.26620534e-04],[1.27662685e-04],[9.69181757e-06],[2.43066237e-01],[2.43152113e-02],[4.09699423e-04]]. I got model's got test set soft R^2 0.0020, hard R^2 -0.4386, accuracy 0.70.. Then make predeced Loan_Status from test set and change out put 1,0 to be Y and N.

[27]:

```
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
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)
    print(i)
    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()
    return theta, j_history

def fill_na(columm_name):
    #num_train=train.value_counts(columm_name)
    num_train=pd.value_counts(data_train[columm_name].values.flatten()).sum()
    Xtrain=pd.value_counts(data_train[columm_name].values.flatten())

    #num_test = test.value_counts(columm_name)
    num_test=pd.value_counts(data_test[columm_name].values.flatten()).sum()
    #print(num_test)
    value_list = list(Xtrain.index)
    for value in value_list:

        ratio_test=num_test/data_test[columm_name].shape[0]
        ratio_test=float(ratio_test)
        num_test = round(ratio_test * data_test[columm_name].isnull().sum())

        if num_test > 0:
```

```
                data_test[column_name].fillna(value, inplace = True, limit =␣
↪num_test)

        ratio_train=num_train/data_train[column_name].shape[0]
        ratio_train=float(ratio_train)

        num_train = round(ratio_train * data_train[column_name].isnull().sum())

        if num_train > 0:
                data_train[column_name].fillna(value, inplace = True, limit =␣
↪num_train)

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)
    print(i)
    plt.plot(j_history)
    plt.xlabel("Iteration")
    plt.ylabel("$J(\theta)$")
    plt.title("Training cost over time with batch gradient descent␣
↪(normalization)")
    plt.show()
    return theta, j_history
```

[28]:
```
print('Missing values for train data:\n------------------------\n', data_train.
↪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
```

[29]:
```
fill_na('Gender')
```

[30]:
```
fill_na('Self_Employed')
fill_na('Self_Employed')
```

[31]:
```
fill_na('Credit_History')
fill_na('Credit_History')
```

[32]:
```
fill_na('Loan_Amount_Term')
```

[33]:
```
print('Missing values for train data:\n------------------------\n', data_train.
↪isnull().sum())
```

```
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
```

[34]:
```
data_train
```

[34]:
```
      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          4      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  146.412162             360.0
```

```
1            4583          1508.0  128.000000             360.0
2            3000             0.0   66.000000             360.0
3            2583          2358.0  120.000000             360.0
4            6000             0.0  141.000000             360.0
..             …               …          …             …
609          2900             0.0   71.000000             360.0
610          4106             0.0   40.000000             180.0
611          8072           240.0  253.000000             360.0
612          7583             0.0  187.000000             360.0
613          4583             0.0  133.000000             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]
```

[35]:

```
gender = {'Male': 1,'Female': 2}
data_train.Gender = [gender[item] for item in data_train.Gender]
data_test.Gender = [gender[item] for item in data_test.Gender]
```

[36]:

```
Education = {'Graduate': 1,'Not Graduate': 0}
data_train.Education = [Education[item] for item in data_train.Education]
data_test.Education = [Education[item] for item in data_test.Education]
```

[37]:

```
status = {'Yes': 1,'No': 0}
data_train.Self_Employed = [status[item] for item in data_train.Self_Employed]
data_test.Self_Employed = [status[item] for item in data_test.Self_Employed]
```

[38]:

```
status = {'Yes': 1,'No': 0}
data_train.Married = [status[item] for item in data_train.Married]
data_test.Married = [status[item] for item in data_test.Married]
```

[39]:

```
status = {'Y': 1,'N':0}
data_train.Loan_Status = [status[item] for item in data_train.Loan_Status]
#data_test.Loan_Status = [status[item] for item in data_test.Loan_Status]
```

[40]:
```
data_train=data_train.drop(columns=['Loan_ID', 'Property_Area'])
data_train = data_train.astype(int)
```

[41]:
```
data_train
```

[41]:
```
     Gender  Married  Dependents  Education  Self_Employed  ApplicantIncome  \
0         1        0           0          1              0             5849
1         1        1           1          1              0             4583
2         1        1           0          1              1             3000
3         1        1           0          0              0             2583
4         1        0           0          1              0             6000
..      …        …           …          …              …                …
609       2        0           0          1              0             2900
610       1        1           4          1              0             4106
611       1        1           1          1              0             8072
612       1        1           2          1              0             7583
613       2        0           0          1              1             4583

     CoapplicantIncome  LoanAmount  Loan_Amount_Term  Credit_History  \
0                    0         146               360               1
1                 1508         128               360               1
2                    0          66               360               1
3                 2358         120               360               1
4                    0         141               360               1
..                 …          …               …              …
609                  0          71               360               1
610                  0          40               180               1
611                240         253               360               1
612                  0         187               360               1
613                  0         133               360               0

     Loan_Status
0              1
1              0
2              1
3              1
4              1
..           …
609            1
610            1
611            1
612            1
613            0

[614 rows x 11 columns]
```

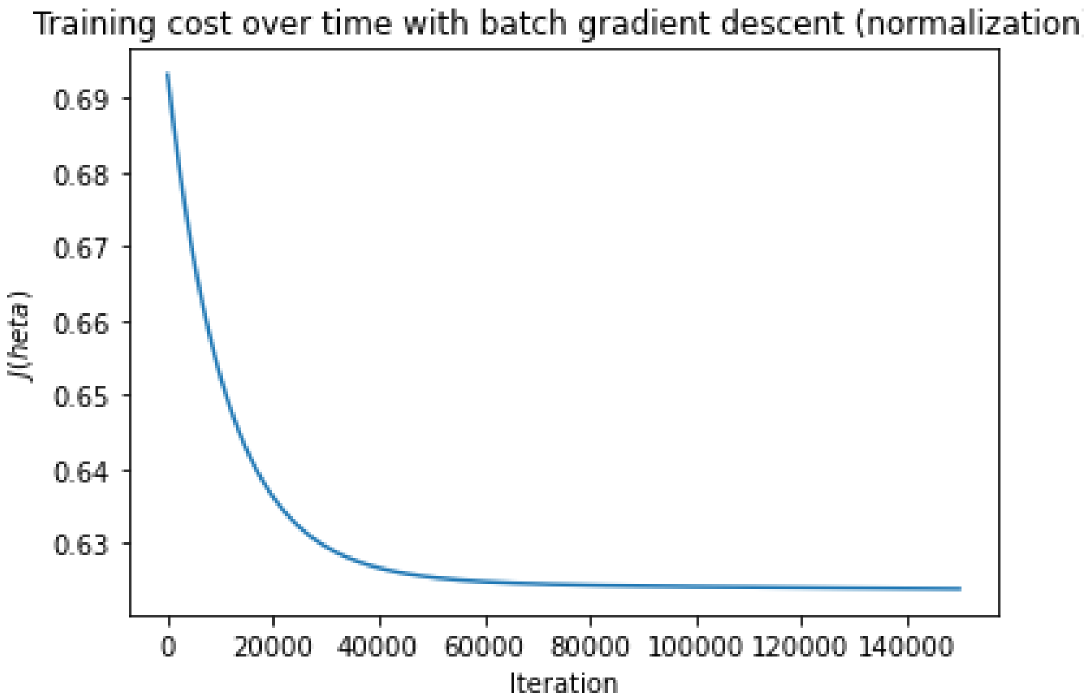
[42]:
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import normalize
X=data_train[['ApplicantIncome','LoanAmount','Gender','Married','Dependents','Education','Self
y= data_train['Loan_Status']
#scaler = normalize()
X = normalize(X)
y=y.to_numpy()

#X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

[43]:
```
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];
```

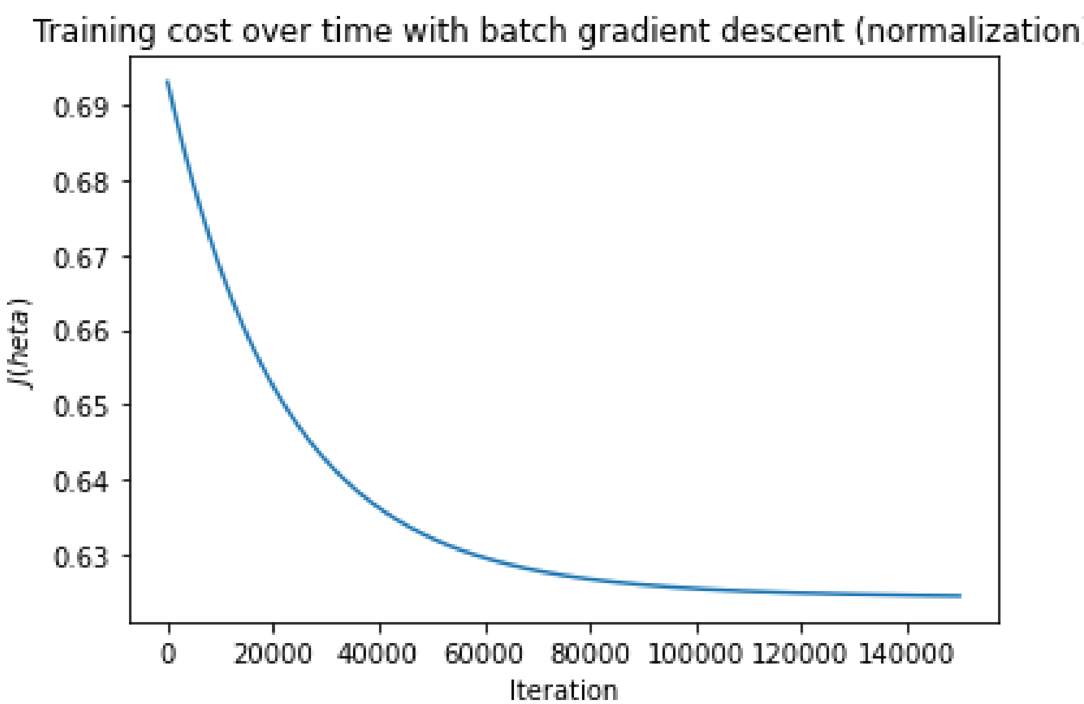
[44]:
```
# grade task: change 'None' value to number(s) or function
theta_initial = np.zeros((n+1, 1))
def trainI(X, y, theta_initial, alpha, num_iters):
    theta = theta_initial
    j_history = []
    cost_old=100000
    for i in range(num_iters):
        cost, grad = j(theta, X, y)
        theta = theta + alpha * grad
        deff=np.ans(cost_old-cost)
        if deff < 0.001:
            break
        cost_old=cost_old
```

```
        j_history.append(cost)
    print(i)
    plt.plot(j_history)
    plt.xlabel("Iteration")
    plt.ylabel("$J(\theta)$")
    plt.title("Training cost over time with batch gradient descent␣
↪normalization")
    plt.show()
    return theta, j_history

# Train for 1000000 iterations on full training set
num_iters = 150000


# declare your alphas
# alpha1 = None
alpha1 = .0001
alpha2 = .00005
theta1, j_history1 = train(X_train, y_train, theta_initial, alpha1, num_iters)
theta2, j_history2 = train(X_train, y_train, theta_initial, alpha2, num_iters)

# alpha2 = None

# initialize thetas as you want
theta_initial1 = theta1
theta_initial2 = theta2


# define your num iterations
# num_iters = None
```

```
149999
```


Training cost over time with batch gradient descent (normalization)
0.69
0.68
0.67
0.66
0.65
0.64
0.63
$J(heta)$
0
20000
40000
60000
80000
100000
120000
140000
Iteration

149999

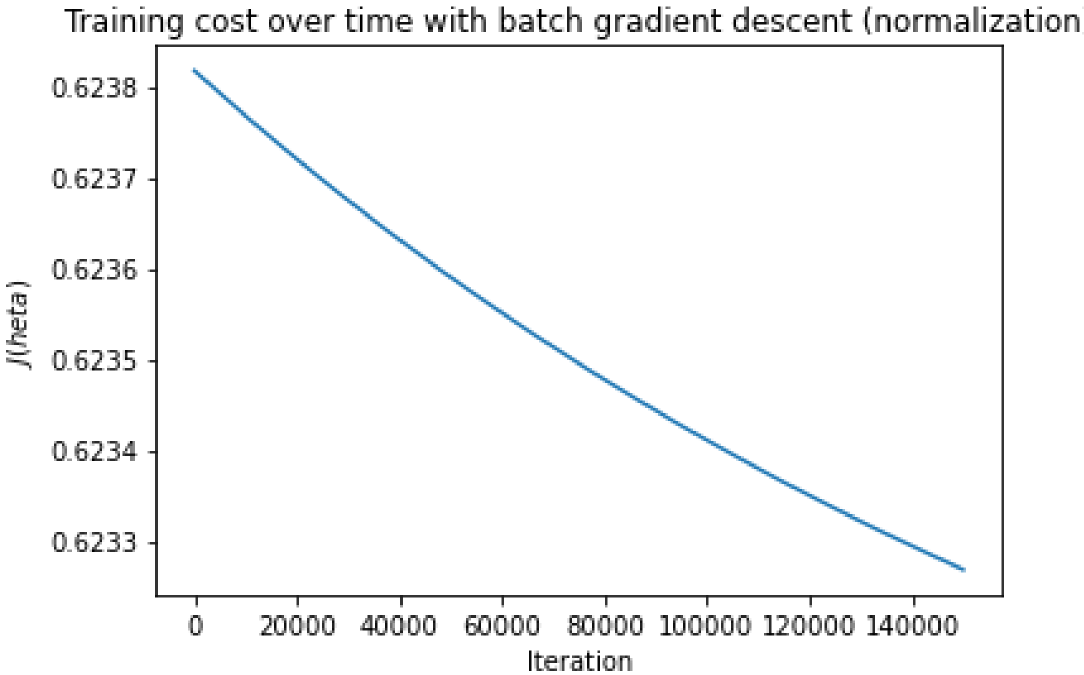

Training cost over time with batch gradient descent (normalization)
0.69
0.68
0.67
0.66
0.65
0.64
0.63
$J(heta)$
0
20000
40000
60000
80000
100000
120000
140000
Iteration

[45]:
```
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)
```

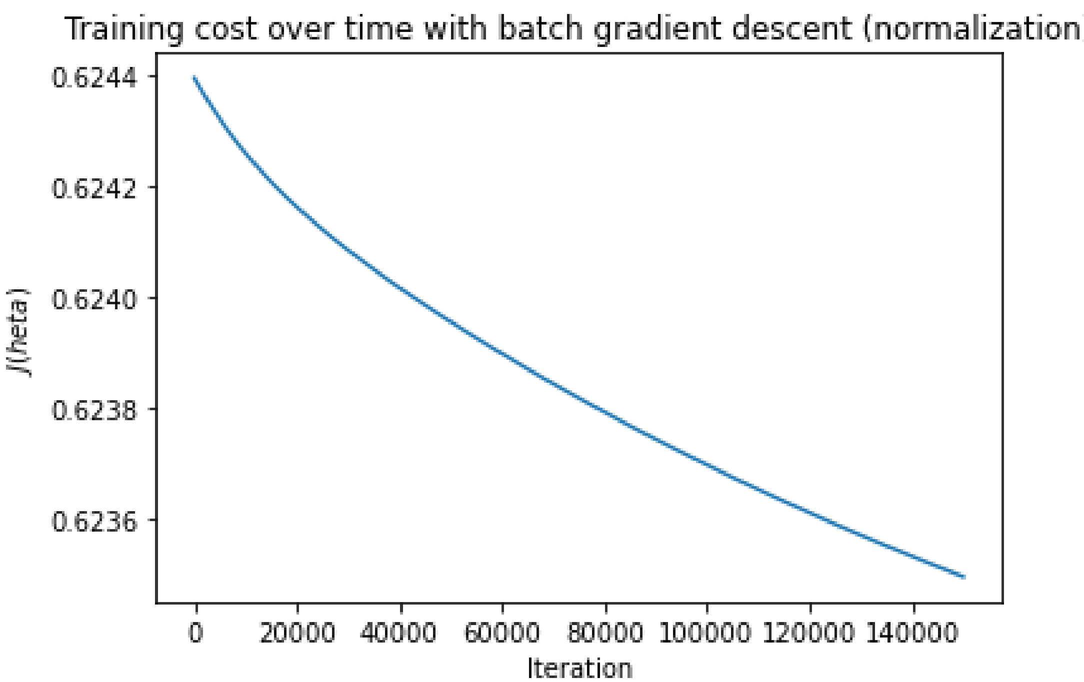
```
alpha 1: 0.0001
alpha 2: 5e-05
theta 1: [[ 4.00496225e-01]
 [ 3.28510381e-01]
 [ 4.13908738e-03]
 [ 9.49288360e-05]
 [ 9.18202451e-05]
 [-1.26620527e-04]
 [ 1.27662697e-04]
 [ 9.69181805e-06]
 [ 2.43066317e-01]
 [ 2.43152172e-02]
 [ 4.09699437e-04]]
theta 2: [[ 3.87579234e-01]
 [ 3.27762945e-01]
 [ 7.46273946e-03]
 [ 9.82513216e-05]
 [ 7.32590139e-05]
 [-3.01045947e-05]
 [ 9.61076979e-05]
 [ 8.99701558e-06]
 [ 1.81465213e-01]
 [ 2.69127116e-02]
 [ 2.41269957e-04]]
Use num iterations: 150000
```

[46]:
```
j_history_list = []
theta_list = []
for alpha in alpha_list:
    for theta_initial in theta_initial_list:
        # YOUR CODE HERE
        theta_i, j_history_i = train(X_train, y_train, theta_initial, alpha,␣
↪num_iters)
        # theta_i, j_history_i = None, None
        j_history_list.append(j_history_i)
        theta_list.append(theta_i)
```

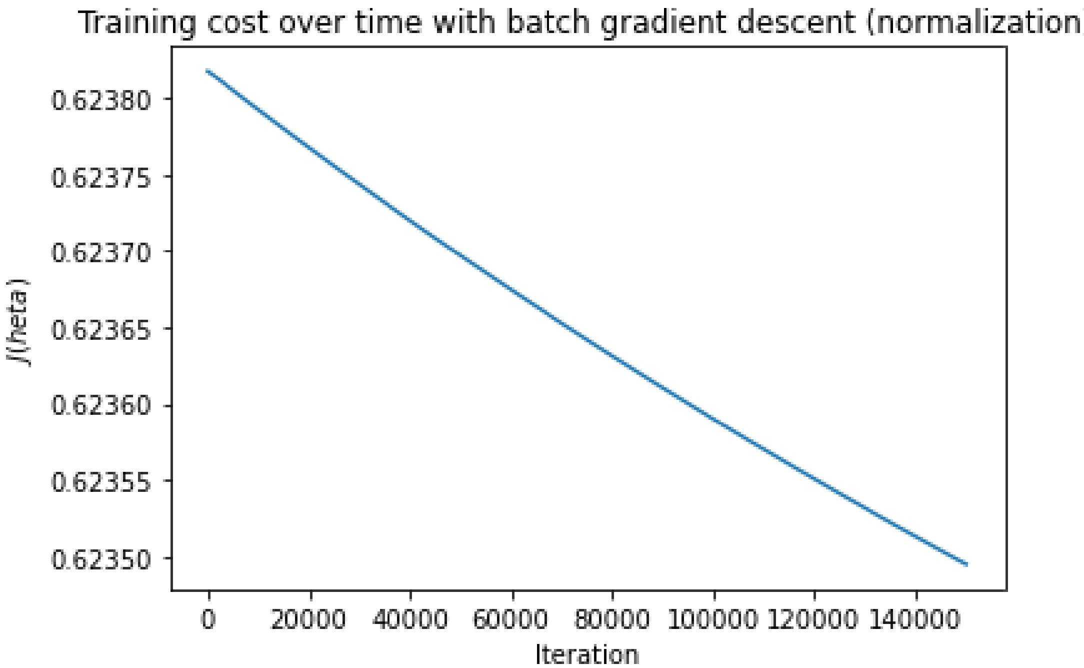
149999



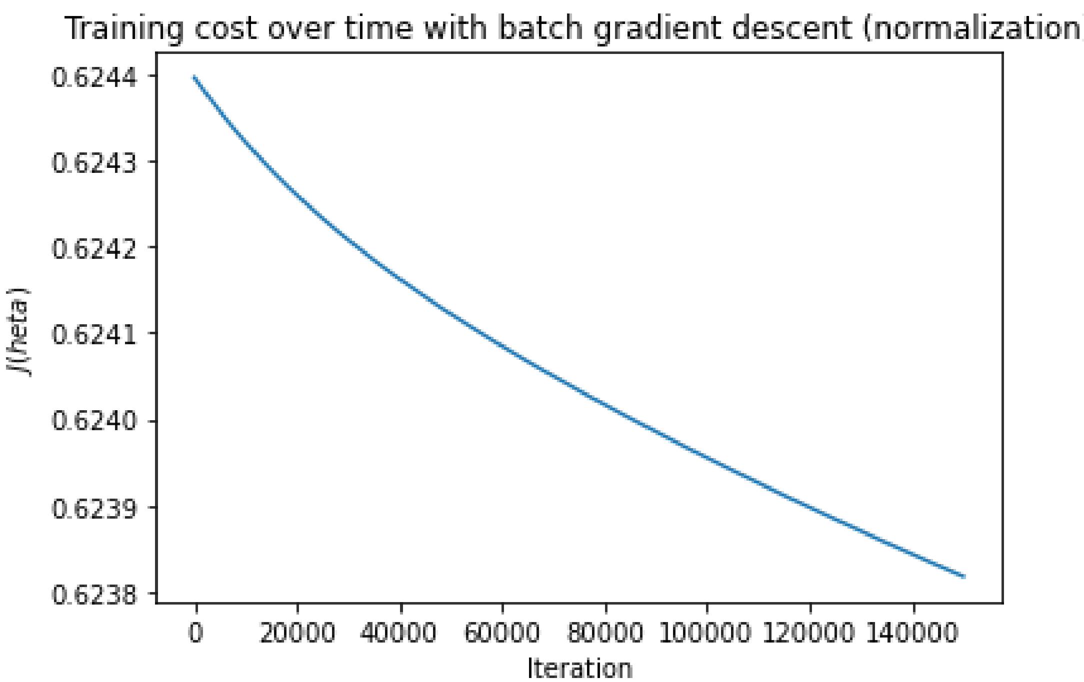
149999



149999



149999



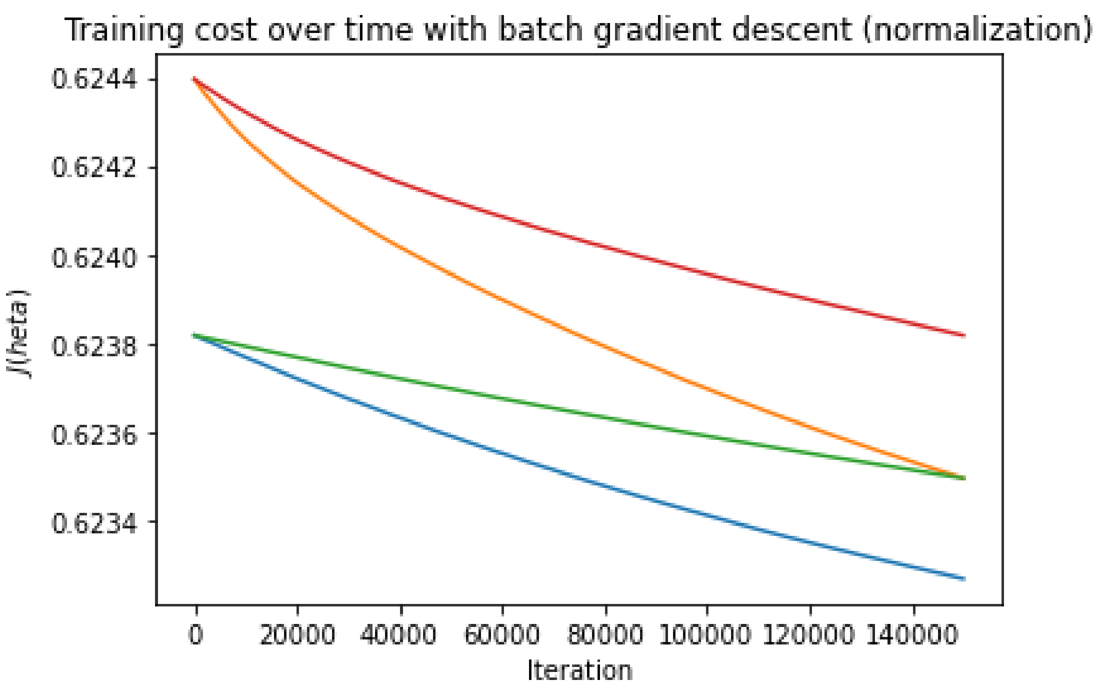
[47]:
```
thrta=np.array(theta_list)
thrta=thrta.reshape(4,11)
thrta.shape
```

[47]: (4, 11)

[48]:
```
len(j_history_list)
```

[48]: 4

[49]:
```
plt.plot(j_history_list[0])
plt.plot(j_history_list[1])
plt.plot(j_history_list[2])
plt.plot(j_history_list[3])

plt.xlabel("Iteration")
plt.ylabel("$J(\theta)$")
plt.title("Training cost over time with batch gradient descent (normalization)")
plt.show()
```



[50]:
```
def r_squared(y, y_pred):
    return 1 - np.square(y - y_pred).sum() / np.square(y - y.mean()).sum()
```

```
y_test_pred_soft = h(X_test, theta_list[2])

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 0.0020, hard R^2 -0.4386, accuracy 0.70
```

[51]:
```
Xtest=data_test[['ApplicantIncome','LoanAmount','Gender','Married','Dependents','Education','S

Xtest = normalize(Xtest)
Xtest=np.insert(Xtest, 0, 1, axis=1)
```

[52]:
```
y_test_pred_soft = h(Xtest, theta_list[3])
y_test_pred_hard = (y_test_pred_soft > 0.5).astype(int)
y_test_pred_hard.shape
result=np.concatenate((Xtest, y_test_pred_hard), axis=1)
```

[53]:
```
result=data_test
result['Result_Loan_Status']=y_test_pred_hard
```

[54]:
```
status = {1: 'Y',0: 'N'}
result.Result_Loan_Status = [status[item] for item in result.Result_Loan_Status]
```

[55]:
```
result
```

[55]:
```
      Loan_ID  Gender  Married Dependents  Education  Self_Employed  \
0    LP001015       1        1          0          1              0
1    LP001022       1        1          1          1              0
2    LP001031       1        1          2          1              0
3    LP001035       1        1          2          1              0
4    LP001051       1        0          0          0              0
..        …       …        …          …          …              …
362  LP002971       1        1          4          0              1
363  LP002975       1        1          0          1              0
364  LP002980       1        0          0          1              0
365  LP002986       1        1          0          1              0
366  LP002989       1        0          0          1              1

     ApplicantIncome  CoapplicantIncome  LoanAmount  Loan_Amount_Term  \
0               5720                  0       110.0             360.0
1               3076               1500       126.0             360.0
```

```
2               5000                1800        208.0              360.0
3               2340                2546        100.0              360.0
4               3276                   0         78.0              360.0
..               …                   …          …                …
362             4009                1777        113.0              360.0
363             4158                 709        115.0              360.0
364             3250                1993        126.0              360.0
365             5000                2393        158.0              360.0
366             9200                   0         98.0              180.0

     Credit_History Property_Area Result_Loan_Status
0               1.0         Urban                  Y
1               1.0         Urban                  Y
2               1.0         Urban                  Y
3               1.0         Urban                  Y
4               1.0         Urban                  Y
..              …           …                 …
362             1.0         Urban                  Y
363             1.0         Urban                  Y
364             1.0     Semiurban                  Y
365             1.0         Rural                  Y
366             1.0         Rural                  Y

[367 rows x 13 columns]
```

[56]:
```
result[['Loan_ID','Result_Loan_Status']]
```

[56]:
```
      Loan_ID Result_Loan_Status
0    LP001015                  Y
1    LP001022                  Y
2    LP001031                  Y
3    LP001035                  Y
4    LP001051                  Y
..        …                …
362  LP002971                  Y
363  LP002975                  Y
364  LP002980                  Y
365  LP002986                  Y
366  LP002989                  Y

[367 rows x 2 columns]
```

For this Dataset, It still have null information in columns Gender, Self_Employed, Loan_Amount_Term and Credit_History. Then I fill them by add same information but do not or less impact to avg of data. Affter that, I set columns drop columns of Loan_ID and Property_Area because Loan_ID it is just identified data from who and Property_Area it is name of Area that is str type that may impact that i don't know how to change to value and do not impact to datasets.

Affer that, I change gender colums from Male and Female to be 1 and 2, Self_Employed colums from Yes and No to be 1 and 0 , Education colums from Graduate and Not Graduate to be 1 and 0, Married colums from Yes and No to be 1 and 0 and Loan_Status colums from Y and N to be 1 and 0.
Next, I set columns Gender, Married, Dependents, Education, Self_Employed, ApplicantIncome, CoapplicantIncome, LoanAmount, Loan_Amount_Term and Credit_History to be X Train and Loan_Status to be Y train. Affter that, I normalization data for help about calculater faster by sklearn.preprocessing.normalize function. Affter that, I train with alpha1 = .0001 alpha2 = .00005. I get best parameters = [[4.00496168e-01],[3.28510362e-01],[4.13908503e-03],[9.49288206e-05],[9.18202284e-05],[-1.26620534e-04],[1.27662685e-04],[9.69181757e-06],[2.43066237e-01],[2.43152113e-02],[4.09699423e-04]]. I got model's got test set soft R^2 0.0020, hard R^2 -0.4386, accuracy 0.70.. Then make predeced Loan_Status from test set and change out put 1,0 to be Y and N.