Problem Set 1: Linear Regression

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 "Text" or "Code" in the drop-down menu above. Here are some useful resources for Markdown guide and LaTeX tutorial if you are not familiar with the basic syntax.

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

We need to submit only a notebook for ps1 submission.

Total: 185 points.

```
In [1]: !pip3 install matplotlib==3.4
```

```
Collecting matplotlib==3.4
  Using cached matplotlib-3.4.0.tar.gz (37.1 MB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: cycler>=0.10 in /opt/homebrew/lib/python3.10/
site-packages (from matplotlib==3.4) (0.12.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /opt/homebrew/lib/python
3.10/site-packages (from matplotlib==3.4) (1.4.7)
Requirement already satisfied: numpy>=1.16 in /opt/homebrew/lib/python3.10/s
ite-packages (from matplotlib==3.4) (2.1.3)
Requirement already satisfied: pillow>=6.2.0 in /opt/homebrew/lib/python3.1
0/site-packages (from matplotlib==3.4) (11.0.0)
Requirement already satisfied: pyparsing>=2.2.1 in /opt/homebrew/lib/python
3.10/site-packages (from matplotlib==3.4) (3.2.0)
Requirement already satisfied: python-dateutil>=2.7 in /Users/jaylee/Librar
y/Python/3.10/lib/python/site-packages (from matplotlib==3.4) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in /Users/jaylee/Library/Python/3.1
0/lib/python/site-packages (from python-dateutil>=2.7->matplotlib==3.4) (1.1
6.0)
Building wheels for collected packages: matplotlib
  Building wheel for matplotlib (setup.py) ... error
  error: subprocess-exited-with-error

  × python setup.py bdist_wheel did not run successfully.
  │ exit code: 1
  ╰─> [934 lines of output]
      /private/var/folders/xt/y_17jz6n3p77v5p6cbm1tstr0000gn/T/pip-install-6
05ooy83/matplotlib_fbf3b75b66bd4693bfaaaeb5d23053c0/setup.py:34: SetuptoolsD
eprecationWarning: The test command is disabled and references to it are dep
recated.
      !!

              ***************************************************************
******************
              Please remove any references to `setuptools.command.test` in a
ll supported versions of the affected package.

              This deprecation is overdue, please update your project and re
move deprecated
              calls to avoid build errors in the future.
              ***************************************************************
******************

      !!
        from setuptools.command.test import test as TestCommand

      Edit setup.cfg to change the build options; suppress output with --qui
et.

      BUILDING MATPLOTLIB
        matplotlib: yes [3.4.0]
            python: yes [3.10.16 (main, Dec  3 2024, 17:27:57) [Clang 16.0.0
                        (clang-1600.0.26.4)]]
          platform: yes [darwin]
             tests: no  [skipping due to configuration]
            macosx: yes [installing]
        File "/private/var/folders/xt/y_17jz6n3p77v5p6cbm1tstr0000gn/T/pip-i
nstall-605ooy83/matplotlib_fbf3b75b66bd4693bfaaaeb5d23053c0/setup.py", line
199, in build_extensions
          package.do_custom_build(env)
        File "/private/var/folders/xt/y_17jz6n3p77v5p6cbm1tstr0000gn/T/pip-i
nstall-605ooy83/matplotlib_fbf3b75b66bd4693bfaaaeb5d23053c0/setupext.py", li
ne 615, in do_custom_build
          subprocess.check_call([make], env=env, cwd=src_path)
        File "/opt/homebrew/Cellar/python@3.10/3.10.16/Frameworks/Python.fra
mework/Versions/3.10/lib/python3.10/subprocess.py", line 369, in check_call
          raise CalledProcessError(retcode, cmd)
      subprocess.CalledProcessError: Command '['make']' returned non-zero ex
it status 2.
      [end of output]

  note: This error originates from a subprocess, and is likely not a problem
with pip.
  ERROR: Failed building wheel for matplotlib
  Running setup.py clean for matplotlib
Failed to build matplotlib
ERROR: Failed to build installable wheels for some pyproject.toml based proj
ects (matplotlib)
```

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 above. [1] [2]

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt

print(np.ones((5, 5)))
```

```
[[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`.

In [3]:
```
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))
```

```
[[2.31606901]
 [2.75453328]
 [2.15330633]
 [2.38128417]
 [2.9637688 ]
 [3.42342179]
 [2.25428761]
 [2.40572295]
 [3.04106936]
 [2.33652812]]
```

In [4]:
```
def vectorized(A, x):
    return np.dot(A, x)

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

```
[[2.31606901]
 [2.75453328]
 [2.15330633]
 [2.38128417]
 [2.9637688 ]
 [3.42342179]
 [2.25428761]
 [2.40572295]
 [3.04106936]
 [2.33652812]]
```

In [5]:
```
for n in [5, 10, 100, 500]:
    A = np.random.rand(n, n)
    x = np.random.rand(n, 1)
    %timeit -n 5 compute_something(A, x)
    %timeit -n 5 vectorized(A, x)
    print('---')
```

```
65 µs ± 6.87 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
The slowest run took 10.62 times longer than the fastest. This could mean th
at an intermediate result is being cached.
16.1 µs ± 16.3 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
---
211 µs ± 15.2 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
4.27 µs ± 1.2 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
---
9.9 ms ± 1.09 ms per loop (mean ± std. dev. of 7 runs, 5 loops each)
The slowest run took 24.72 times longer than the fastest. This could mean th
at an intermediate result is being cached.
4.11 µs ± 7.66 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
---
232 ms ± 8.99 ms per loop (mean ± std. dev. of 7 runs, 5 loops each)
The slowest run took 33.96 times longer than the fastest. This could mean th
at an intermediate result is being cached.
57 µs ± 115 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
---
```

2. Linear regression with one variable

In this part of this 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.

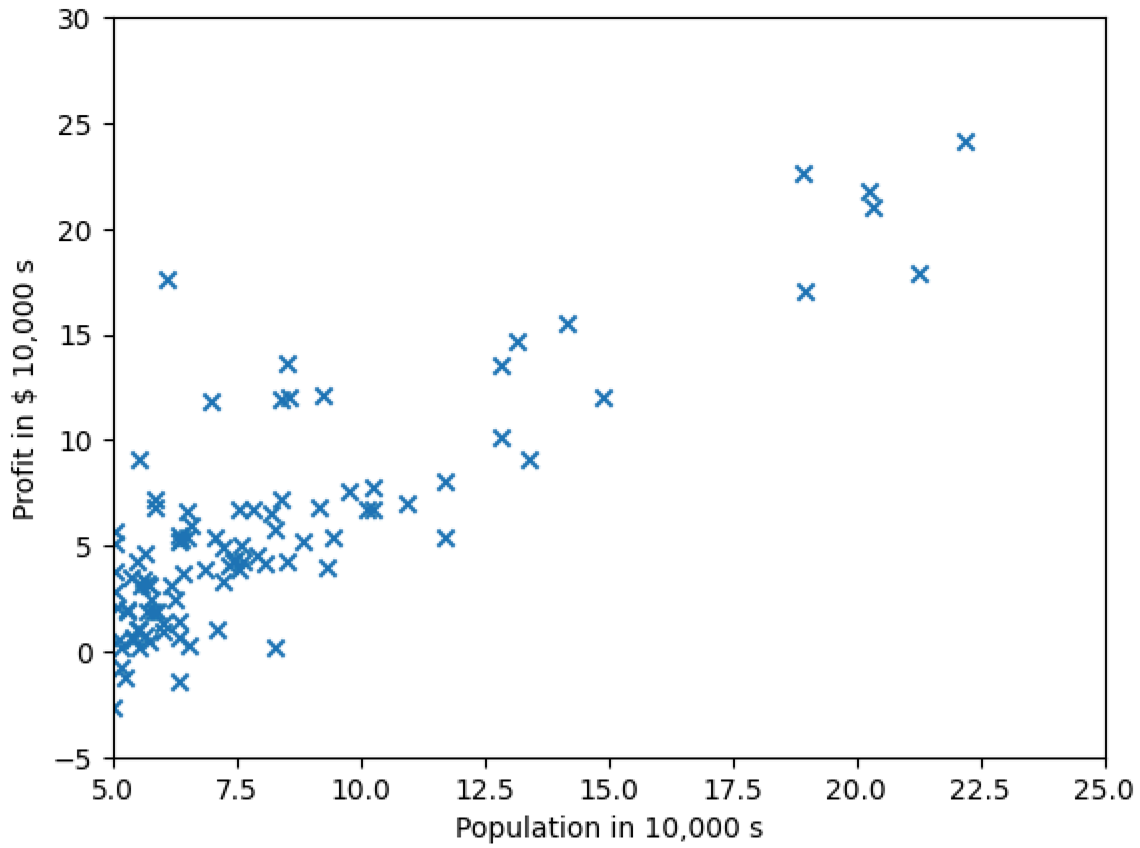
2.1 [10pt] Get a plot similar to below

[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.

In [6]:
```python
data = np.loadtxt('ex1data1.txt', delimiter=',')
X, y = data[:, 0, np.newaxis], data[:, 1, np.newaxis]
n = data.shape[0] # Num of Row
print(X.shape, y.shape, n)
print(X[:10], '\n', y[:10])

import matplotlib.pyplot as plt
plt.scatter(X, y, marker="x")
plt.xlim(5, 25)
plt.xlabel("Population in 10,000 s")
plt.ylim(-5, 30)
plt.ylabel("Profit in $ 10,000 s")

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 cost J(θ). One way to do this is to use the gradient descent algorithm. In batch gradient descent algorithm, each iteration performs the update.

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

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

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

The update rule of gradient descent is that get the gradent of of the θ_j and multiply it with the learning rate η then subtract from the each steps's θ_j.

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.

In [7]:
```
# assertions below are true only for this
# specific case and are given to ease debugging!

def add_column(X):
    assert len(X.shape) == 2 and X.shape[1] == 1
    X = np.insert(X, 0, 1, axis=1)
    return X

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)
    pred = X_prime @ theta

    return pred

def loss(X, y, theta): # this is SSD, basically
    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)

    hypothesis = predict(X, theta)
    loss = (np.dot((hypothesis - y).T, (hypothesis - y)) / (2 * X_prime.shap
    return loss

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

```
[32.07273388]
```

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 program, 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.

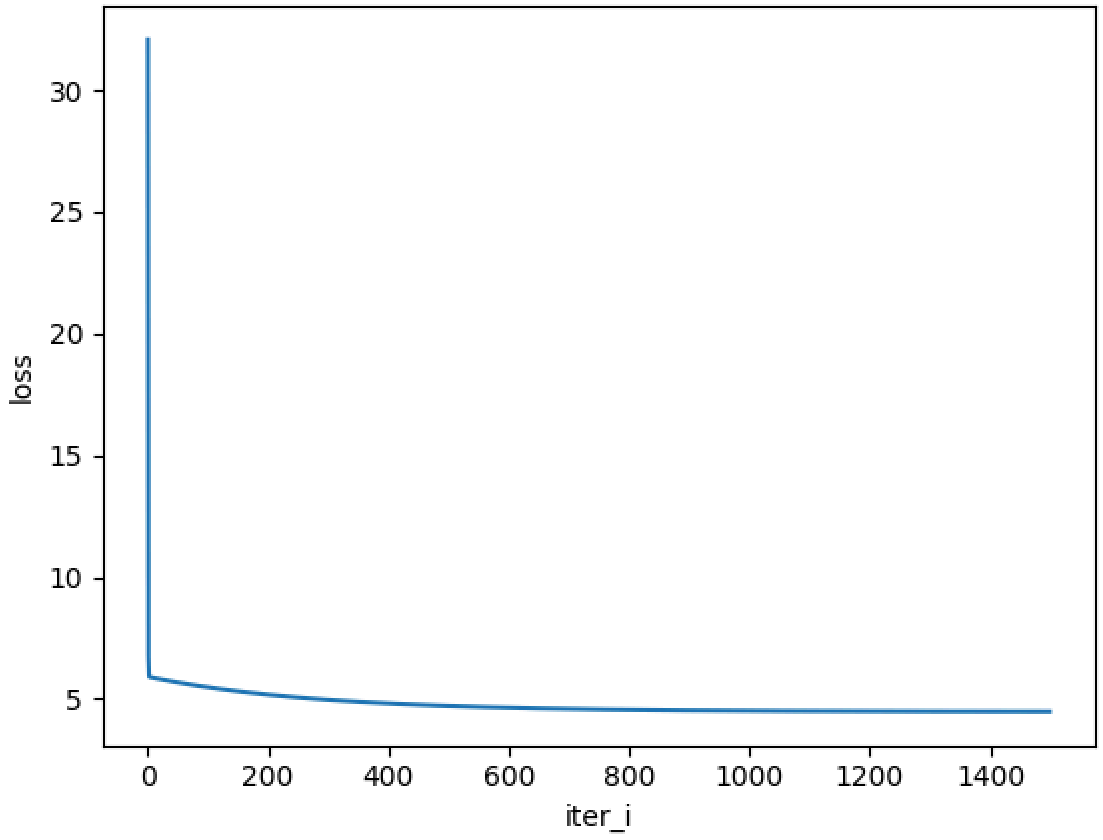
A good way to verify that gradient descent is working correctly is to look at the value of and check that it is decreasing with each step. Your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm. Another way of making sure your gradient estimate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

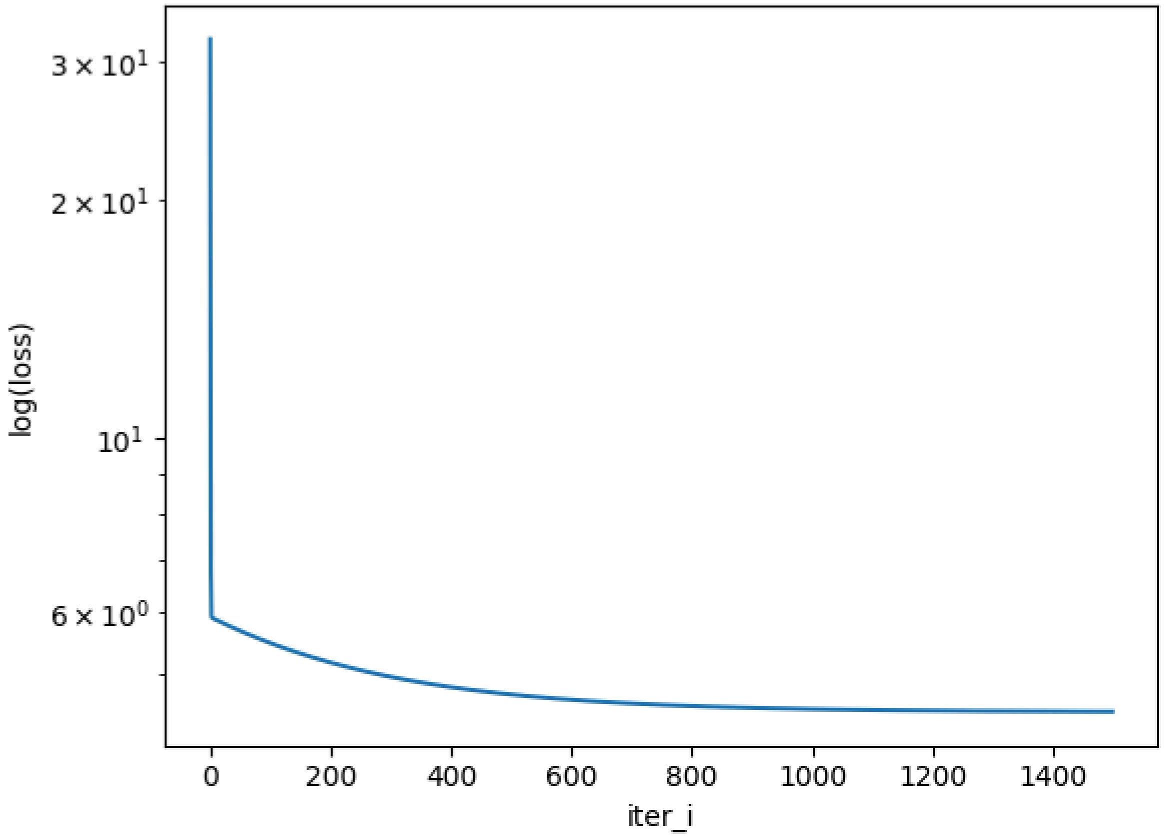
We also initialize the initial parameters to 0 and the learning rate alpha to `0.01`.

In [8]:
```
import scipy.optimize
from functools import partial

def loss_gradient(X, y, theta):
    X_prime = add_column(X)
    hypothesis = predict(X, theta)
    loss_grad = loss_grad = X_prime.T @ (hypothesis - y) / X_prime.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)
        loss_gradient_value = loss_gradient(X, y, theta_current)
        theta_current = theta_current - lr * loss_gradient_value
        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())
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()
```

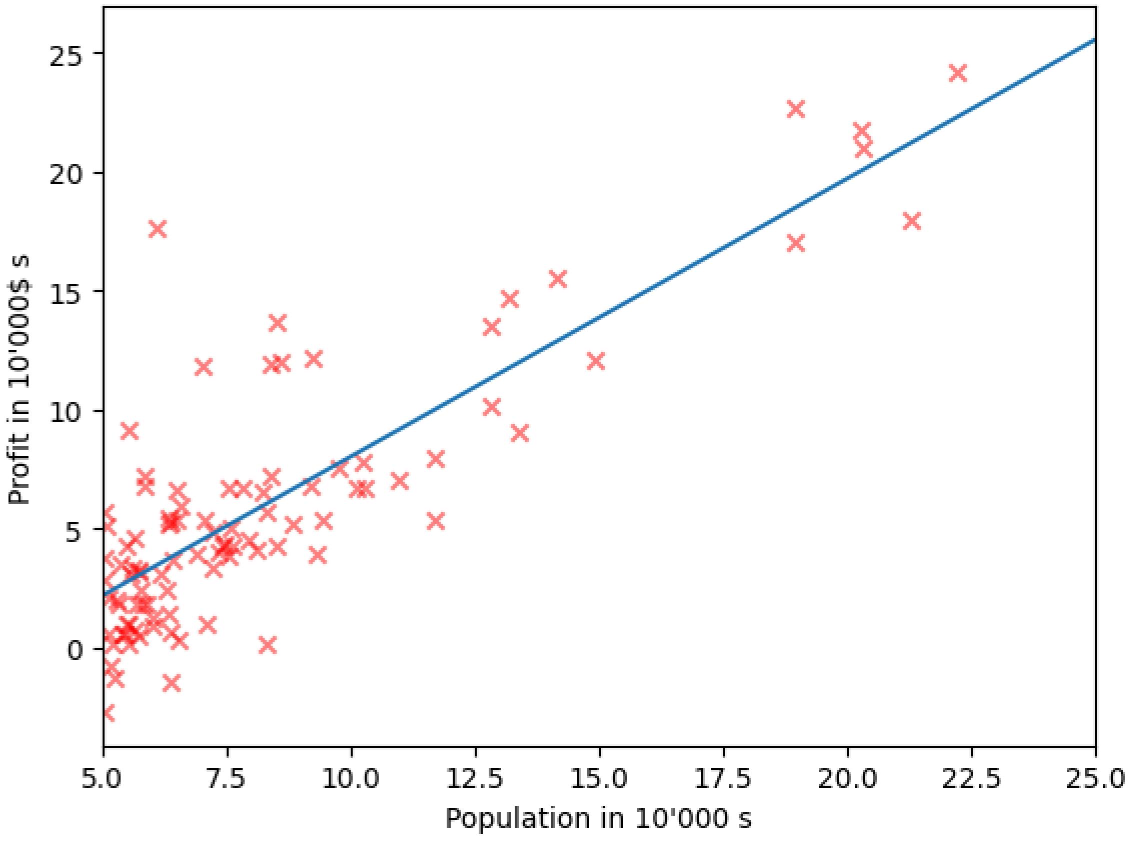
```
max grad comp error 1.9431944416226088e-05
estimated theta value [-3.63029144  1.16636235]
resulting loss [4.48338826]
```





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.

In [9]:
```
plt.scatter(X, y, marker='x', color='r', alpha=0.5)
x_start, x_end = 5, 25
x = np.linspace(x_start, x_end).reshape(-1, 1) # get the 50 points in line s
prediction = predict(x, theta_est)
plt.plot(x, prediction)
plt.xlim(x_start, x_end)
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.

In [10]:
```
area1, area2 = 35000, 70000
areas = np.array([area1, area2]).reshape(-1, 1)
prediction = predict(areas, theta_est)
print(prediction)
```

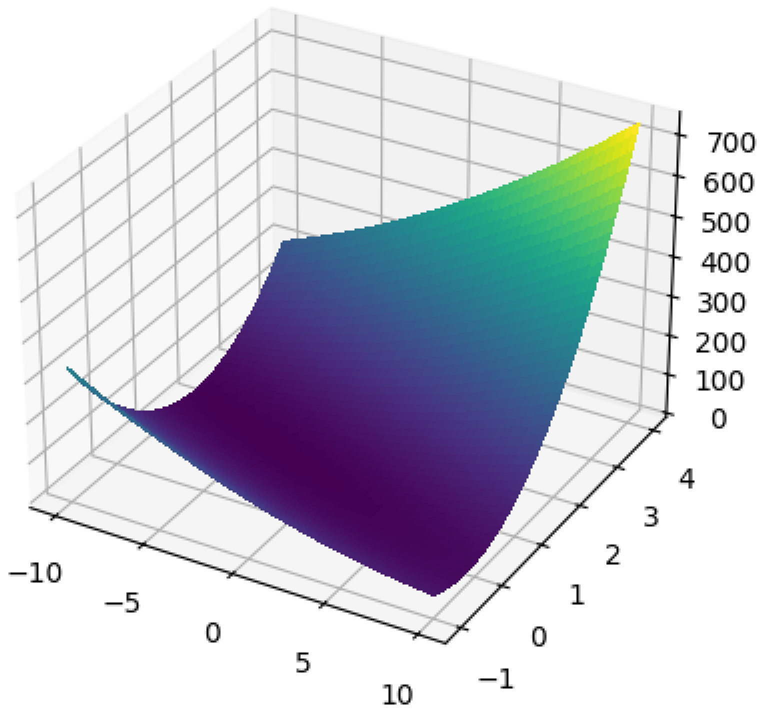
```
[[40819.05197031]
 [81641.73423205]]
```

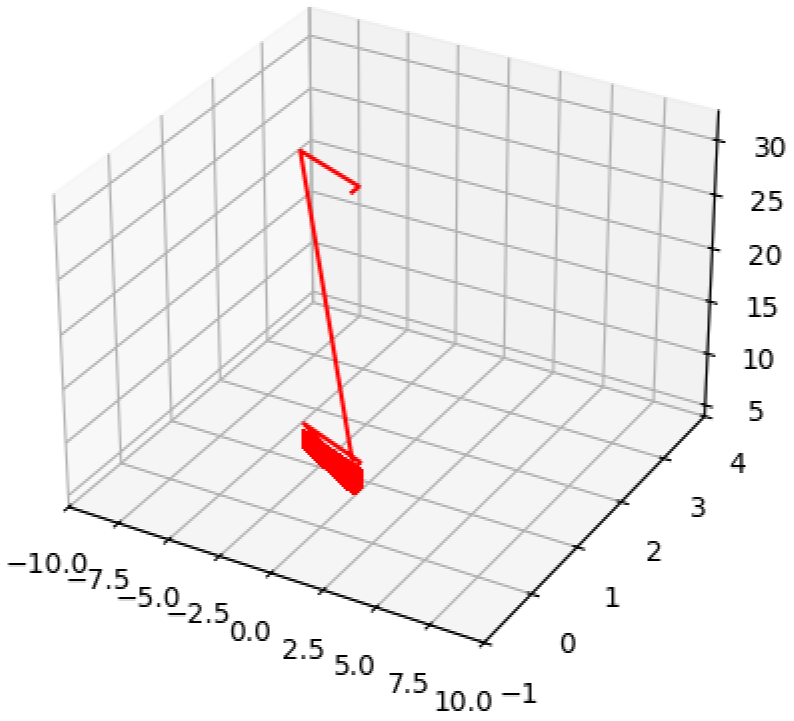
To understand the cost function better, you will now plot the cost over a 2-dimensional grid of values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images.

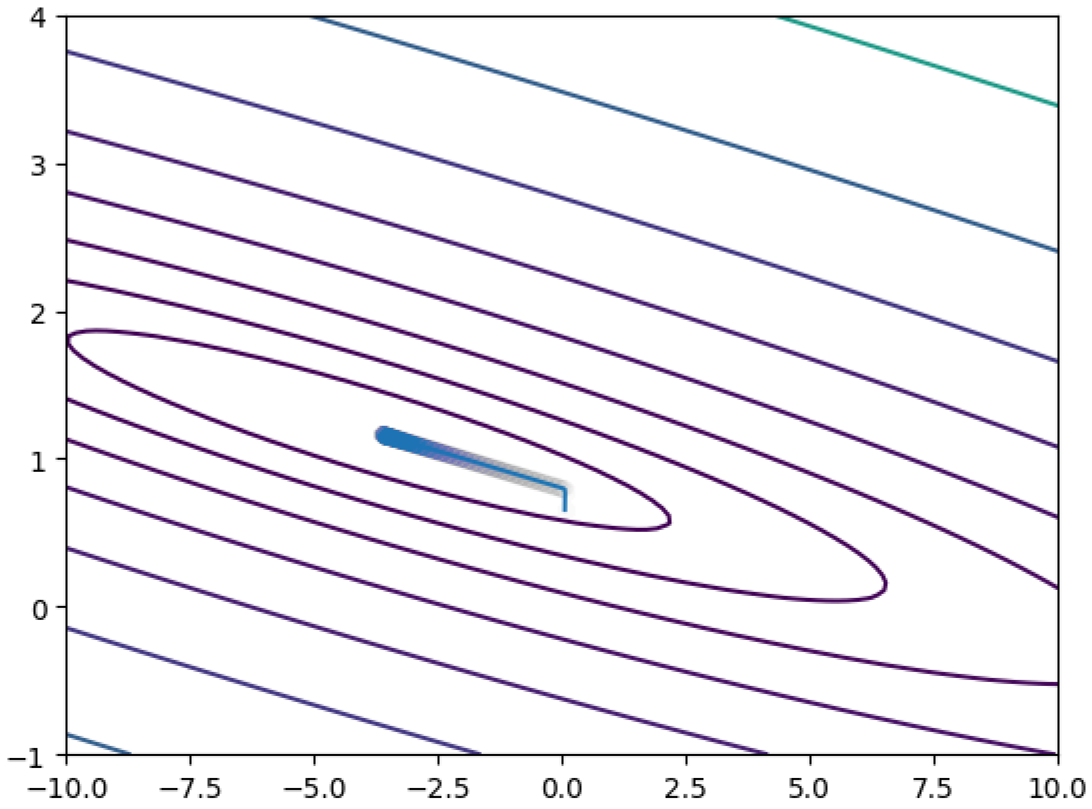
In [11]:
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.cm as cm
limits = [(-10, 10), (-1, 4)]
```

```
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
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.figure().add_subplot(projection='3d').plot_surface(theta_1_grid, theta_2
                                       loss_test_vals_grid, cmap=cm.viridis,
                                       linewidth=0, antialiased=False)
xs, ys = np.hstack(theta_values).tolist()
zs = np.array(loss_values)
plt.figure().add_subplot(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.logsp
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)
```




30
25
20
15
10
5
4
3
2
1
0
−1
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


4
3
2
1
0
−1
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

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.)

In [12]:
```
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])
```

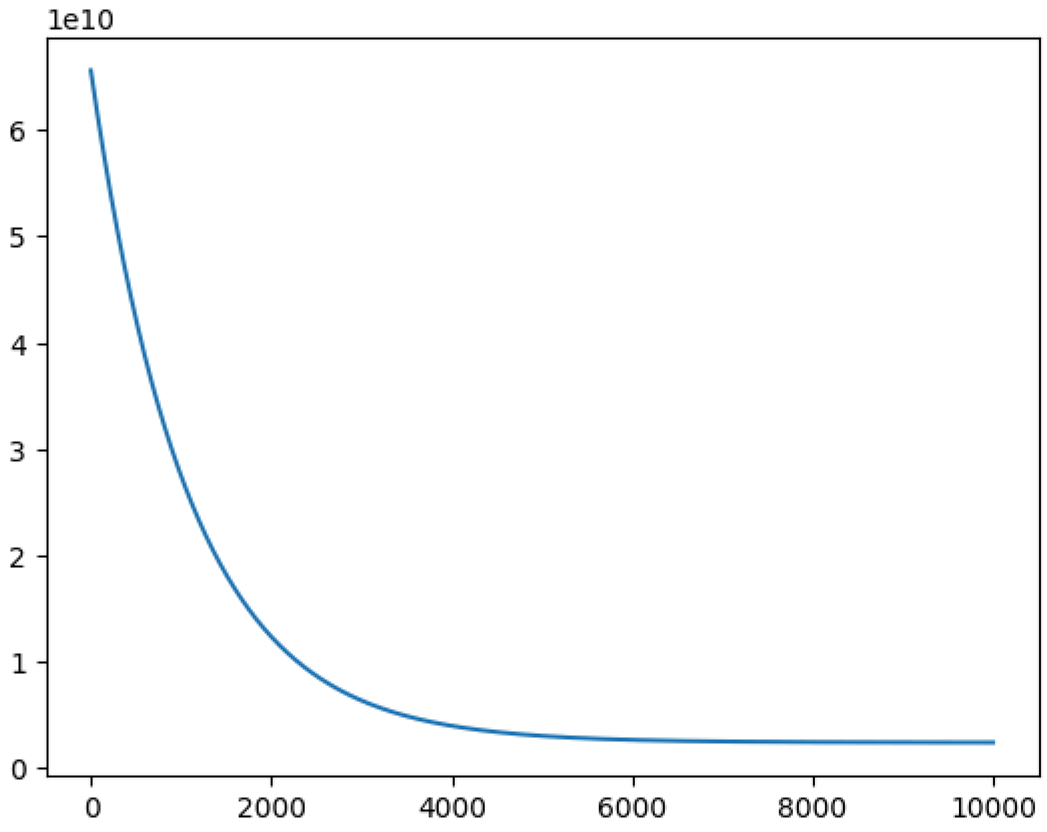
```
(47, 2) (47, 1) 47
[[2.104e+03 3.000e+00]
 [1.600e+03 3.000e+00]
 [2.400e+03 3.000e+00]
 [1.416e+03 2.000e+00]
 [3.000e+03 4.000e+00]
 [1.985e+03 4.000e+00]
 [1.534e+03 3.000e+00]
 [1.427e+03 3.000e+00]
 [1.380e+03 3.000e+00]
 [1.494e+03 3.000e+00]]
 [[399900.]
 [329900.]
 [369000.]
 [232000.]
 [539900.]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

In [13]:
```
X_new =  np.insert(X, 0, 1, axis=1)
print(X_new)
print(X_new.shape)
```

```
[[1.000e+00 2.104e+03 3.000e+00]
 [1.000e+00 1.600e+03 3.000e+00]
 [1.000e+00 2.400e+03 3.000e+00]
 [1.000e+00 1.416e+03 2.000e+00]
 [1.000e+00 3.000e+03 4.000e+00]
 [1.000e+00 1.985e+03 4.000e+00]
 [1.000e+00 1.534e+03 3.000e+00]
 [1.000e+00 1.427e+03 3.000e+00]
 [1.000e+00 1.380e+03 3.000e+00]
 [1.000e+00 1.494e+03 3.000e+00]
 [1.000e+00 1.940e+03 4.000e+00]
 [1.000e+00 2.000e+03 3.000e+00]
 [1.000e+00 1.890e+03 3.000e+00]
 [1.000e+00 4.478e+03 5.000e+00]
 [1.000e+00 1.268e+03 3.000e+00]
 [1.000e+00 2.300e+03 4.000e+00]
 [1.000e+00 1.320e+03 2.000e+00]
 [1.000e+00 1.236e+03 3.000e+00]
 [1.000e+00 2.609e+03 4.000e+00]
 [1.000e+00 3.031e+03 4.000e+00]
 [1.000e+00 1.767e+03 3.000e+00]
 [1.000e+00 1.888e+03 2.000e+00]
 [1.000e+00 1.604e+03 3.000e+00]
 [1.000e+00 1.962e+03 4.000e+00]
 [1.000e+00 3.890e+03 3.000e+00]
 [1.000e+00 1.100e+03 3.000e+00]
 [1.000e+00 1.458e+03 3.000e+00]
 [1.000e+00 2.526e+03 3.000e+00]
 [1.000e+00 2.200e+03 3.000e+00]
 [1.000e+00 2.637e+03 3.000e+00]
 [1.000e+00 1.839e+03 2.000e+00]
 [1.000e+00 1.000e+03 1.000e+00]
 [1.000e+00 2.040e+03 4.000e+00]
 [1.000e+00 3.137e+03 3.000e+00]
 [1.000e+00 1.811e+03 4.000e+00]
 [1.000e+00 1.437e+03 3.000e+00]
 [1.000e+00 1.239e+03 3.000e+00]
 [1.000e+00 2.132e+03 4.000e+00]
 [1.000e+00 4.215e+03 4.000e+00]
 [1.000e+00 2.162e+03 4.000e+00]
 [1.000e+00 1.664e+03 2.000e+00]
 [1.000e+00 2.238e+03 3.000e+00]
 [1.000e+00 2.567e+03 4.000e+00]
 [1.000e+00 1.200e+03 3.000e+00]
 [1.000e+00 8.520e+02 2.000e+00]
 [1.000e+00 1.852e+03 4.000e+00]
 [1.000e+00 1.203e+03 3.000e+00]]
(47, 3)
```

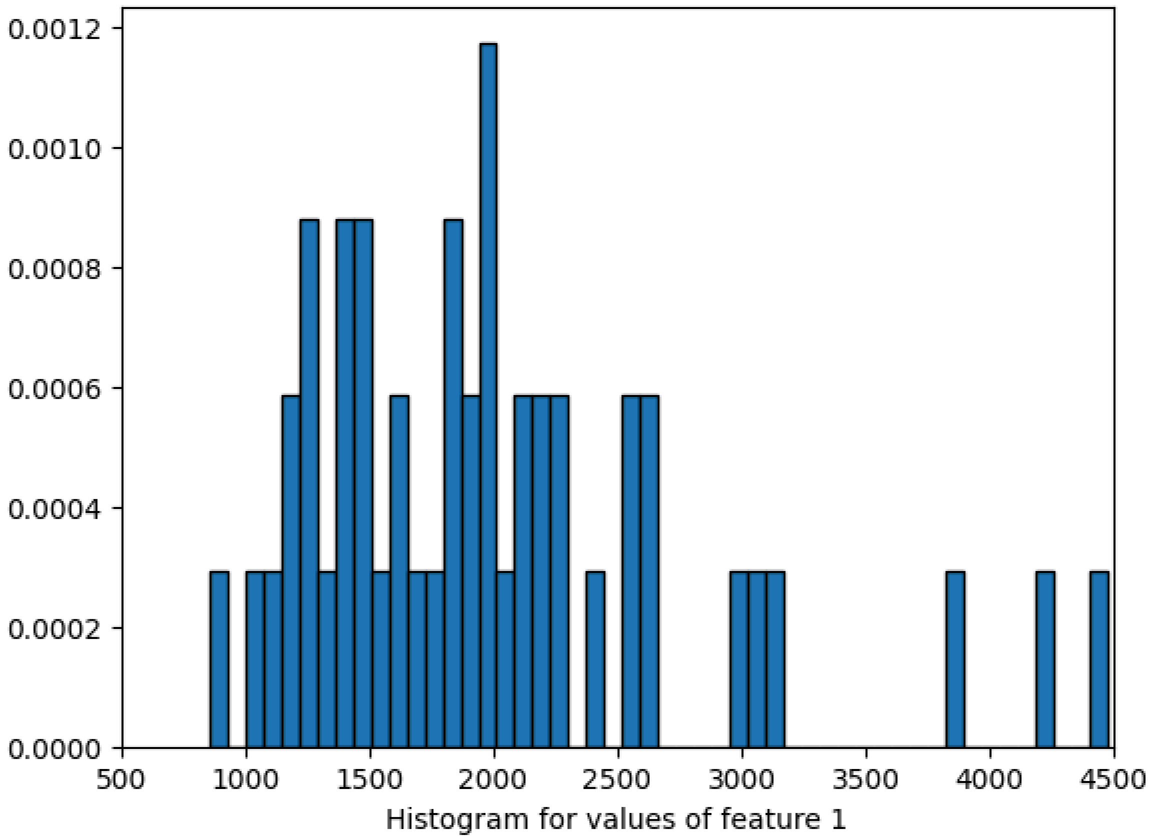
In [14]:

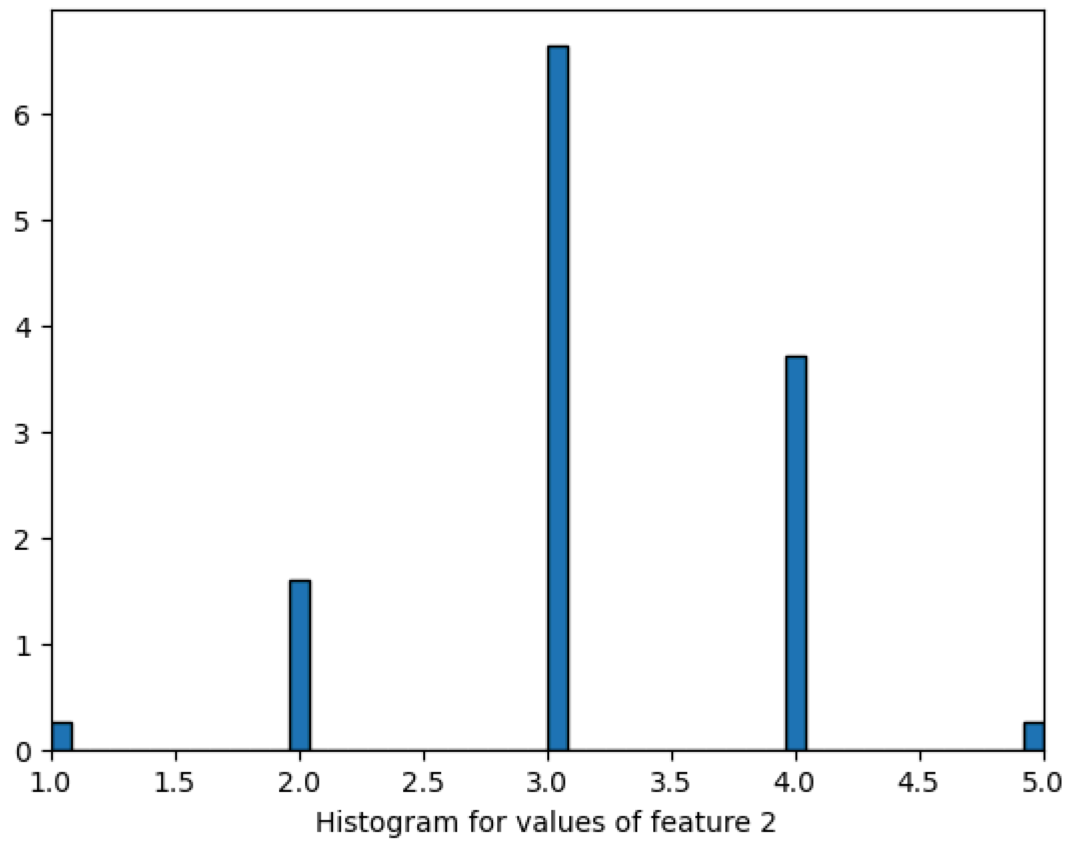
```
def add_column(X):
    X = np.insert(X, 0, 1, axis=1)
    return X

def predict(X, theta):
    """ Computes h(x; theta) """
    X_prime = add_column(X)
    pred = X_prime @ theta

    return pred

def loss(X, y, theta): # this is SSD, basically
    X_prime = add_column(X)
    hypothesis = predict(X, theta)
    loss = (np.dot((hypothesis - y).T, (hypothesis - y)) / (2 * X_prime.shap

    return loss

def loss_gradient(X, y, theta):
    X_prime = add_column(X)
    hypothesis = predict(X, theta)
    loss_grad = loss_grad = X_prime.T @ (hypothesis - y) / X_prime.shape[0]

    return loss_grad

theta_init = np.zeros((3, 1))
result = run_gd(loss, loss_gradient, X, y, theta_init, n_iter=10000, lr=1e-1
theta_est, loss_values, theta_values = result
plt.plot(loss_values)
plt.show()
```



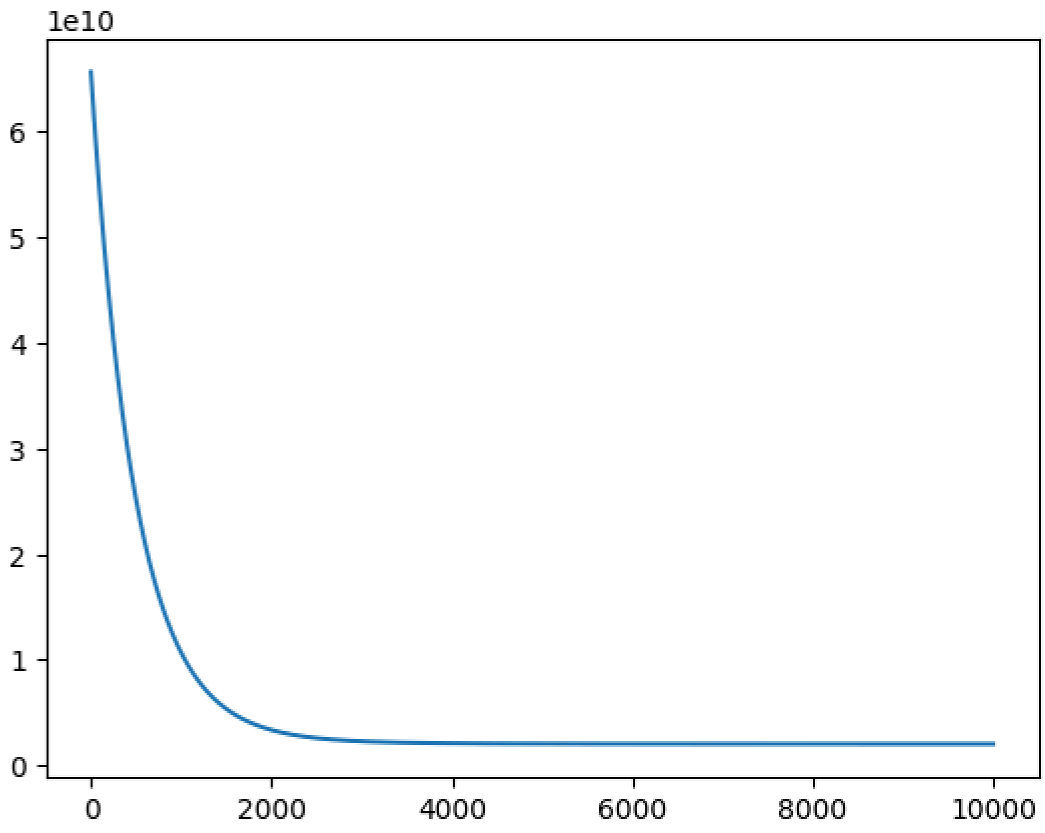
3.2 [20pt] Draw a histogam of values for the first and second feature. Why is feature normalization important? Normalize features and re-run the gradient decent. Compare loss plots that you get with and without feature normalization.

In [15]:
```
plt.hist(X[:,0], bins=50, density=True, edgecolor="black")
plt.xlim(500, 4500)
plt.xlabel("Histogram for values of feature 1")
plt.show()


plt.hist(X[:,1], bins=50, density=True, edgecolor="black")
plt.xlim(1, 5)
plt.xlabel("Histogram for values of feature 2")
plt.show()
```





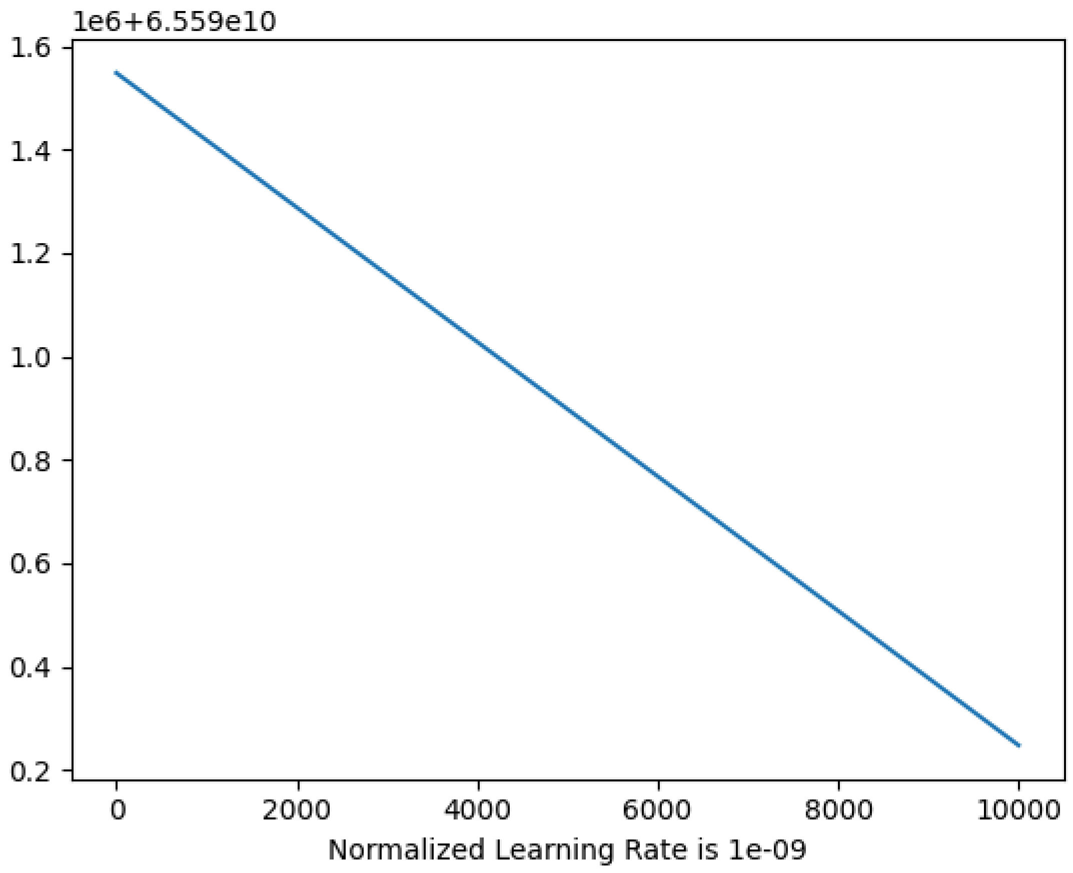
In [16]:
```
theta_init = np.zeros((3, 1))
X_normed = np.zeros_like(X) # For normalization, I used ((each data - columr
X_normed[:,0] = ((X[:,0] - np.mean(X[:,0])) / np.std(X[:,0]))
X_normed[:,1] = ((X[:,1] - np.mean(X[:,1]))/ np.std(X[:,1]))
# raise NotImplementedError("Run gd on normalized versions of feature vector
result = run_gd(loss, loss_gradient, X_normed, y, theta_init, n_iter=10000,
theta_est, loss_values, theta_values = result

plt.plot(loss_values)
plt.show()
```

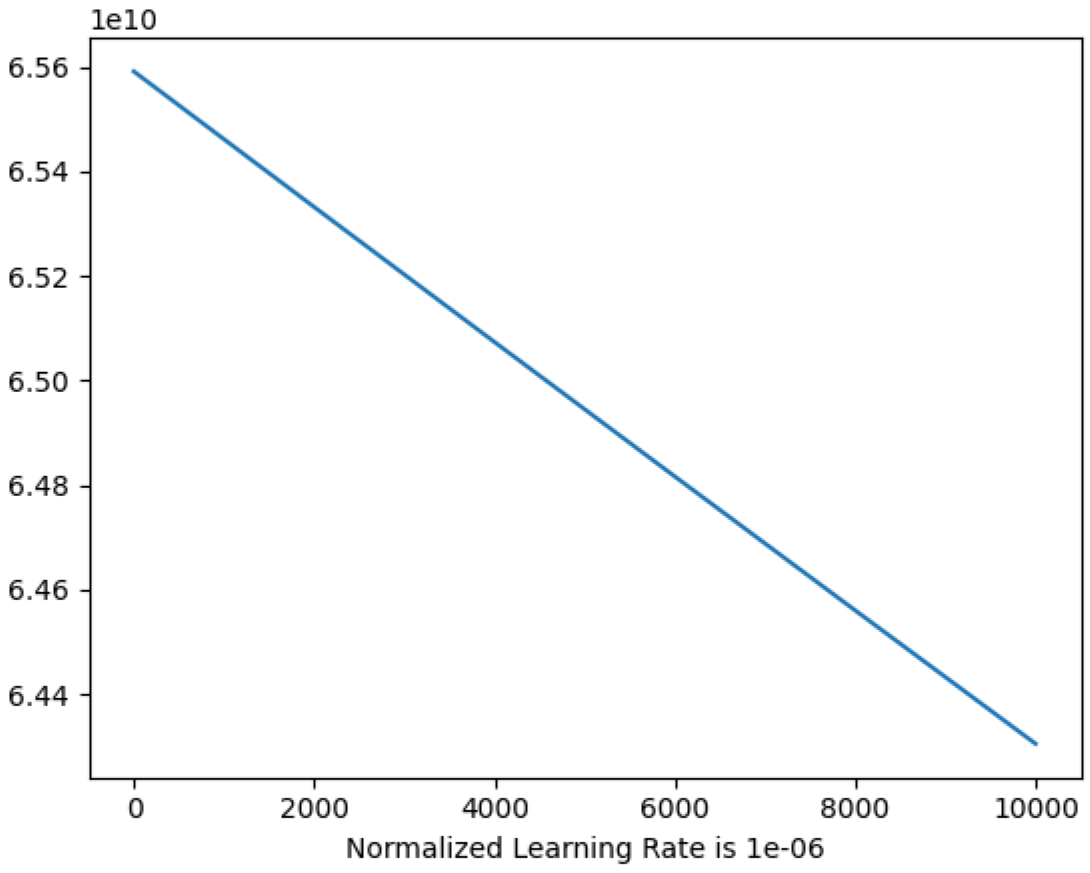


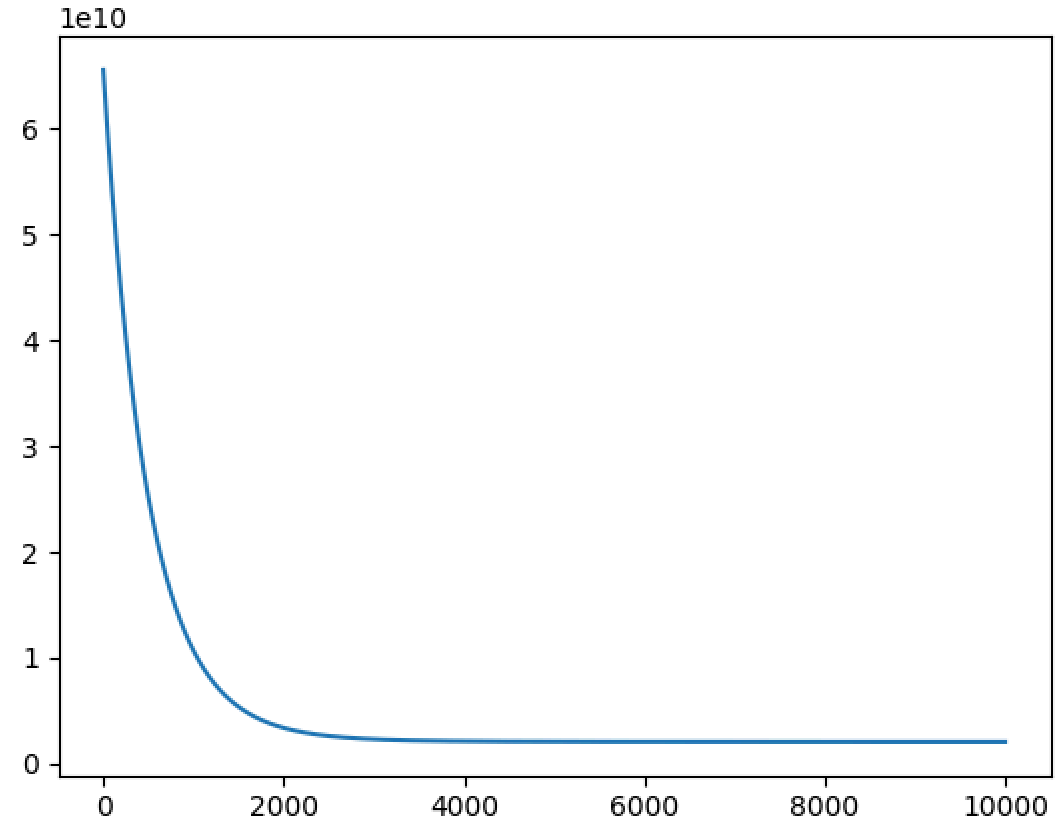
Looking at the two different loss plots, I can observe that the normalized X value loss plot reacts more sharply to zero. This implies that it will be faster for the gradient to converge to zero in the normalized X data compared to the regular X data. This is because the normalized X data is scaled to 0 as the mean, which allows the gradient to converge more quickly.

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?

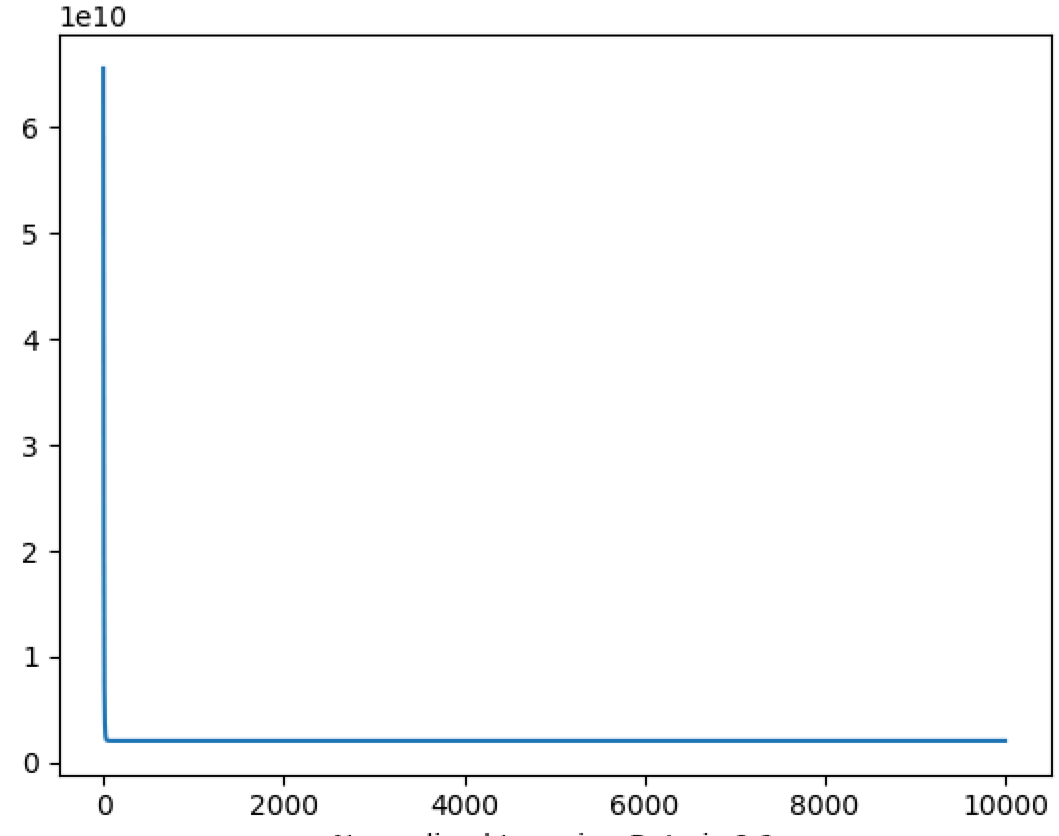
In [19]:
```
#Plot loss behaviour when with multiple different learning rates

learning_rate = [1e-9, 1e-6, 1e-3, 1e-1]

for lr in learning_rate:
    normalized_result = run_gd(loss, loss_gradient, X_normed, y, theta_init,
    normalized_theta_est, noarmalized_loss_values, normalized_theta_values =

    plt.plot(noarmalized_loss_values)
    plt.xlabel(f"Normalized Learning Rate is {lr}")
    plt.show()

for lr in learning_rate:
    result = run_gd(loss, loss_gradient, X, y, theta_init, n_iter=10000, lr=
    theta_est, loss_values, theta_values = result

    plt.plot(loss_values)
    plt.xlabel(f"Learning Rate is {lr}")
    plt.show()
```



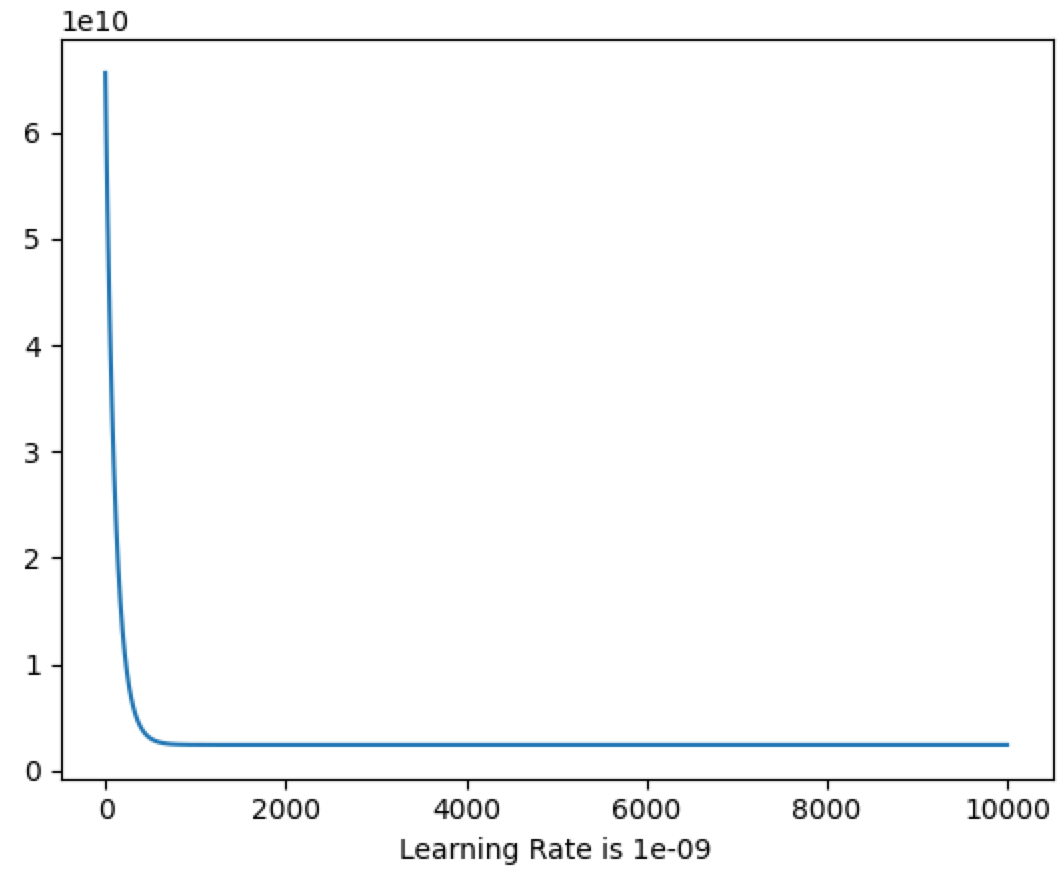




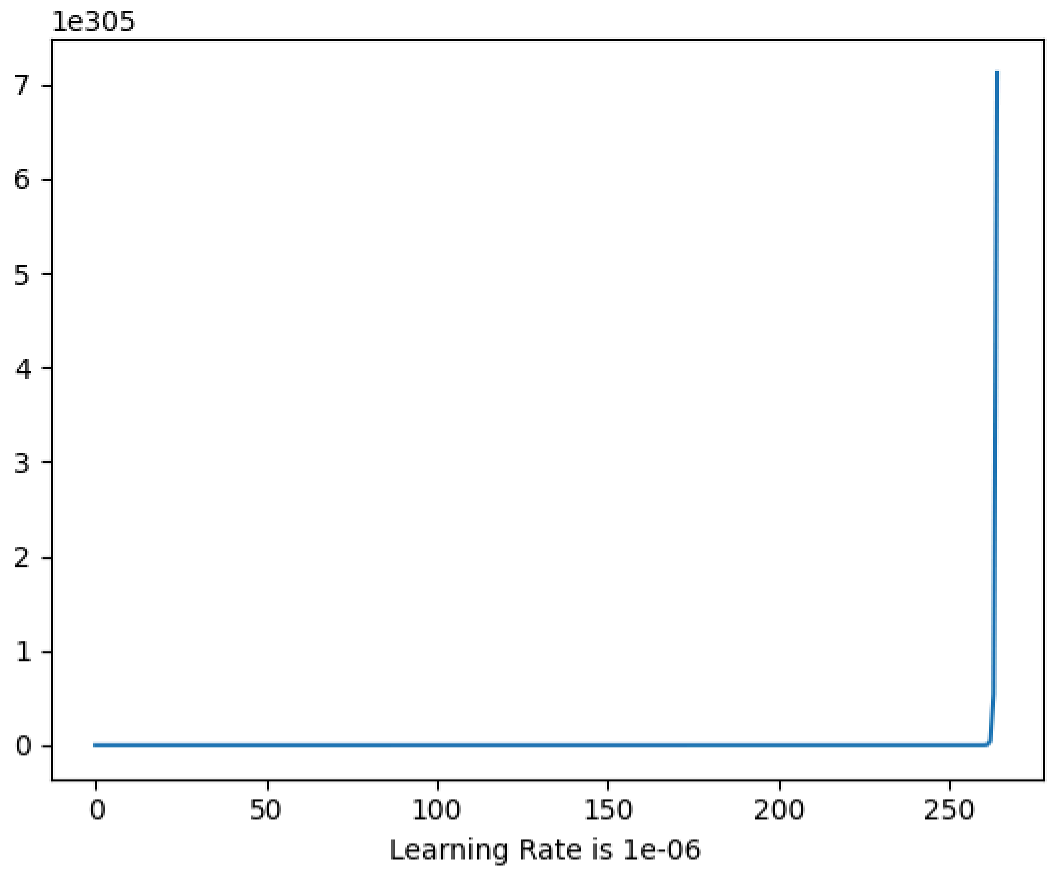
Normalized Learning Rate is 0.001

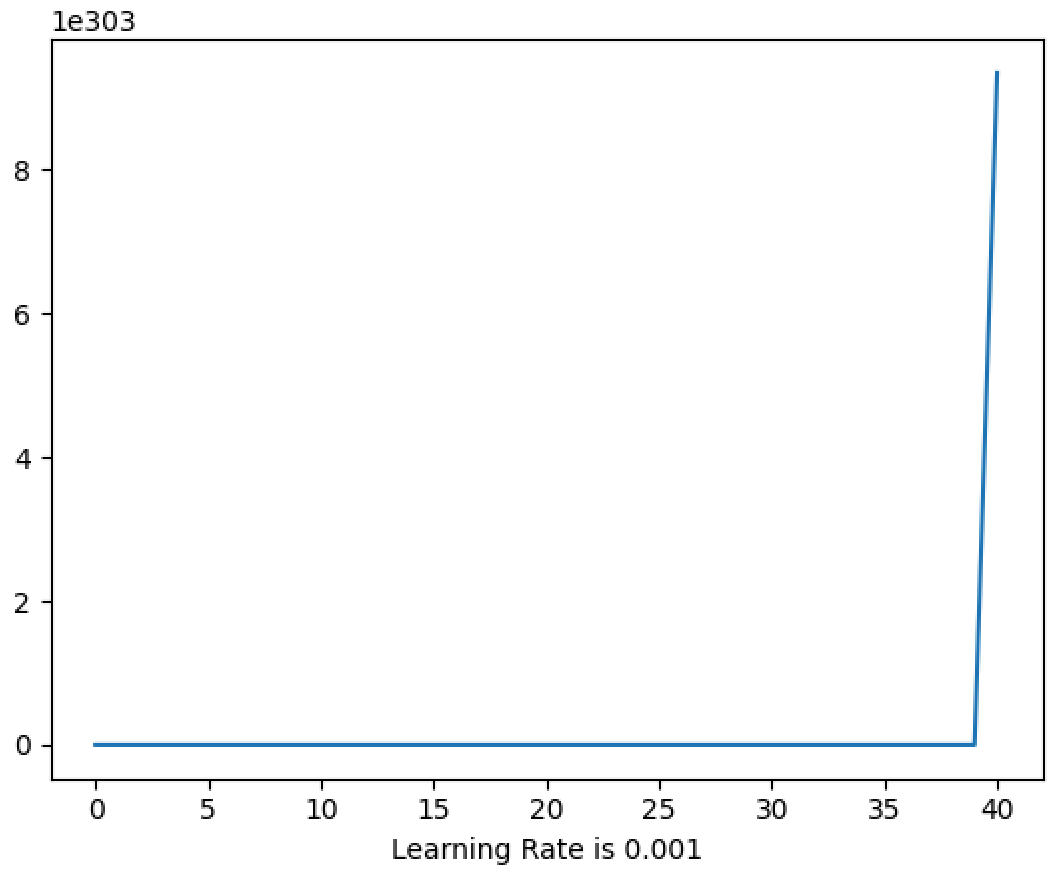


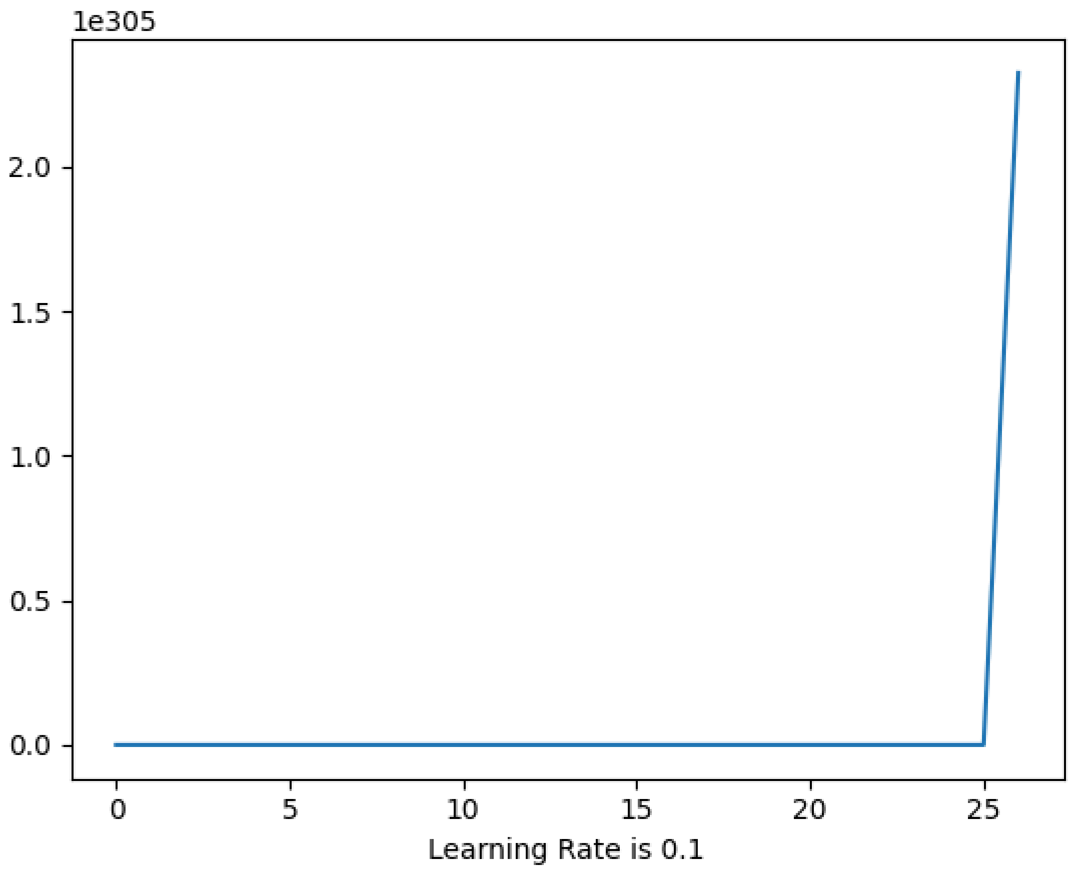
Normalized Learning Rate is 0.1



```
/var/folders/xt/y_17jz6n3p77v5p6cbm1tstr0000gn/T/ipykernel_9140/2357155297.p
y:22: RuntimeWarning: overflow encountered in matmul
  loss_grad = loss_grad = X_prime.T @ (hypothesis - y) / X_prime.shape[0] #
divide by the number of n to get the average difference
/var/folders/xt/y_17jz6n3p77v5p6cbm1tstr0000gn/T/ipykernel_9140/1840832893.p
y:42: RuntimeWarning: invalid value encountered in subtract
  theta_current = theta_current - lr * loss_gradient_value
```


1e305
7
6
5
4
3
2
1
0
0
50
100
150
200
250
Learning Rate is 1e-06


1e303
8
6
4
2
0
0
5
10
15
20
25
30
35
40
Learning Rate is 0.001



4. Written Questions

These problems are extremely important preparation for the exam. Submit solutions to each problem by filling the markdown cells below.

4.1 [10 pt] Maximum Likelihood Estimate for Coin Toss

The probability distribution of a single binary variable that takes value with probability is given by the Bernoulli distribution

$$\mathrm{Bern}(x|\mu) = \mu^x (1-\mu)^{1-x}$$

For example, we can use it to model the probability of seeing 'heads' ($x = 1$) or 'tails' ($x = 0$) after tossing a coin, with μ being the probability of seeing 'heads'. Suppose we have a dataset of independent coin flips $D = \{x^{(1)}, \ldots, x^{(m)}\}$ and we would like to estimate μ using Maximum Likelihood. Recall that we can write down the likelihood function as

$$\mathcal{L}(x^{(i)}|\mu) = \mu^{x^{(i)}} (1-\mu)^{1-x^{(i)}}$$

$$P(D|\mu) = \prod_i \mathcal{L}(x^{(i)}|\mu)$$

The log of the likelihood function is

$$\ln P(D|\mu) = \sum_i x^{(i)} \ln \mu + (1 - x^{(i)}) \ln(1-\mu)$$

Show that the ML solution for μ is given by $\mu_{ML} = \frac{h}{m}$ where h is the total number of 'heads' in the dataset. Show all of your steps.

[My Solution]

In order to get thw ML solution for μ, I need to set the derivative of the log of likelihood is 0. Let derivative the log of likelihodd function that the problem gave me here.

$$\ln P(D|\mu) = \sum_i x^{(i)} \ln \mu + (1 - x^{(i)}) \ln(1-\mu)$$

$$\frac{\partial}{\partial \mu} \ln P(D|\mu) = \frac{\partial}{\partial \mu} (\sum_i x^{(i)} \ln \mu + (1 - x^{(i)}) \ln(1-\mu))$$

$$\frac{\partial}{\partial \mu} \ln P(D|\mu) = \sum_i x^{(i)} \cdot \frac{1}{\mu} + \sum_i (1 - x^{(i)}) \cdot \frac{1}{1-\mu} \cdot (-1)$$

$$\frac{\partial}{\partial \mu} \ln P(D|\mu) = \sum_i \frac{x^{(i)}}{\mu} - \sum_i \frac{(1 - x^{(i)})}{1-\mu}$$

When derivative result of the maximum likelihood function is 0, it gives us the maximum outcome value of using the parameters. So put this equation to 0.

$$0 = \sum_i \frac{x^{(i)}}{\mu} - \sum_i \frac{(1 - x^{(i)})}{1-\mu}$$

$$\sum_i \frac{x^{(i)}}{\mu} = \sum_i \frac{(1 - x^{(i)})}{1-\mu}$$

Since h is the total number of 'heads' which is $x = 1$ in the dataset, $\sum_i x^{(i)} = h$ and $\sum_i (1 - x^{(i)}) = m - h$, when m is the total number of the dataset. So use this parameters instead.

$$\frac{h}{\mu} = \frac{(m-h)}{1-\mu}$$

$$(m-h) \times \mu = h \times (1-\mu)$$

$$m\mu - h\mu = h - h\mu$$

$$m\mu = h$$

$$\mu = \frac{h}{m}$$

The final value of μ is $\frac{h}{m}$.

4.2 [10 pt] Localized linear regression

Suppose we want to estimate localized linear regression by weighting the contribution of the data points by their distance to the query point x_q, i.e. using the cost

$$E(x_q) = \frac{1}{2}\sum_i^m \frac{(y^{(i)} - h(x^{(i)}|\theta))^2}{||x^{(i)} - x_q||^2}$$

where $\frac{1}{||x^{(i)}-x_q||} = w^{(i)}$ is the inverse Euclidean distance between the training point $x^{(i)}$ and query (test) point x_q.

Derive the modified normal equations for the above cost function $E(x_q)$. Hint: first, re-write the cost function in matrix/vector notation, using a diagonal matrix to represent the weights $w^{(i)}$.

[My Solution]

Let's rewrite the cost function in matrix/vector form using $w^{(i)}$. Since $\frac{1}{||x^{(i)}-x_q||} = w^{(i)}$, $\frac{1}{||x^{(i)}-x_q||^2} = w^{(i)^2}$.

$$E(x_q) = \frac{1}{2}\sum_i^m \frac{(y^{(i)} - h(x^{(i)}|\theta))^2}{||x^{(i)} - x_q||^2}$$

$$E(x_q) = \frac{1}{2}\sum_i^m (y^{(i)} - h(x^{(i)}|\theta))^2 \cdot \frac{1}{||x^{(i)} - x_q||^2}$$

$$E(x_q) = \frac{1}{2}\sum_i^m (y^{(i)} - h(x^{(i)}|\theta))^2 \cdot w^{(i)^2}$$

The inverse Euclidean distance, $w^{(i)}$ is a scalar value to the designated i value. But for all value from i to m, I can rewrite this as a diagnal matrix, $W = diag(w^i, w^2, w^3, \ldots, w^m)$. The $w^{(i)}$ term can come out of the summation notaton, $\sum_i^m$ by using W notation instead.

$$W = \begin{bmatrix} w^{(1)} & 0 & \cdots & 0 \\ 0 & w^{(2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & w^{(m)} \end{bmatrix}, m \times m \text{ matrix}$$

$$E(x_q) = \frac{1}{2} \cdot W^2 \cdot \sum_i^m (y^{(i)} - h(x^{(i)}|\theta))^2$$

Since W is a m x m diagonal matrix, W^2 is also a m x m diagonal matrix.

$$W^2 = \begin{bmatrix} w^{(1)^2} & 0 & \cdots & 0 \\ 0 & w^{(2)^2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & w^{(m)^2} \end{bmatrix}, m \times m \text{ matrix}$$

Just like the direct solution of SSD from the class, I can come up with the new notation,

$$X = \begin{bmatrix} --(X^{(1)})^T-- \\ --(X^{(2)})^T-- \\ \vdots \\ --(X^{(m)})^T-- \end{bmatrix}, y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Using these three new notations, $h(x^{(i)}|\theta) = X\theta$

$$E(x_q) = \frac{1}{2} \cdot W^2 \cdot (y - X\theta)^T(y - X\theta)$$

$(y - x\theta)$ is a $m \times 1$ vector, but W^2 is a $m \times m$ diagonal matrix. So I need to move W^2 for the dimension compatablities.

$$E(x_q) = \frac{1}{2} \cdot (y - X\theta)^T \cdot W^2 \cdot (y - X\theta)$$

Now the dimension of the cost function $E(x_q)$ is $(1 \times m) \cdot (m \times m) \cdot (m \times 1) = (1 \times 1)$, a scalar value, which is what we are looking for.

In order to derive the normal equation, I need to derivate the cost function and simplfy respect to θ.

From the matrix cookbook, the theorem (84): Assume W is symmetric, then

$$\frac{\partial}{\partial s}(x - As)^T W(x - As) = -2A^T W(x - As)$$

Use this to get the derivative of the cost function.

$$\frac{\partial}{\partial \theta}\{(y - X\theta)^T \cdot W^2 \cdot (y - X\theta)\} = -2 \cdot X^T \cdot W^2 \cdot (y - X\theta)$$

$$\frac{\partial}{\partial \theta}\{E(x_q)\} = \frac{1}{2} \cdot -2 \cdot X^T \cdot W^2 \cdot (y - X\theta) = -X^T \cdot W^2 \cdot (y - X\theta)$$

$$0 = -X^T \cdot W^2 \cdot (y - X\theta)$$

$$0 = -X^T W^2 y + X^T W^2 X\theta$$

$$X^T W^2 y = X^T W^2 X\theta$$

$$\theta = (X^T W^2 X)^{-1} \cdot X^T W^2 y$$

4.3 [10 pt] Betting on Trick Coins

A game is played with three coins in a jar: one is a normal coin, one has “heads” on both sides, one has “tails” on both sides. All coins are “fair”, i.e. have equal probability of landing on either side. Suppose one coin is picked randomly from the jar and tossed, and lands with “heads” on top. What is the probability that the bottom side is also “heads”? Show all your steps.

[My Solution]

The question is asking if there is the top side of the coin is given as "heads", get the probabilities that the other side of the coin is bottom. We know from the three different coins the only possibile coins of this case is the head, head coin. I can rewrite this $P(C_{H,H}|H)$, given that the top side is the "heads", the probability of this coin is head, head coin.

Using bayes theorem I can write this.

$$P(C_{H,H}|H) = \frac{P(H|C_{H,H}) \times P(C_{H,H})}{P(H)}$$

The $P(C_{H,H}) = \frac{1}{3}$ since there are only three coins. And $P(H|C_{H,H})$ is 1 beacuse $C_{H,H}$ only have heads.

$$P(H) = P(C_{T,H} \cap\ H) + P(C_{H,H} \cap\ H) + P(C_{T,T} \cap\ H) = \frac{1}{3} \times \frac{1}{2} + \frac{1}{3} \times 1 + \frac{1}{3} \times 0$$

so

$$P(C_{H,H}|H) = \frac{P(H|C_{H,H}) \times P(C_{H,H})}{P(H)} = \frac{1 \times \frac{1}{3}}{\frac{1}{2}} = \frac{2}{3}$$