Problem Set 1: 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, run Anaconda and choose this file in Anaconda's file explorer. Use `Python 3` version. Below statements assume 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. 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.

Only **PDF** files are accepted for ps1 submission. To print this notebook to a pdf file, you can go to "File" -> "Download as" -> "PDF via LaTex(.pdf)" or simply use "print" in browser.

Total: 185 points.

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 [384…

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

```
a = np.array([[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.]
print(a)
```

```
[[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 [385…

```
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))
```

```
[[2.86368908]
 [2.53700631]
 [1.54899392]
 [2.6264359 ]
 [2.53970349]
 [1.87195178]
 [2.12812452]
 [1.80727753]
 [2.8129602 ]
 [1.88062138]]
```

In [386…

```
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
```

```
[[2.86368908]
 [2.53700631]
 [1.54899392]
 [2.6264359 ]
 [2.53970349]
 [1.87195178]
 [2.12812452]
 [1.80727753]
 [2.8129602 ]
 [1.88062138]]
```

In [387…

```
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('---')
```

```
81.4 µs ± 4.9 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
The slowest run took 5.46 times longer than the fastest. This could mean that an intermediate resu
lt is being cached.
9.16 µs ± 6.17 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
---
532 µs ± 152 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
The slowest run took 122.67 times longer than the fastest. This could mean that an intermediate re
sult is being cached.
44.6 µs ± 76.1 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
---
31 ms ± 2.83 ms per loop (mean ± std. dev. of 7 runs, 5 loops each)
The slowest run took 25.73 times longer than the fastest. This could mean that an intermediate res
ult is being cached.
13.5 µs ± 25.6 µs per loop (mean ± std. dev. of 7 runs, 5 loops each)
---
753 ms ± 13.4 ms per loop (mean ± std. dev. of 7 runs, 5 loops each)
```

```
The slowest run took 11.21 times longer than the fastest. This could mean that an intermediate res
ult is being cached.
35.7 µs ± 51.6 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.

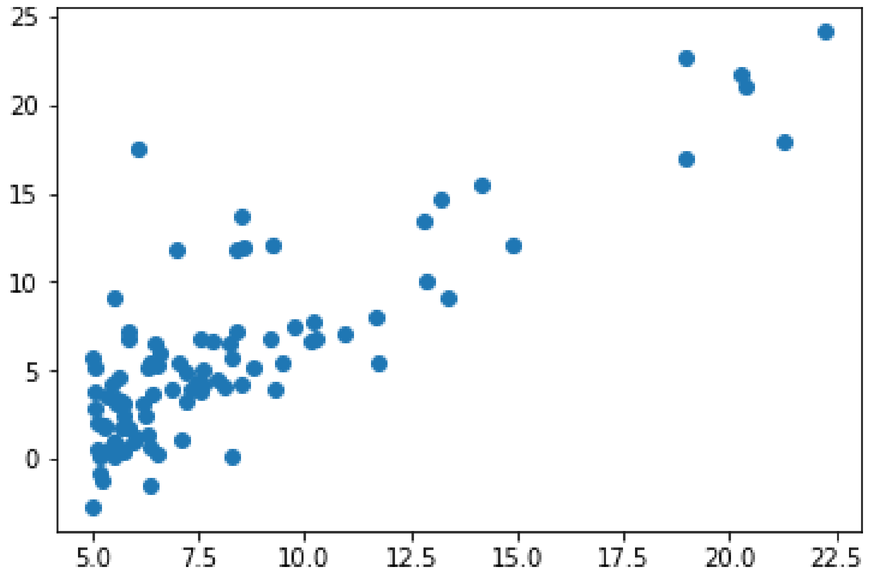
2.1 [10pt] Get a plot similar to below : [1] [2] [3]

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 [429…

```
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])

plt.scatter(X,y)

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?

Gradient Descent. Update by subtracting by gradient of the current cost until convergence.

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 [430…

```
# 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

    return np.insert(X, 0, np.ones(X.shape[0]), axis = 1)
    #raise NotImplementedError("Insert a column of ones to the _left_ side of the matrix")

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 = None
    #raise NotImplementedError("Compute the regression predictions")
    pred = np.dot(np.transpose(theta),np.transpose(X_prime))
    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)

    hypothesis = predict(X, theta)
    #print(hypothesis.shape)
    #print(y.shape)
    loss = np.transpose(hypothesis) - y
    m = X.shape[0]
    cost = np.sum(loss**2)/(2*m)
    #raise NotImplementedError("Compute the model loss; use the predict() function")
    #loss = None
    #return loss
    return cost
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 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.

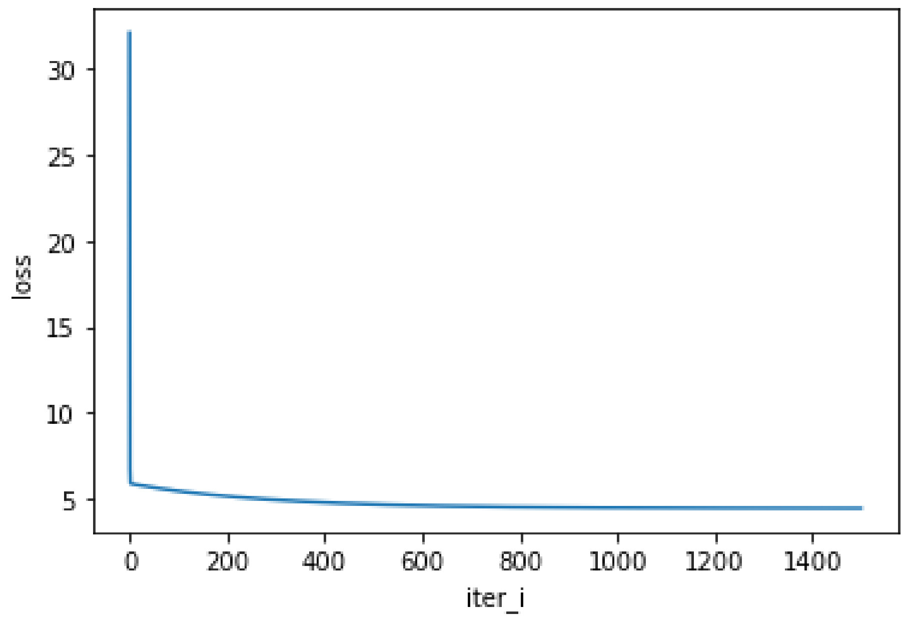
A good way to verify that gradient descent is working correctly is to look at the value of and check that it is decreasing with each step. Your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm. Another way of making sure your gradient estimate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

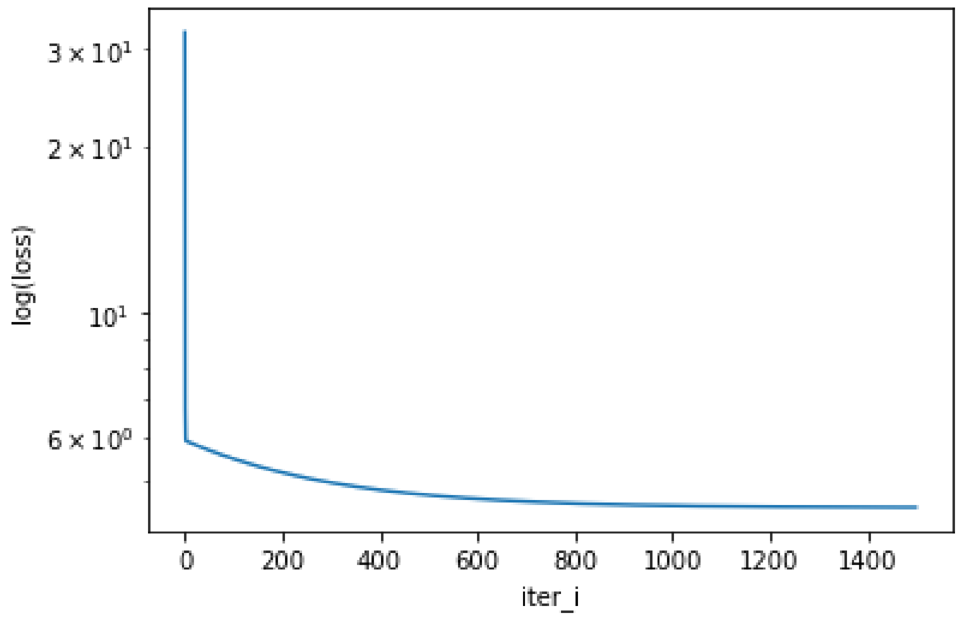
We also initialize the initial parameters to 0 and the learning rate alpha to `0.01` .

In [433…

```
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!")
    hypothesis = predict(X, theta)
    loss = np.transpose(hypothesis) - y
    loss_grad = np.dot(np.transpose(X_prime),loss)/X.shape[0]
    #print(loss_grad.shape)
    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")
        m = X.shape[0]
        gradient = loss_gradient(X,y, theta_current)
        theta_current = theta_current - lr * gradient

        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 3.639020753354316e-05
estimated theta value [-3.63029144  1.16636235]
resulting loss 4.483388256587726
```

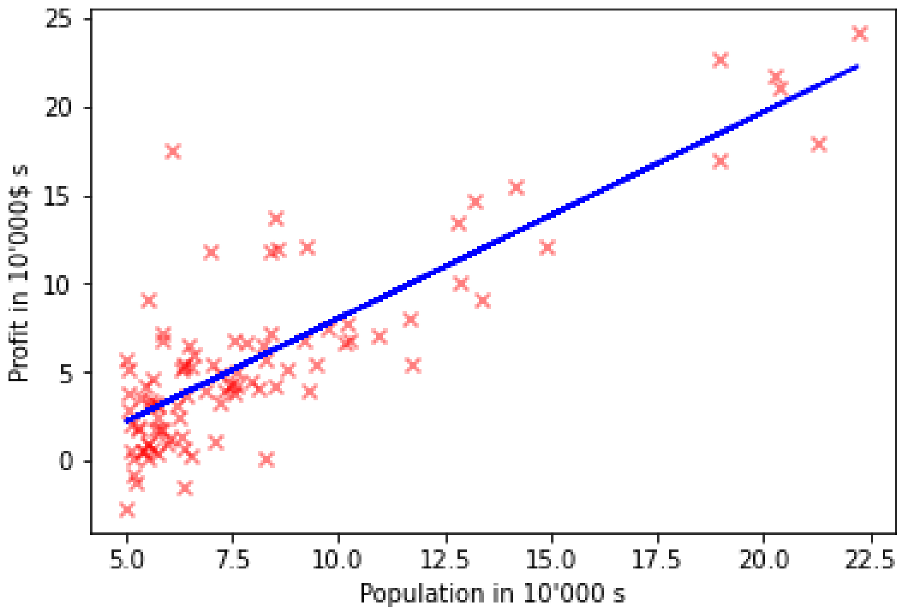




2.2.4 [10pt] After you are finished, use your final parameters to plot the linear fit. The result should look something like on the figure below. Use the `predict()` function.

In [434…

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


#print(theta_values)
#raise NotImplementedError("Put code that plots a regression line here")

plt.xlabel('Population in 10\'000 s')
plt.ylabel('Profit in 10\'000$ s')
plt.plot(X, predict(X, theta_est).transpose(), color = 'b')
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 [435…

```
#raise NotImplementedError("Predict values given inputs")
new_areas = np.array([[35000,70000]]).transpose()
print(predict(new_areas, theta_est).transpose())
```

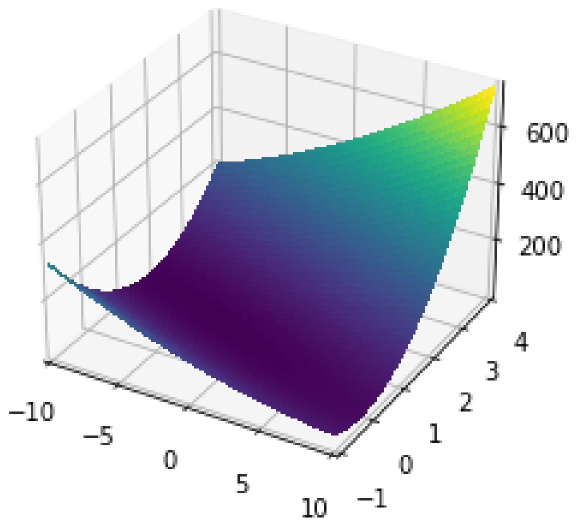
```
[[40819.05197031]
 [81641.73423205]]
```

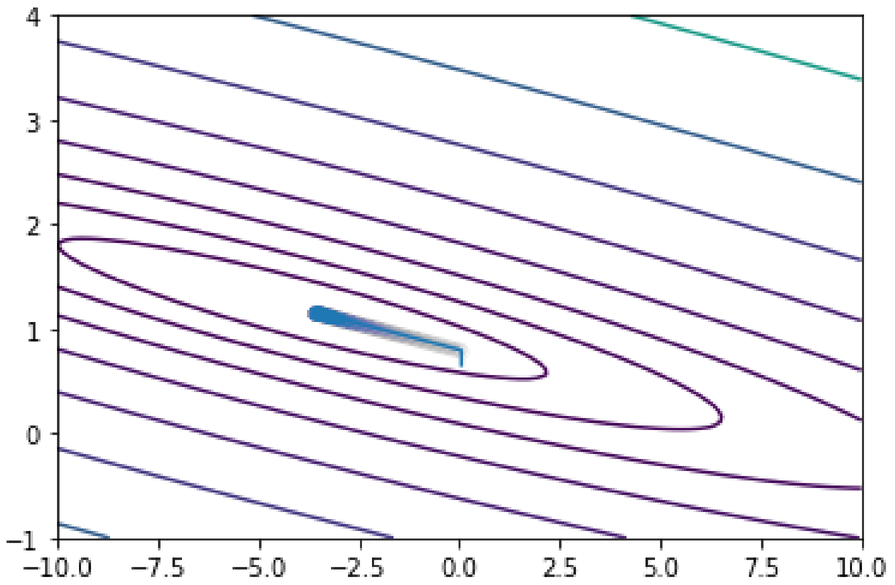
To understand the cost function better, you will now plot the cost over a 2-dimensional grid of values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images.

In [436…

```
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)
```





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 [465…

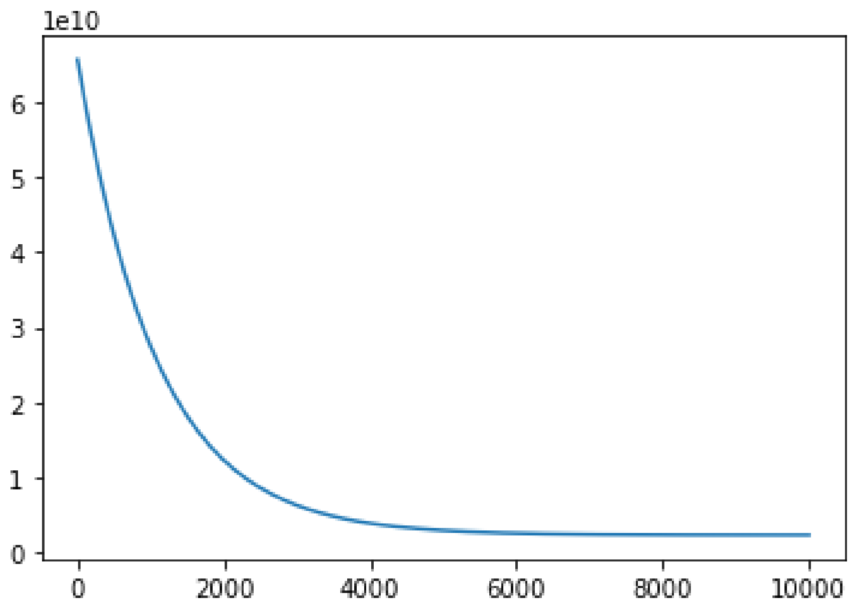
```
data = np.loadtxt('ex1data2.txt', delimiter=',')
X, y = data[:, :-1], data[:, -1, np.newaxis]
n = data.shape[0]
print(X.shape, y.shape, n)
print(X[:10], '\n', y[:10])
```

```
(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 [466…

```
#raise NotImplementedError("Implement new add_column(), predict(), loss(), loss_gradient() here fo


def add_column(X):
    #assert len(X.shape) == 2 and X.shape[1] == 1

    return np.insert(X, 0, np.ones(X.shape[0]), axis = 1)
    #raise NotImplementedError("Insert a column of ones to the _left_ side of the matrix")
```

```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)
    #pred = None
    #raise NotImplementedError("Compute the regression predictions")
    pred = np.dot(np.transpose(theta),np.transpose(X_prime))
    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)

    hypothesis = predict(X, theta)
    loss = np.transpose(hypothesis) - y
    m = X.shape[0]
    #print(loss)
    cost = np.sum(pow(loss,2))/(2*m)
    #raise NotImplementedError("Compute the model loss; use the predict() function")
    #loss = None
    #return loss
    return cost

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!")
    hypothesis = predict(X, theta)
    loss = np.transpose(hypothesis) - y
    loss_grad = np.dot(np.transpose(X_prime),loss)/X.shape[0]
    #print(loss_grad.shape)
    return loss_grad

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.show()

# raise NotImplementedError("Put your multivariate regression code here")
```

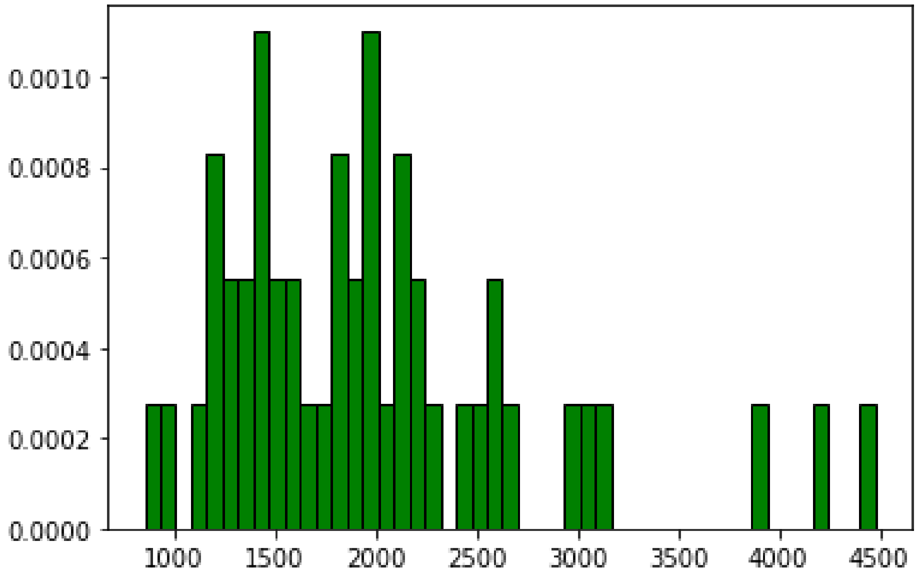


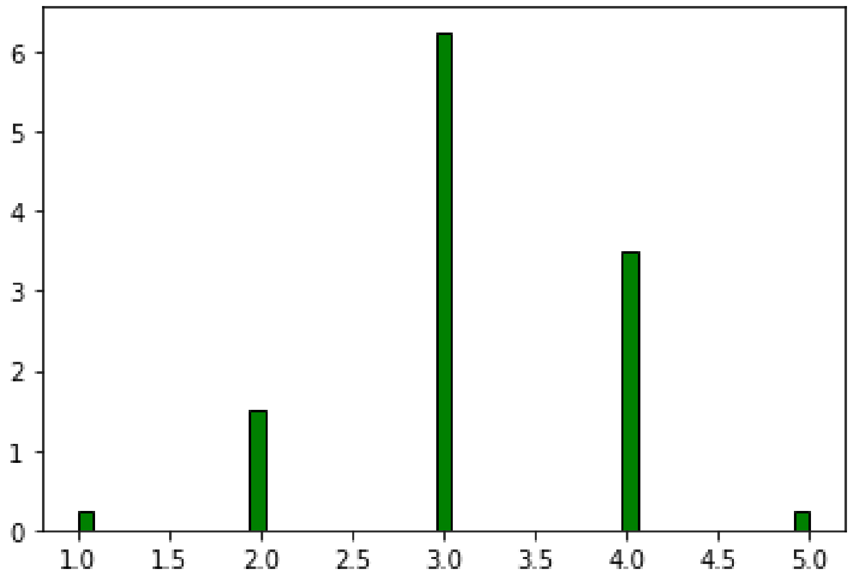
3.2 [20pt] Draw a histogam of values for the first and second feature. Why is feature normalization important? Normalize features and re-run the gradient decent. Compare loss plots that you get with and without feature normalization.

Feature normalization is important because features have very differnt scale The cost is higher for normalized data, but can achieve higher learning rate

In [494…

```
#raise NotImplementedError("Draw histogram for values of feature 1")
plt.hist(X.transpose()[0], bins = X.shape[0], density = X.shape[1], color = 'g', edgecolor= 'black
plt.show()

#raise NotImplementedError("Draw histogram for values of feature 2")
plt.hist(X.transpose()[1], bins = X.shape[0], density = X.shape[1], color = 'g', edgecolor= 'black
plt.show()
```

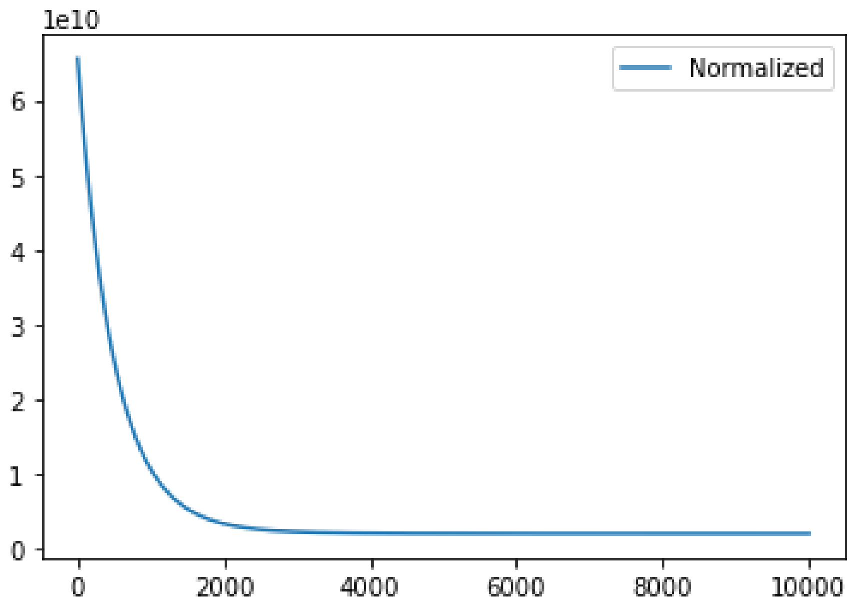




In [495…

```
theta_init = np.zeros((3, 1))

#Normalize X

#X_normed = np.zeros_like(X)
X_normed = np.copy(X)
mean = np.mean(X_normed, axis = 0)
standard_deviation = np.sqrt(np.sum((X-mean.transpose())**2, axis = 0)/X.shape[0])
X_normed = (X_normed - mean.transpose())/standard_deviation.transpose()

#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, label = 'Normalized')
#print(loss_values[:10])
#print(X_normed[:10])

#result = run_gd(loss, loss_gradient, X, y, theta_init, n_iter=10000, lr=1e-9)
#theta_est, loss_values, theta_values = result
#plt.plot(loss_values, label = "Without Normalized")
```

```
#print(loss_values[:10])
#print(X[:10])

plt.legend()
plt.show()
```



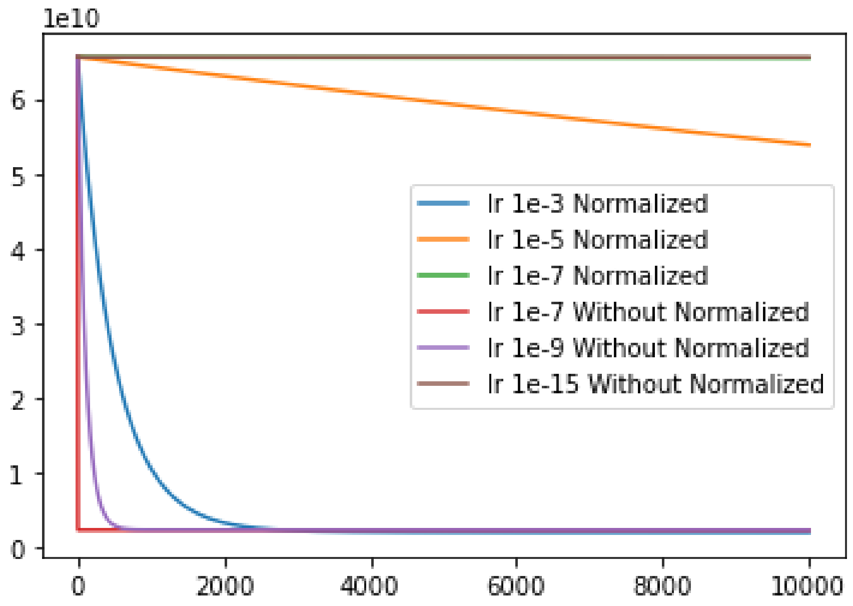
3.3 [10pt] How can we choose an appropriate learning rate? See what will happen if the learning rate is too small or too large for normalized and not normalized cases?

We try different learning rate until meet suitable one

In [496…

```
#raise NotImplementedError("Plot loss behaviour when with multiple different learning rates")
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, label = "lr 1e-3 Normalized")


result = run_gd(loss, loss_gradient, X_normed, y, theta_init, n_iter=10000, lr=1e-5)
theta_est, loss_values, theta_values = result
```

```
plt.plot(loss_values, label = 'lr 1e-5 Normalized')

result = run_gd(loss, loss_gradient, X_normed, y, theta_init, n_iter=10000, lr=1e-7)
theta_est, loss_values, theta_values = result

plt.plot(loss_values, label = 'lr 1e-7 Normalized')

result = run_gd(loss, loss_gradient, X, y, theta_init, n_iter=10000, lr=1e-7)
theta_est, loss_values, theta_values = result
plt.plot(loss_values, label = "lr 1e-7 Without Normalized")


result = run_gd(loss, loss_gradient, X, y, theta_init, n_iter=10000, lr=1e-9)
theta_est, loss_values, theta_values = result

plt.plot(loss_values, label = 'lr 1e-9 Without Normalized')

result = run_gd(loss, loss_gradient, X, y, theta_init, n_iter=10000, lr=1e-15)
theta_est, loss_values, theta_values = result

plt.plot(loss_values, label = 'lr 1e-15 Without Normalized')
plt.legend()
plt.show()
```



4. Written part

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.

We have:

$$\sum_i x^{(i)} \ln \mu + (1-x^{(i)}) \ln(1-\mu)$$

Take derivative of this with respect to μ we get

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

We set above = 0 to obtain the solution for μ, we get:

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

$$\sum x^{(i)} - \mu x^{(i)} = \sum \mu - \mu x^{(i)}$$

$$\sum_i x^{(i)} = \sum_i \mu = \mu + \mu + \mu + \dots + \mu = m\mu$$

Thus, $\mu = \frac{\sum_i x^{(i)}}{m} = \frac{h}{m}$ where h is $\sum$ all $x^{(i)}$

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 $w^{(i)}$.

$E(x_q)$ can be written as:

$$= \frac{1}{2}W^2(Y - X^T\theta)^2 = \frac{1}{2}(WY - WX^T\theta)^2$$

Substitute WY as Y and WX as X, apply the normal equations direct solutions, derivative with respect to θ we get:

$$\frac{(}{(}WX)^T WX)\theta - ((WX)^T WY) = 0$$

$$\theta = ((WX)^T WX)^{-1} + (WX)^T WY$$

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.

let head - head coin = HH, probability 1/3 tail - tail coin = TT, probability 1/3
head - tail coin = HT, probability 1/3
Having head on one side = H
The other side of the coin that is also H = AH
Thus we have

$$P(AH|H) = \frac{P(HandAH)}{P(H)} = \frac{\frac{1}{3}}{P(H|HH)*P(HH)+P(H|TT)*P(TT)+P(H|HT)*P(HT)}$$

$$= \frac{1/3}{1*1/3+0*1/3+1/2*1/3} = \frac{1/3}{1/2} = \frac{2}{3}$$

In [ ]: