ziqi1756_Ziqi_Tan_ps2

February 24, 2020

0.1 Problem Set 2: Linear Regression

To run and solve this assignment, one must have a working IPython Notebook installation. The easiest way to set it up for both Windows and Linux is to install Anaconda. Then save this file to your computer (use ”Raw” link on gist/github), run Anaconda and choose this file in Anaconda’s file explorer. Use the `Python 3` version. Everything that follows assumes that you have already followed these instructions. If you are new to Python or its scientific library, Numpy, there are some nice tutorials here and here.

To run code in a cell or to render Markdown+LaTeX press `Ctr+Enter` or `[>|]`(like “play”) button above. To edit any code or text cell [double]click on its content. To change cell type, choose “Markdown” or “Code” in the drop-down menu above.

If certain output is given for some cells, that means that you are expected to get similar results.

Total: 155 points.

0.1.1 1. Numpy Tutorial

1.1 [5pt] Modify the cell below to return a 5x5 matrix of ones. Put some code there and press `Ctrl+Enter` to execute contents of the cell. You should see something like the output below. [1] [2]

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

# raise NotImplementedError("Replace this raise statement with the code "
#                            "that prints 5x5 matrix of ones")
ones_matrix = np.ones((5,5))
print(ones_matrix)
```

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

1.2 [5pt] Vectorizing your code is very important to get results in a reasonable time. Let A be a 10x10 matrix and x be a 10-element column vector. Your friend writes the following code. How would you vectorize this code to run without any for loops? Compare execution speed for different values of `n` with `%timeit`.

[2]:
```
n = 10
def compute_something(A, x):
    v = np.zeros((n, 1))
    for i in range(n):
        for j in range(n):
            v[i] += A[i, j] * x[j]
    return v

A = np.random.rand(n, n)
x = np.random.rand(n, 1)
print(compute_something(A, x))
```

```
[[2.12000841]
 [2.11181126]
 [1.82145072]
 [2.60146155]
 [3.02349051]
 [1.37609514]
 [2.37565226]
 [2.18886539]
 [2.54750721]
 [2.50917717]]
```

[3]:
```
# How would you vectorize this code to run without any for loops?
# Vectorizing your code is very important to get results in a reasonable time.
def vectorized(A, x):
    # raise NotImplementedError('Put your vectorized code here!')
    return np.dot(A,x)

print(vectorized(A, x))
assert np.max(abs(vectorized(A, x) - compute_something(A, x))) < 1e-3
```

```
[[2.12000841]
 [2.11181126]
 [1.82145072]
 [2.60146155]
 [3.02349051]
 [1.37609514]
 [2.37565226]
 [2.18886539]
 [2.54750721]
 [2.50917717]]
```

[4]:
```
# Compare execution speed for different values of n with %timeit.
for n in [5, 10, 100, 500]:
    A = np.random.rand(n, n)
    x = np.random.rand(n, 1)
```

```python
    print('n=', n)
    print("Code segment1: ")
    %timeit -n 5 compute_something(A, x)
    print("vectorize: ")
    %timeit -n 5 vectorized(A, x)
    print('---')
    print()
```

```
n= 5
Code segment1:
69 µs ± 1.56 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
vectorize:
The slowest run took 4.86 times longer than the fastest. This could mean that an
intermediate result is being cached.
1.13 µs ± 967 ns per loop (mean ± std. dev. of 7 runs, 5 loops each)
---

n= 10
Code segment1:
268 µs ± 18.7 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
vectorize:
The slowest run took 4.22 times longer than the fastest. This could mean that an
intermediate result is being cached.
1.12 µs ± 795 ns per loop (mean ± std. dev. of 7 runs, 5 loops each)
---

n= 100
Code segment1:
26.3 ms ± 813 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
vectorize:
The slowest run took 8.95 times longer than the fastest. This could mean that an
intermediate result is being cached.
5.05 µs ± 5.43 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
---

n= 500
Code segment1:
638 ms ± 11.6 ms per loop (mean ± std. dev. of 7 runs, 5 loops each)
vectorize:
The slowest run took 9225.99 times longer than the fastest. This could mean that
an intermediate result is being cached.
12.8 ms ± 31.3 ms per loop (mean ± std. dev. of 7 runs, 5 loops each)
---
```

0.1.2 2. Linear regression with one variable

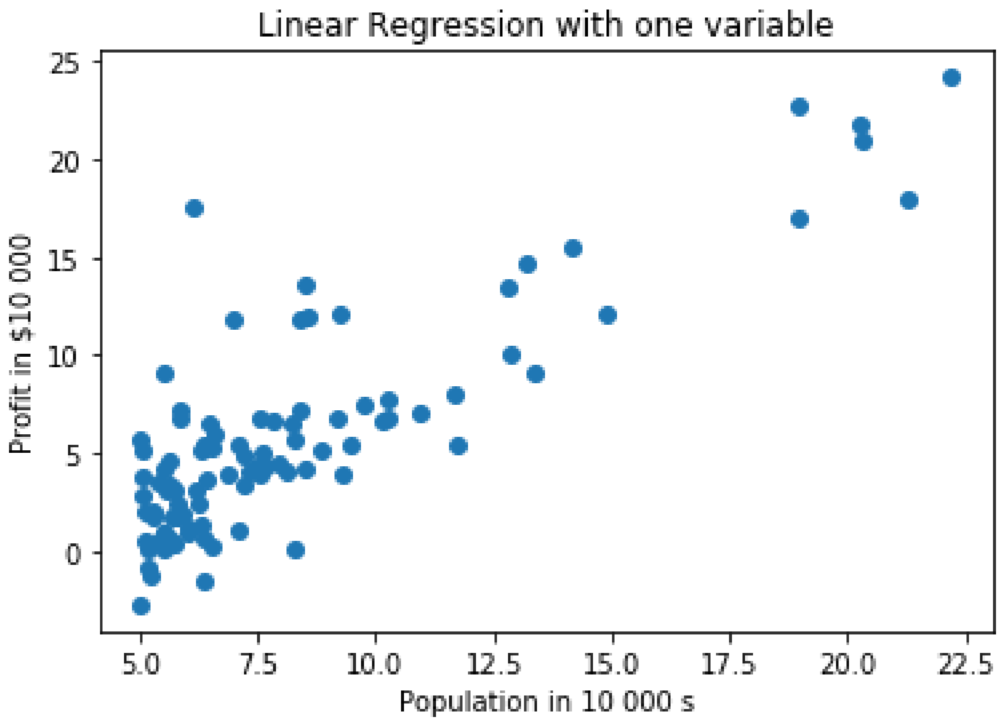
In this part of the exercise, you will implement linear regression with one variable to predict profits for a food truck. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities and you have data for profits and populations from the cities. You would like to use this data to help you select which city to expand to next. The file ex1data.txt contains the dataset for our linear regression problem. The first column is the population of a city and the second column is the profit of a food truck in that city. A negative value for profit indicates a loss.

2.1 [10pt] Generate a plot similar to the one below [1] [2] [3]

Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population). Many other problems that you will encounter in real life are multi-dimensional and can't be plotted on a 2-d plot.

[5]:
```python
data = np.loadtxt('ex1data1.txt', delimiter=',')
X, y = data[:, 0, np.newaxis], data[:, 1, np.newaxis]
n = data.shape[0]
print(X.shape, y.shape, n)
print(X[:10], '\n', y[:10])

# raise NotImplementedError('Put the visualziation code here.')
plt.scatter(X, y)
plt.title('Linear Regression with one variable')
plt.xlabel('Population in 10 000 s')
plt.ylabel('Profit in $10 000')
plt.show()
```

```
(97, 1) (97, 1) 97
[[6.1101]
 [5.5277]
 [8.5186]
 [7.0032]
 [5.8598]
 [8.3829]
 [7.4764]
 [8.5781]
 [6.4862]
 [5.0546]]
 [[17.592 ]
 [ 9.1302]
 [13.662 ]
 [11.854 ]
 [ 6.8233]
 [11.886 ]
 [ 4.3483]
 [12.    ]
 [ 6.5987]
 [ 3.8166]]
```



2.2 Gradient Descent

In this part, you will fit the linear regression parameter θ to our dataset using gradient descent. The objective of linear regression is to minimize the cost function

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h(x^{(i)};\theta) - y^{(i)}\right)^2$$

where the hypothesis $h(x;\theta)$ is given by the linear model (x' has an additional fake feature always equal to '1')

$$h(x;\theta) = \theta^T x' = \theta_0 + \theta_1 x$$

Recall that the parameters of your model are the θ_j values. These are the values you will adjust to minimize the cost J(θ). One way to do this is to use the gradient descent. In batch gradient descent algorithm, value of θ is updated iteratively using the gradient of J(θ).

$$\theta_j^{(k+1)} = \theta_j^{(k)} - \eta\frac{1}{m}\sum_i\left(h(x^{(i)};\theta) - y^{(i)}\right)x_j^{(i)}$$

With each step of gradient descent, your parameter θ_j comes closer to the optimal values that will achieve the lowest cost J(θ).

2.2.1 [5pt] Where does this update rule come from?

Answer: It comes from the derivative of the cost function.

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h(x^{(i)};\theta) - y^{(i)}\right)^2$$

$$\begin{aligned}\frac{\partial}{\partial\theta_j}J(\theta) &= \frac{1}{2m}\sum_{i=1}^{m}\frac{\partial}{\partial\theta_j}(h_\theta(x^{(i)})-y^{(i)})^2\\ &= \frac{1}{2m}\sum_{i=1}^{m}\cdot 2\cdot(h_\theta(x^{(i)})-y^{(i)})\cdot\frac{\partial}{\partial\theta_j}(h_\theta(x^{(i)})-y^{(i)})\\ &= \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})\frac{\partial}{\partial\theta_j}(\theta_0+\theta_1x_1^{(i)}+\theta_2x_2^{(i)}+\cdots+\theta_nx_n^{(i)}-y^{(i)})\\ &= \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})x_j\end{aligned}$$

Therefore,

$$\theta_j^{(k+1)} = \theta_j^{(k)} - \eta\frac{\partial J(\theta)}{\partial\theta_j}$$

where η is the learning rate.

2.2.2 [30pt] Cost Implementation

As you perform gradient descent to learn to minimize the cost function, it is helpful to monitor the convergence by computing the cost. In this section, you will implement a function to calculate $J(\theta)$ so you can check the convergence of your gradient descent implementation.

In the following lines, we add another dimension to our data to accommodate the intercept term and compute the prediction and the loss. As you are doing this, remember that the variables X and y are not scalar values, but matrices whose rows represent the examples from the training set. In order to get x' add a column of ones to the data matrix `X`.

You should expect to see a cost of approximately 32.

[6]:
```python
# assertions below are true only for this
# specific case and are given to ease debugging!


def add_column(X):
    '''
        In order to get x' add a column of ones to the data matrix X.
        x'  has an additional fake feature always equal to '1'.
    '''
    assert len(X.shape) == 2 and X.shape[1] == 1
    '''
        numpy.insert(arr, obj, values, axis=None)
            parameters:
                arr: Input array.
                obj: Object that defines the index or indices before which␣
 ↪values is inserted.
                values: Values to insert into arr.
                axis: Axis along which to insert values.
    '''
    # raise NotImplementedError("Insert a column of ones to the _left_ side of␣
 ↪the matrix")
    return np.insert(X, 0, 1, 1)
```

```python
def predict(X, theta):
    """
        Computes h(x; theta)
    """
    assert len(X.shape) == 2 and X.shape[1] == 1
    assert theta.shape == (2, 1)

    X_prime = add_column(X)
    # raise NotImplementedError("Compute the regression predictions")
    # X(97,2) theta(2,1)
    pred = np.dot(X_prime, theta)
    return pred

def loss(X, y, theta):
    assert X.shape == (n, 1)
    assert y.shape == (n, 1)
    assert theta.shape == (2, 1)

    X_prime = add_column(X)
    assert X_prime.shape == (n, 2)

    # raise NotImplementedError("Compute the model loss; use the predict()␣
 ↪function")
    pred = predict(X, theta)
    groundT = y
    loss = np.sum((pred-groundT)**2)/(2*y.shape[0])
    return loss


theta_init = np.zeros((2, 1))
print(loss(X, y, theta_init))
```

```
32.072733877455676
```

2.2.3 [40pt] GD Implementation

Next, you will implement gradient descent. The loop structure has been written for you, and you only need to supply the updates to θ within each iteration.

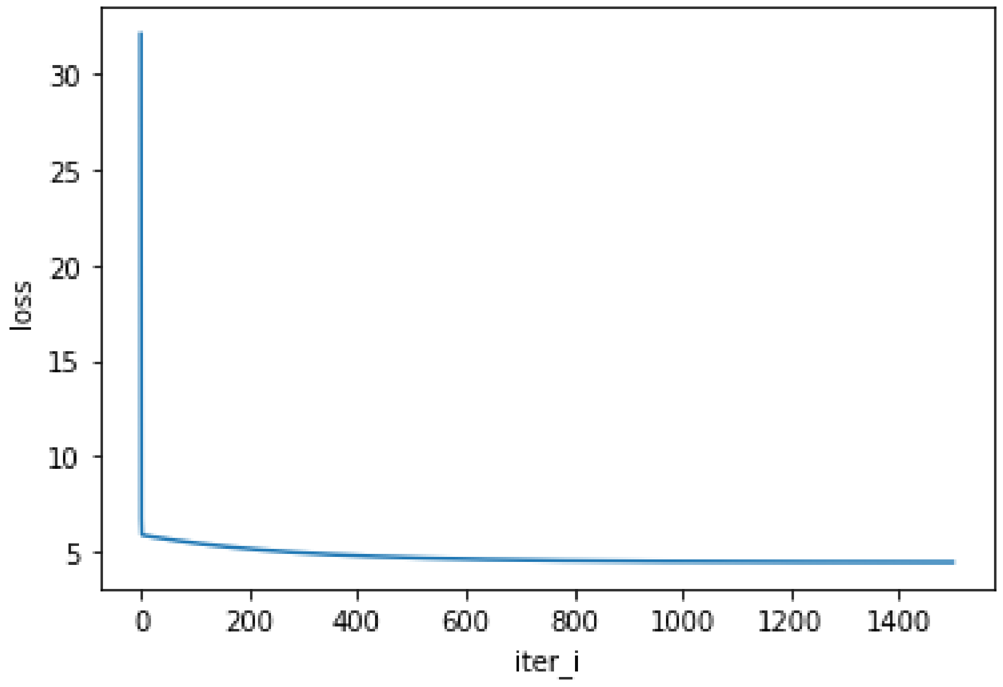
As you write your code, make sure you understand what you are trying to optimize and what is being updated. Keep in mind that the cost is parameterized by the vector θ not X and y. That is, we minimize the value of $J(\theta)$ by changing the values of the vector θ, not by changing X or y.

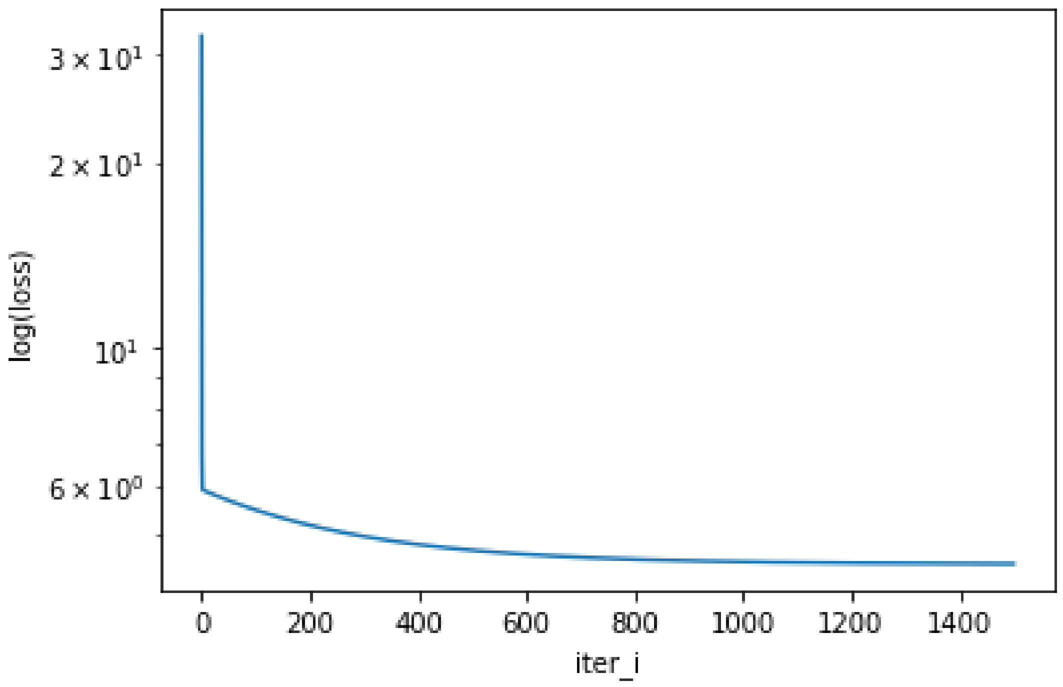
A good way to verify that gradient descent is working correctly is to look at the value of $J(\theta)$ and check that it is decreasing with each step. Your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm. Another way of making sure your gradient estimate is correct is to check it againts a finite difference approximation.

We also initialize the initial parameters to 0 and the learning rate alpha to `0.01`.

[7]:
```
import scipy.optimize
from functools import partial

def loss_gradient(X, y, theta):
    X_prime = add_column(X)
#     raise NotImplementedError("Compute the model loss gradient; "
#                               "use the predict() function; "
#                               "this also must be vectorized!")

    pred = predict(X, theta)
    loss_grad = np.dot((pred -y).T,X_prime).T / X.shape[0]
    return loss_grad

assert loss_gradient(X, y, theta_init).shape == (2, 1)

def finite_diff_grad_check(f, grad, points, eps=1e-10):
    errs = []
    for point in points:
        point_errs = []
        grad_func_val = grad(point)
        for dim_i in range(point.shape[0]):
            diff_v = np.zeros_like(point)
            diff_v[dim_i] = eps
            dim_grad = (f(point+diff_v) - f(point-diff_v))/(2*eps)
            point_errs.append(abs(dim_grad - grad_func_val[dim_i]))
        errs.append(point_errs)
    return errs

test_points = [np.random.rand(2, 1) for _ in range(10)]
finite_diff_errs = finite_diff_grad_check(
    partial(loss, X, y), partial(loss_gradient, X, y), test_points
)

print('max grad comp error', np.max(finite_diff_errs))
assert np.max(finite_diff_errs) < 1e-3, "grad computation error is too large"

def run_gd(loss, loss_gradient, X, y, theta_init, lr=0.01, n_iter=1500):
    theta_current = theta_init.copy()
    loss_values = []
    theta_values = []

    for i in range(n_iter):
        loss_value = loss(X, y, theta_current)
        # raise NotImplementedError("Put update step code here")
        theta_current = theta_current - lr * loss_gradient(X, y, theta_current)
        loss_values.append(loss_value)
        theta_values.append(theta_current)
    return theta_current, loss_values, theta_values

result = run_gd(loss, loss_gradient, X, y, theta_init)
theta_est, loss_values, theta_values = result

print('estimated theta value', theta_est.ravel())  # Return a contiguous␣
 ↪flattened array.
print('resulting loss', loss(X, y, theta_est))
plt.ylabel('loss')
plt.xlabel('iter_i')
plt.plot(loss_values)
plt.show()

plt.ylabel('log(loss)')
plt.xlabel('iter_i')
plt.semilogy(loss_values)
plt.show()
```

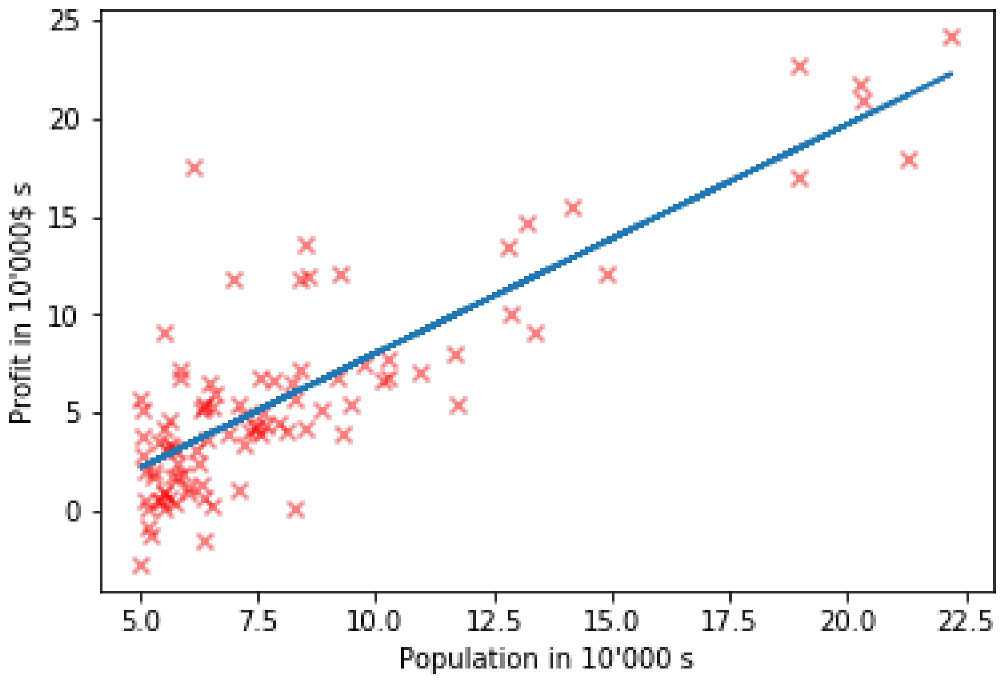
```
max grad comp error 4.1643053059203794e-05
estimated theta value [-3.63029144  1.16636235]
resulting loss 4.483388256587726
```





2.2.4 [10pt] After you are finished, use your final parameters to plot the linear fit. The result should look something like on the figure below. Use the `predict()` function.

[8]:
```
plt.scatter(X, y, marker='x', color='r', alpha=0.5)
x_start, x_end = 5, 25

# raise NotImplementedError("Put code that plots a regression line here")
plt.plot(X, predict(X, theta_est.ravel().reshape(2,1)))

plt.xlabel('Population in 10\'000 s')
plt.ylabel('Profit in 10\'000$ s')
plt.show()
```



Now use your final values for θ and the `predict()` function to make predictions on profits in areas of 35,000 and 70,000 people.

[9]:
```
# raise NotImplementedError("Predict values given inputs")
test_X = np.array([35000, 70000]).reshape(2, 1)
print(predict(test_X, theta_est))
```

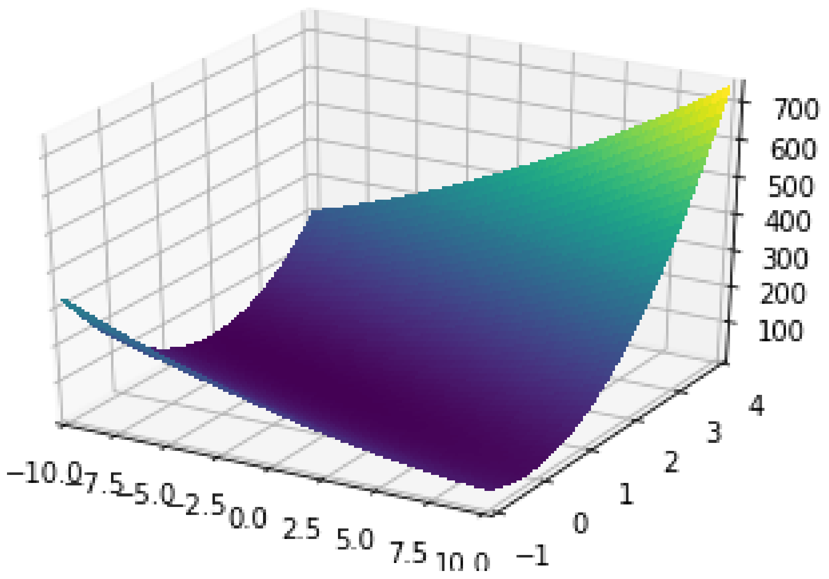
```
[[40819.05197031]
 [81641.73423205]]
```

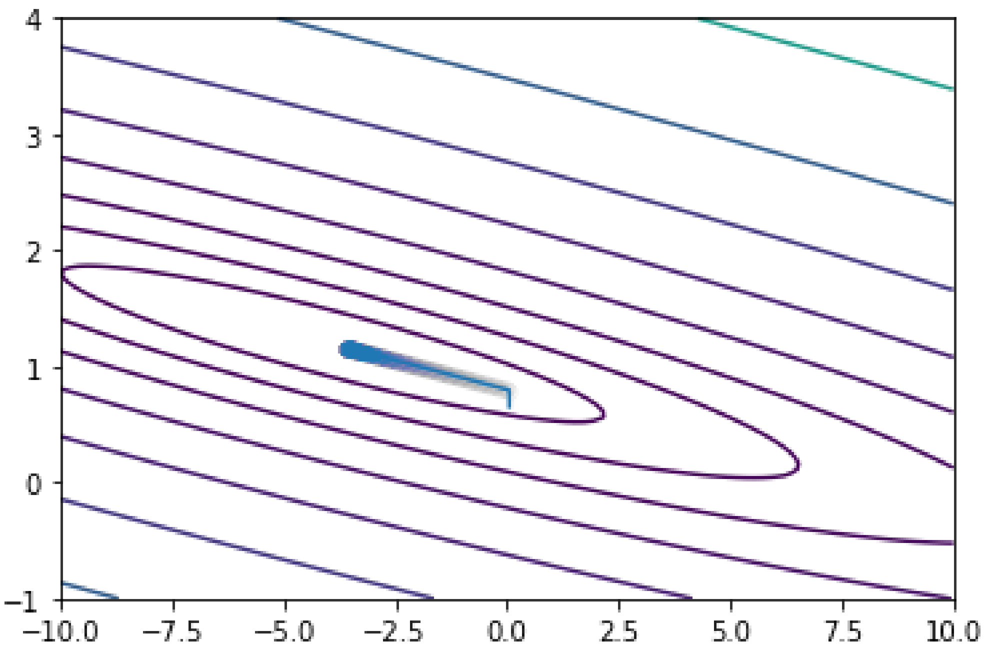
To understand the cost function better, you will now plot the cost over a 2-dimensional grid of values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images.

[10]:
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm
limits = [(-10, 10), (-1, 4)]
space = [np.linspace(*limit, 100) for limit in limits]
theta_1_grid, theta_2_grid = np.meshgrid(*space)
theta_meshgrid = np.vstack([theta_1_grid.ravel(), theta_2_grid.ravel()])
loss_test_vals_flat = (((add_column(X) @ theta_meshgrid - y)**2).mean(axis=0)/2)
loss_test_vals_grid = loss_test_vals_flat.reshape(theta_1_grid.shape)
print(theta_1_grid.shape, theta_2_grid.shape, loss_test_vals_grid.shape)

plt.gca(projection='3d').plot_surface(theta_1_grid, theta_2_grid,
                                      loss_test_vals_grid, cmap=cm.viridis,
                                      linewidth=0, antialiased=False)
```

```
xs, ys = np.hstack(theta_values).tolist()
zs = np.array(loss_values)
plt.gca(projection='3d').plot(xs, ys, zs, c='r')
plt.xlim(*limits[0])
plt.ylim(*limits[1])
plt.show()

plt.contour(theta_1_grid, theta_2_grid, loss_test_vals_grid, levels=np.
 ↪logspace(-2, 3, 20))
plt.plot(xs, ys)
plt.scatter(xs, ys, alpha=0.005)
plt.xlim(*limits[0])
plt.ylim(*limits[1])
plt.show()
```

```
(100, 100) (100, 100) (100, 100)
```





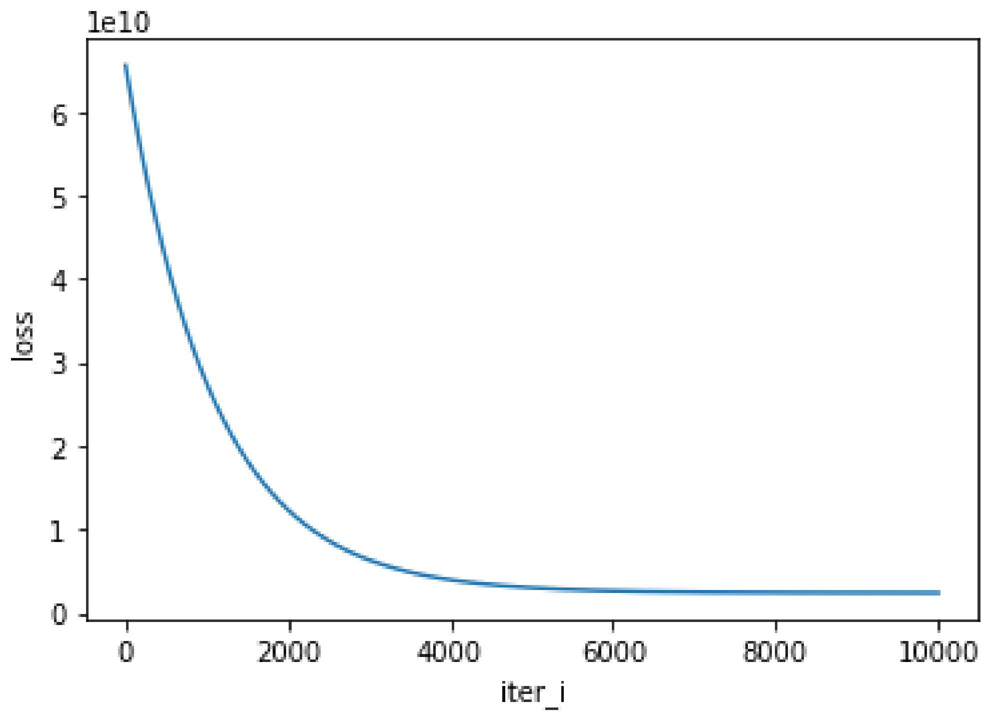
0.1.3 3. Linear regression with multiple input features

3.1 [20pt] Copy-paste your `add_column`, `predict`, `loss` and `loss grad` implementations from above and modify your code of linear regression with one variable to support any number of input features (vectorize your code.)

[11]:
```
data = np.loadtxt('ex1data2.txt', delimiter=',')
X, y = data[:, :-1], data[:, -1, np.newaxis]
n = data.shape[0]
print(X.shape, y.shape, n)
print(X[:10], '\n', y[:10])
```

```
(47, 2) (47, 1) 47
[[2.104e+03 3.000e+00]
 [1.600e+03 3.000e+00]
 [2.400e+03 3.000e+00]
 [1.416e+03 2.000e+00]
 [3.000e+03 4.000e+00]
 [1.985e+03 4.000e+00]
 [1.534e+03 3.000e+00]
 [1.427e+03 3.000e+00]
 [1.380e+03 3.000e+00]
 [1.494e+03 3.000e+00]]
 [[399900.]
 [329900.]
 [369000.]
 [232000.]
 [539900.]
 [299900.]
 [314900.]
 [198999.]
 [212000.]
 [242500.]]
```

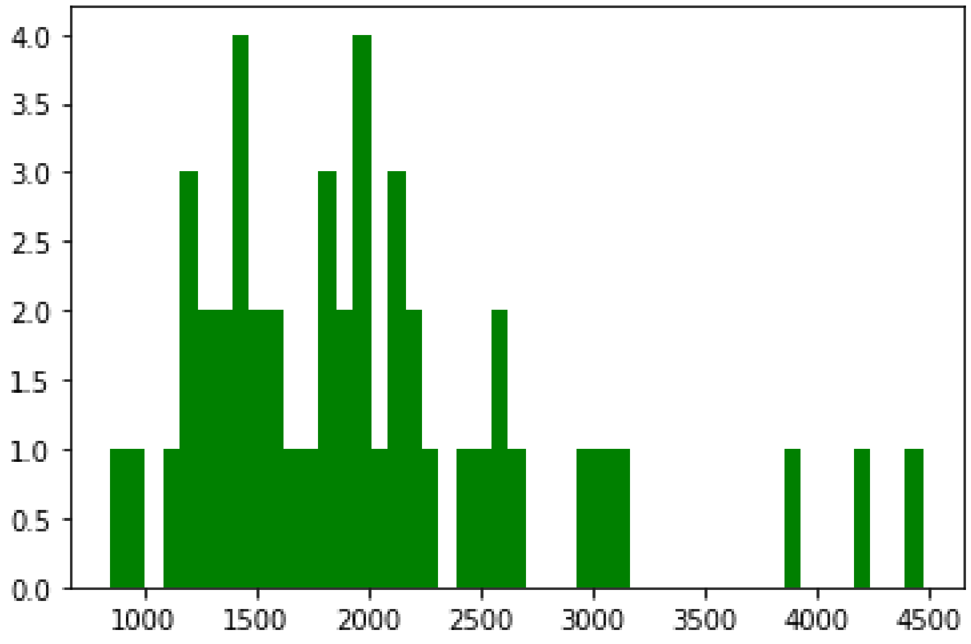
[12]:
```python
# raise NotImplementedError("Implement new add_column(), predict(), loss(),␣
 ↪loss_gradient() here for multivariate regression")

def add_column(X):

    assert len(X.shape) == 2
    '''
        numpy.insert(arr, obj, values, axis=None)
            parameters:
                arr: Input array.
                obj: Object that defines the index or indices before which␣
 ↪values is inserted.
                values: Values to insert into arr.
                axis: Axis along which to insert values.
    '''
    # raise NotImplementedError("Insert a column of ones to the _left_ side of␣
 ↪the matrix")
    return np.insert(X, 0, 1, 1)


def predict(X, theta):
    """
        Computes h(x; theta)
    """
    assert len(X.shape) == 2

    X_prime = add_column(X)
    # raise NotImplementedError("Compute the regression predictions")
    # X(97,2) theta(2,1)
    pred = np.dot(X_prime, theta)
    return pred

def loss(X, y, theta):

    X_prime = add_column(X)

    # raise NotImplementedError("Compute the model loss; use the predict()␣
 ↪function")
    pred = predict(X, theta)
    groundT = y
    loss = np.sum((pred-groundT)**2)/(2*y.shape[0])
    return loss

def loss_gradient(X, y, theta):
    X_prime = add_column(X)
#     raise NotImplementedError("Compute the model loss gradient; "
#                               "use the predict() function; "
#                               "this also must be vectorized!")

    pred = predict(X, theta)
    loss_grad = np.dot((pred -y).T,X_prime).T / X.shape[0]
    return loss_grad

# main
theta_init = np.zeros((3, 1))
result = run_gd(loss, loss_gradient, X, y, theta_init, n_iter=10000, lr=1e-10)
theta_est, loss_values, theta_values = result
plt.plot(loss_values)
plt.xlabel('iter_i')
plt.ylabel('loss')

plt.show()

print('theta es')
```

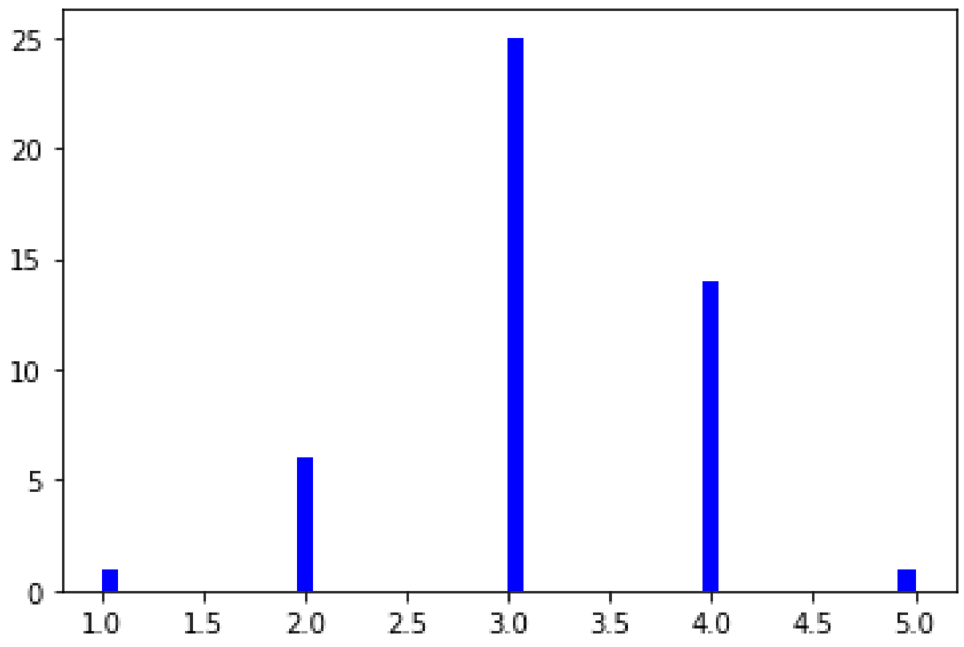


```
theta es
```

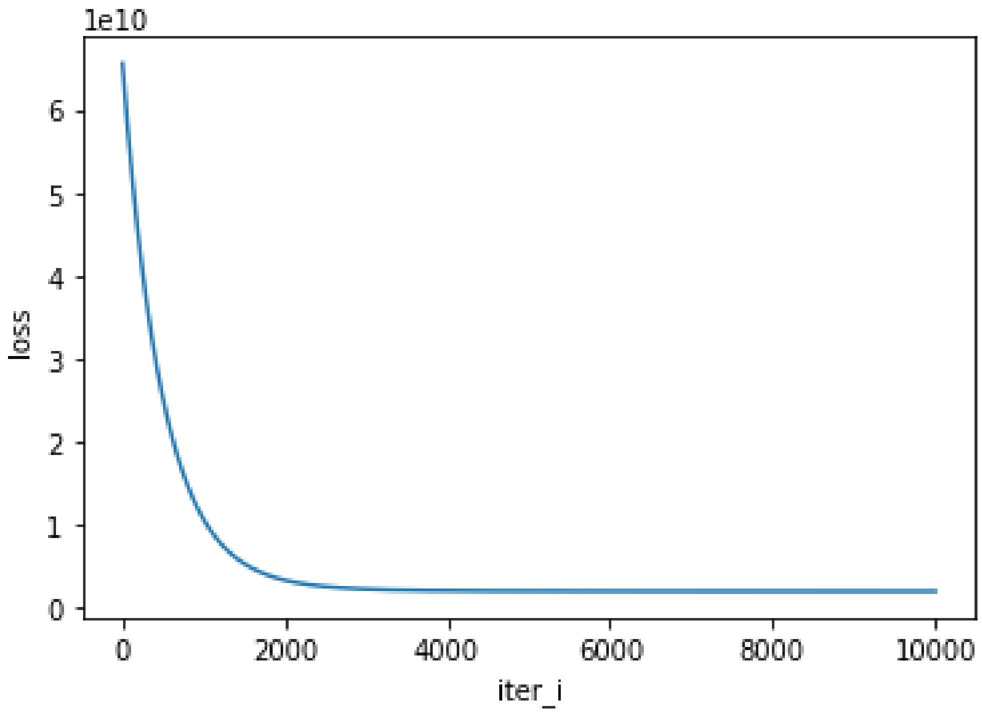
3.2 [20pt] Draw a histogam of values for the first and second feature. Why is feature normalization important? Normalize features and re-run the gradient decent. Compare loss plots that you get with and without feature normalization.

[13]:
```python
# raise NotImplementedError("Draw histogram for values of feature 1")
plt.hist(X[:,0], bins=47, color='g')
plt.show()

# raise NotImplementedError("Draw histogram for values of feature 2")
plt.hist(X[:,1], bins=50, color='b')
plt.show()
```





[17]:
```
theta_init = np.zeros((3, 1))

# numpy.mean: Compute the arithmetic mean along the specified axis.
# numpy.var: Compute the variance along the specified axis.

# Feature normalization

# Zero-mean normalization
X_normed = (X - np.mean(X, axis=0))/np.std(X, axis=0)

# Unit-norm normalization
# X_normed = X / np.max(X, axis=0)
# X_normed = X / np.mean(X, axis=0)

# raise NotImplementedError("Run gd on normalized versions of feature vectors")
result = run_gd(loss, loss_gradient, X_normed, y, theta_init, n_iter=10000,␣
 ↪lr=1e-3)
theta_est, loss_values, theta_values = result

plt.plot(loss_values)
plt.xlabel('iter_i')
plt.ylabel('loss')
plt.show()

print('estimated theta value', theta_est.ravel())  # Return a contiguous␣
 ↪flattened array.
```

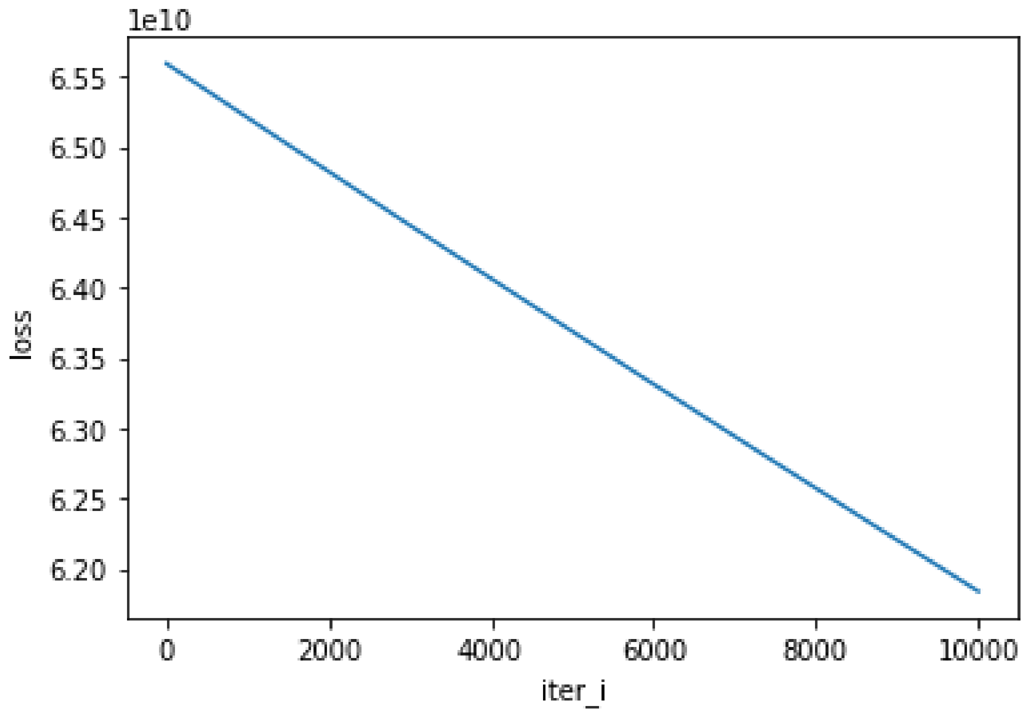


```
estimated theta value [340397.28199562 108736.46443478  -5867.03988583]
```

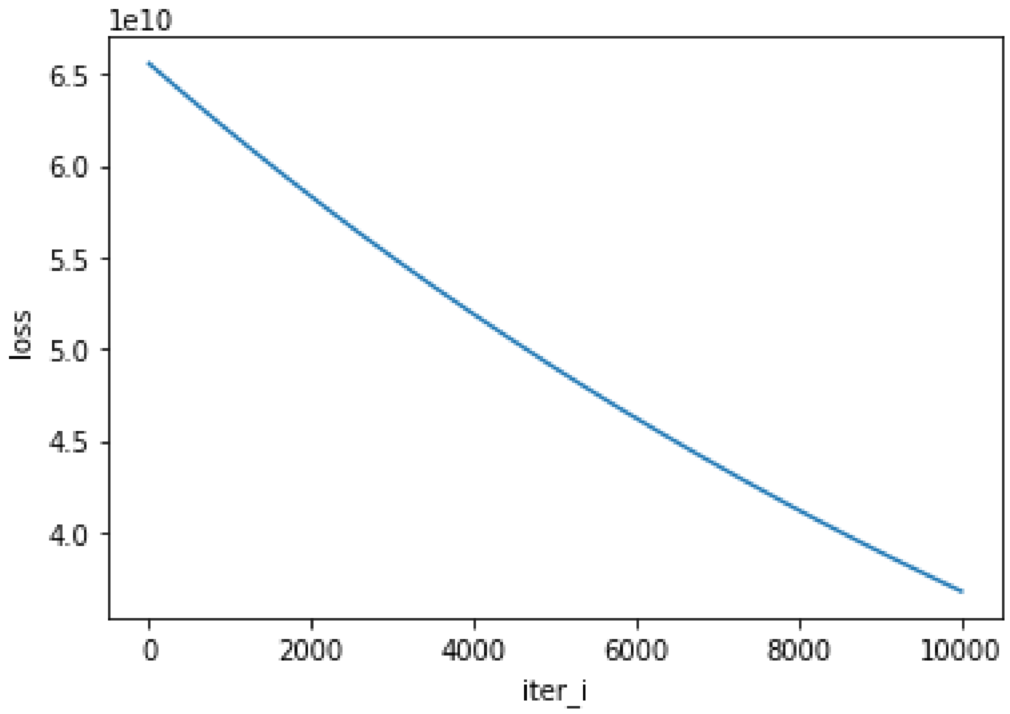
3.3 [10pt] How can we choose an appropriate learning rate? See what will happen if the learning rate is too small or too large for normalized and not normalized cases?

[15]:
```
# raise NotImplementedError("Plot loss behaviour with multiple different␣
↪learning rates")

for lr in [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]:
    result = run_gd(loss, loss_gradient, X_normed, y, theta_init, n_iter=10000,␣
↪lr=lr)
    theta_est, loss_values, theta_values = result

    print('lr=', lr)
    plt.plot(loss_values)
    plt.xlabel('iter_i')
    plt.ylabel('loss')
    plt.show()
```

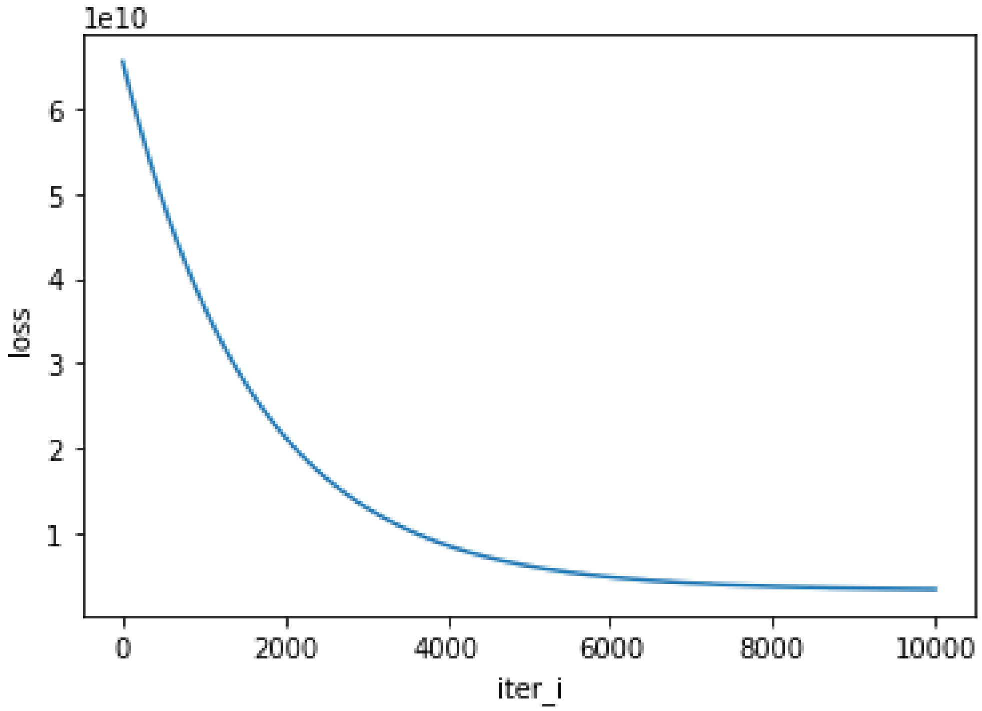
```
lr= 1e-06
```


1e10
loss
iter_i
6.55
6.50
6.45
6.40
6.35
6.30
6.25
6.20
0
2000
4000
6000
8000
10000

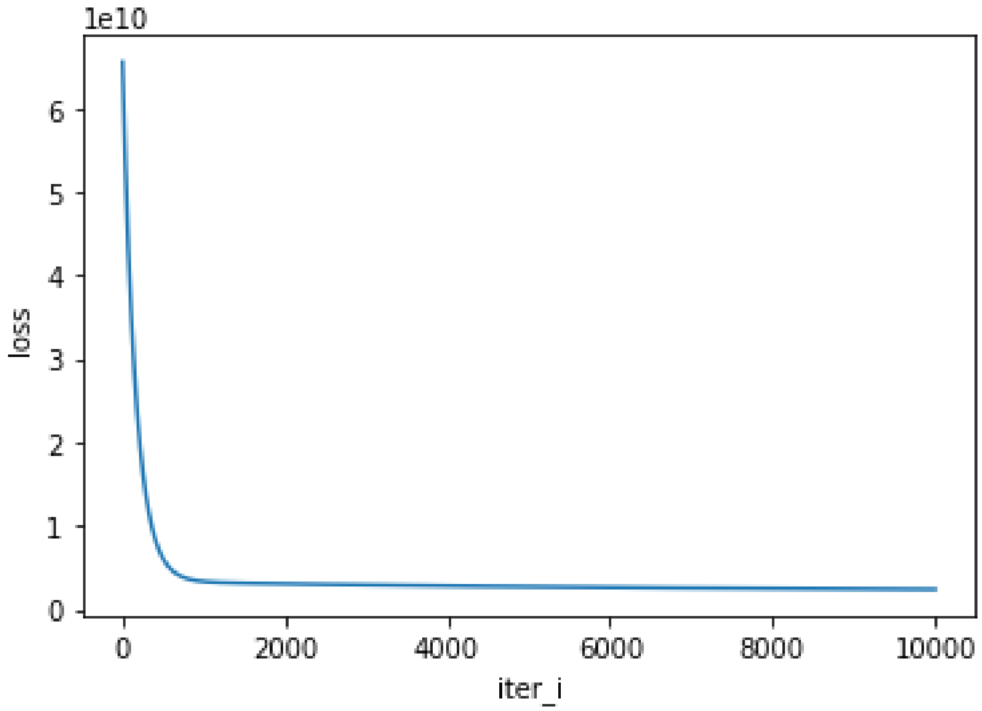
```
lr= 1e-05
```


1e10
loss
iter_i
6.5
6.0
5.5
5.0
4.5
4.0
0
2000
4000
6000
8000
10000

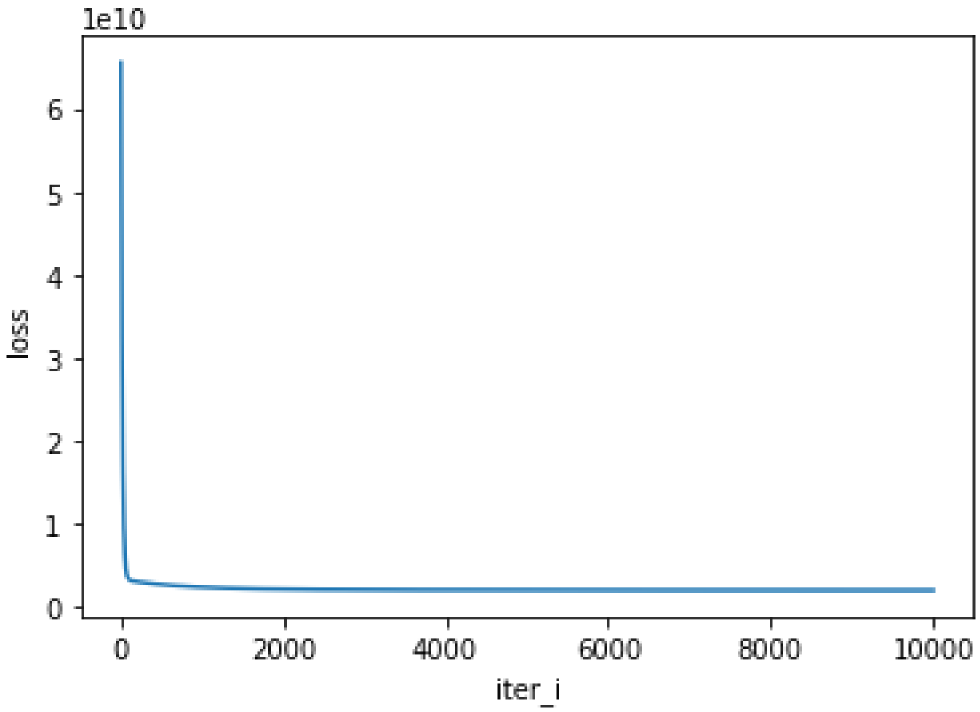
```
lr= 0.0001
```



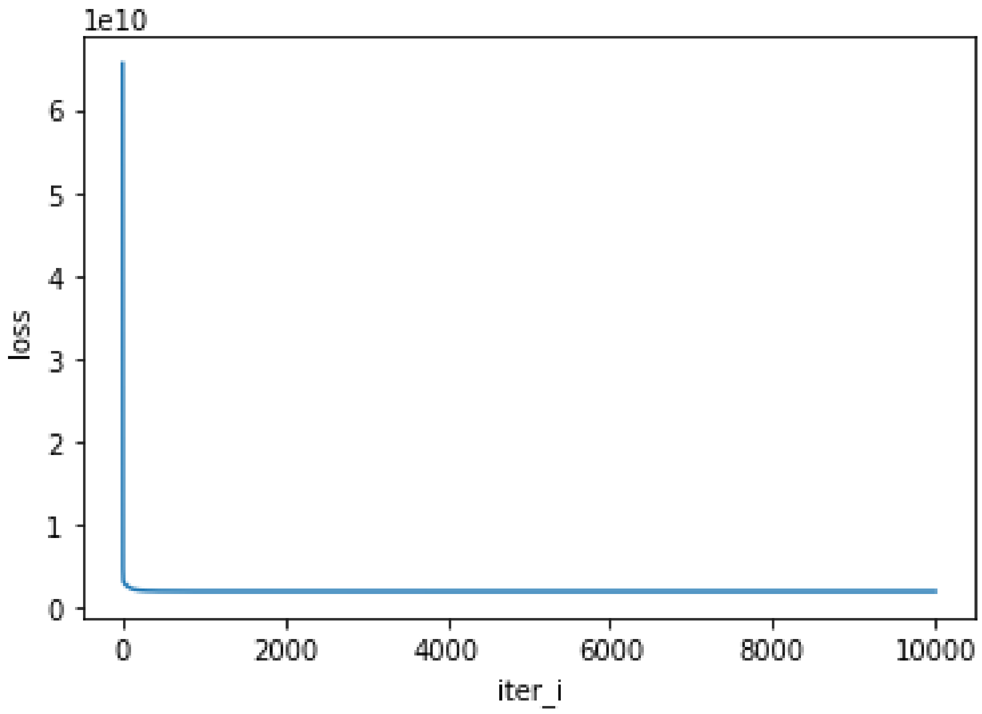
```
lr= 0.001
```



```
lr= 0.01
```



```
lr= 0.1
```


1e10
6
5
4
3
2
1
0
loss
0
2000
4000
6000
8000
10000
iter_i

[ ]: