Tyler Bryk
CPE-659 HW2

1.) Prove Baye's Theorem:

Consider two arbitrary events A and B where P(A) is the probability of event A happening and P(B) is the probability of event B happening.
P(A ∩ B) is the intersection of probability of both events happening.
This intersection can be written as:

$$P(A \cap B) = P(B) \cdot P(A|B)$$

or

$$P(A \cap B) = P(A) \cdot P(B|A)$$

therefore:

$$P(B) \cdot P(A|B) = P(A) \cdot P(B|A)$$

thus,

$$P(A|B) = \frac{P(A) \cdot P(B|A)}{P(B)}$$

[Baye's Theorem]

$$\text{Posterior} = \frac{\text{Prior} \times \text{Liklihood}}{\text{Evidence}} \longrightarrow P(\text{model} \mid \text{Data}) = \frac{P(\text{Data} \mid \text{Model}) * P(\text{Model})}{P(\text{Data})}$$

Baye's Theorem is useful in Machine Learning because it tells us how to gradually update our knowledge on something as more evidence becomes available. Rather than making one prediction on something, Baye's Theorem can be treated as a distribution where the knowledge (Model) gets better with each new piece of evidence (data). Prior probability represents an original belief, and Posterior probability takes new evidence into account.

2.) Prove Closed-Form Solution for Ridge Regression:

$W = (\lambda I + X^T \cdot X)^{-1} \cdot X^T \cdot y$ $\qquad E(w) = \sum_{i=1}^{m} (w^T \cdot x^i - y^i)^2 + \lambda \sum_{i=1}^{m} w_i^2$

Assuming we already know the regression solution without the regularization term:

$\beta = (X^T X)^{-1} \cdot X^T \cdot y$ then prove that adding the L2 term gives us W.

$L2 = \lambda \|\beta\|_2^2$

We will take the objective function $f(\beta) = (y - X\beta)^T \cdot (y - X\beta)$

We then add the L2 term as $\hookrightarrow (y - X\beta)^T \cdot (y - X\beta) + \lambda \beta^T \beta$

We then derive with respect to $\beta \rightarrow \frac{d}{d\beta}[\lambda \beta^T \beta] = 2\lambda\beta = \lambda I \beta$

Therefore, the ridge estimator is $[X^T X + \lambda I] \cdot \beta = X^T X$

Plugging this back into our original regression function gives:

$$W = (X^T X + \lambda I)^{-1} \cdot X^T \cdot y$$

Therefore, the closed-form solution for Ridge Regression is proven.

3.) Softmax Regression $\hat{p}_k = \delta(s_k(x))_k = \dfrac{e^{s_k(x)}}{\sum_{j=1}^{K} e^{s_j(x)}}$

Score $s_k(x) = \theta_k^T \cdot x$

A.) How many parameters need to be estimated?

The number of parameters that need to be estimated will be based off the number of Classes K. Since this is regression, each term will have a Cost Coefficient (Parameter) that determines the score for that class. Therefore, K classes will have K parameters estimated in the Softmax Regression Model, one parameter for each Class.

B.) Derive the gradient of $J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(\hat{p}_k^{(i)})$ regarding θ_k

$$\frac{\partial}{\partial\theta_k}\left[-\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \operatorname{Log}(\hat{p}_k^{(i)})\right] = \left[-\frac{1}{m}\sum_{i=1}^{m} y_k^{(i)} \frac{\partial}{\partial\theta}\operatorname{Log}(\hat{p}_k^{(i)})\right]$$

$$= \left[-\frac{1}{m}\sum_{i=1}^{m} y_k^{(i)} \frac{\partial}{\partial p_k}\operatorname{Log}(p_k)\frac{\partial p_k}{\partial\theta_k}\right] = \left[-\frac{1}{m}\sum_{i=1}^{m} y_k^{(i)}\frac{1}{\hat{p}_k^{(i)}}\frac{\partial p_k}{\partial\theta_k}\right]$$

Plugging in $\dfrac{\partial p_k}{\partial\theta_k} = \hat{p}_k^{(i)}\left(y_{(i)} + \sum y_k^{(i)}\right) - y_{(i)}$ which is the derivative of the Softmax Posterior Func.

Therefore,

$$\boxed{\frac{\partial}{\partial\theta_k}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{p}_k^{(i)} - y_k^{(i)}\right)x^{(i)}}$$

CPE695_HW2

October 3, 2020

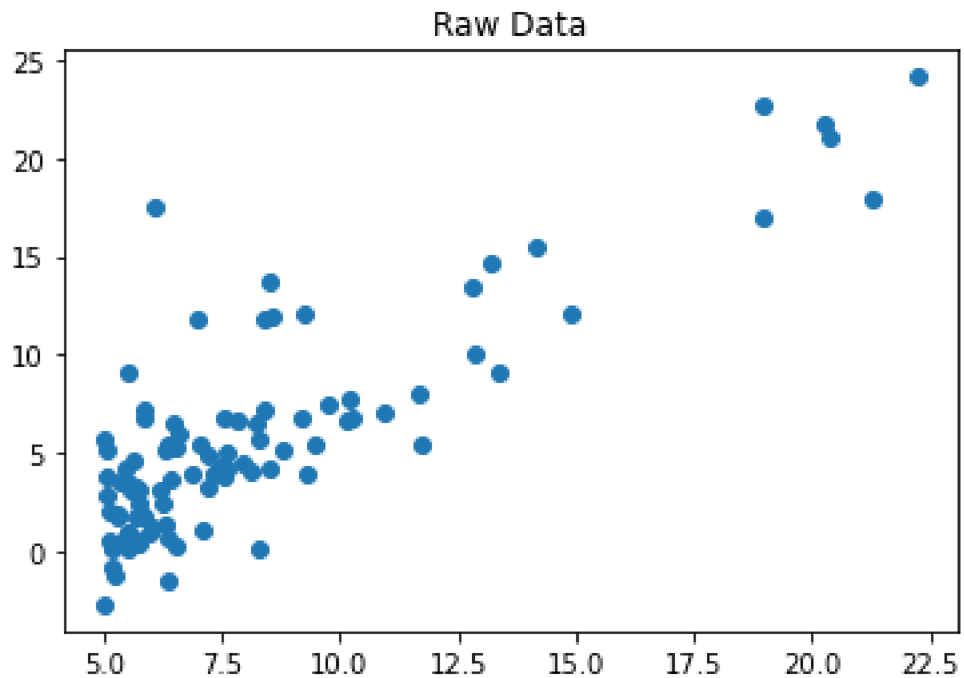
1 CPE695 HW2

By: Tyler Bryk

[13]:
```python
# Import Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from numpy.linalg import inv
```

[14]:
```python
# Input Raw Data and Plot
data = pd.read_csv('data2.txt', header = None)
x = np.expand_dims(data[0].to_numpy(), axis=1)
y = np.expand_dims(data[1].to_numpy(), axis=1)
xB = np.c_[np.ones((97, 1)), x] # Set Bias to 1

plt.scatter(x, y)
plt.title('Raw Data')
plt.show()
```



2 Normal Closed-Form Equation

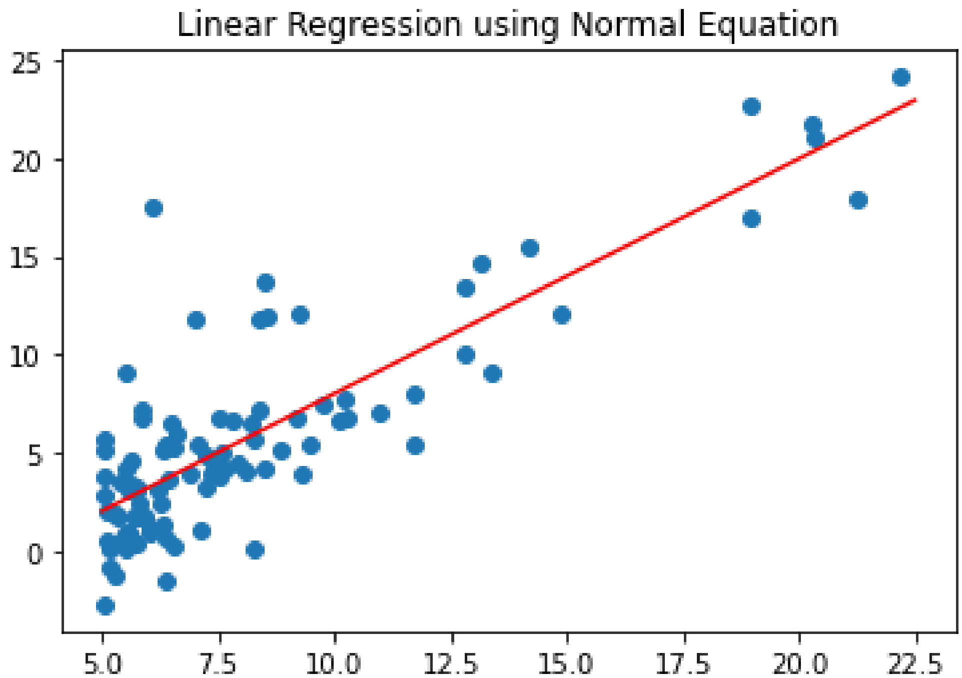
Here, the normal closed-form equation will be used to calculate the linear regression coefficients. The code and results for this method are shown below.

[15]:
```
# Implement Normal Equation where:    Coefficients = (X.T * X)^(-1) * X.T * Y
def normEquation(x, y):
  return inv((x.T).dot(x)).dot(x.T).dot(y)

coeffNorm = normEquation(xB,y)
print("Using Normal EQ the Coefficients are: ", coeffNorm[0], coeffNorm[1])
```

```
Using Normal EQ the Coefficients are:  [-3.89578088] [1.19303364]
```

[16]:
```
# Plot Regression Line for Normal Equation
xRange = np.linspace(5, 22.5)
yNorm = [coeffNorm[0] + coeffNorm[1]*w  for w in xRange]
plt.scatter(x, y)
plt.plot(xRange, yNorm, 'r-')
plt.title('Linear Regression using Normal Equation')
plt.show()
```



3 Batch Gradient Descent

Here, we will implement the batch gradient descent technique by attempting to minimize the cost function J using the entire data set. The code for the BGD method is shown below. The returned coefficients are similar to the ones returned from using the normal equation. Underneath the BGD code, a visualization is made which shows the optimal number of iterations, and the learning rate. As for the iteration number, the graph shows that the cost converges around 500 iterations, however, we selected 1000 iterations as our termination condition since it gave better accuracy. Equally, a few various learning rates were plotted, and the results show that a learning rate of alpha=0.02 was the best. Any learning rate of 0.03 or larger will not converge to a local minimum. Therefore, our parameters for this model were alpha=0.02 and iter=1000. The results for this method are shown below.

[17]:
```
# Implement Batch Gradient Descent where All Training Data is Used

iterations = 1000
alpha = 0.02                    # Learning Rate
theta = np.array([0, 0])        # Placeholder for Coefficients

# Cost J = (1/N) * SUM[ (xPred - Y)^2 ]
def costFunction(x, y, theta):
    return np.sum((x.dot(theta) - y)**2) / len(x)

def batchGD(x, y, theta, alpha, iterations):
```

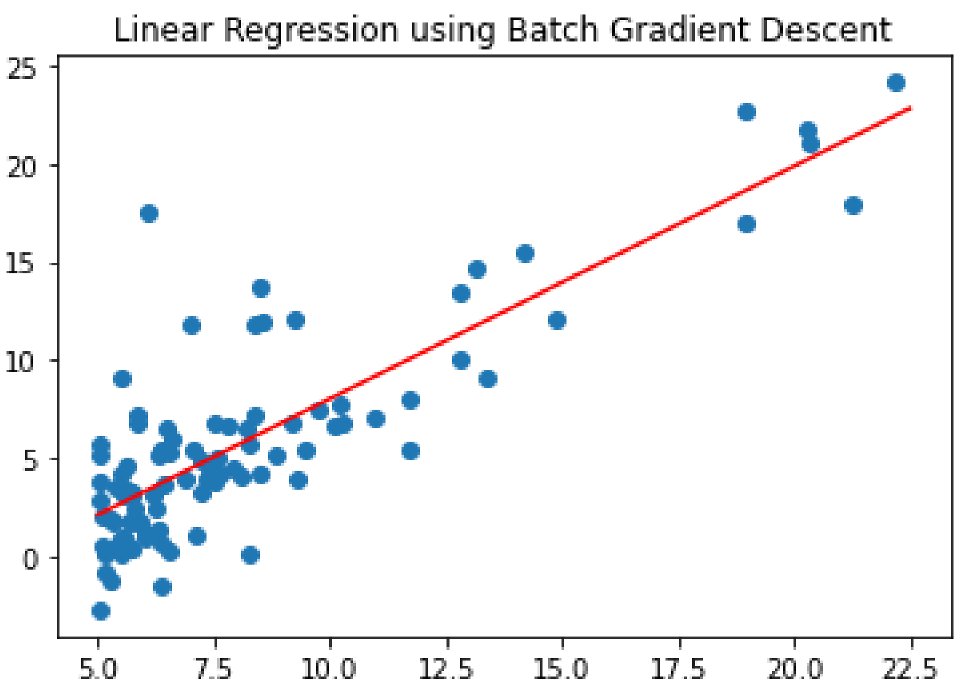
```
    costHistory = [0] * iterations
    for iteration in range(iterations):
        hypothesis = x.dot(theta)
        loss = hypothesis - y
        gradient = x.T.dot(loss) / len(x)
        theta = theta - alpha*gradient
        costHistory[iteration] = costFunction(x, y, theta)
    return theta, costHistory

coeffBatch, cost = batchGD(xB, y.flatten(), theta, alpha, iterations)
print("Using Batch Gradient Descent the Coefficients are: ", coeffBatch[0], ',␣
 ↪', coeffBatch[1])
```

```
Using Batch Gradient Descent the Coefficients are:  -3.7884192615511836 ,
1.1822480052540147
```

[18]:

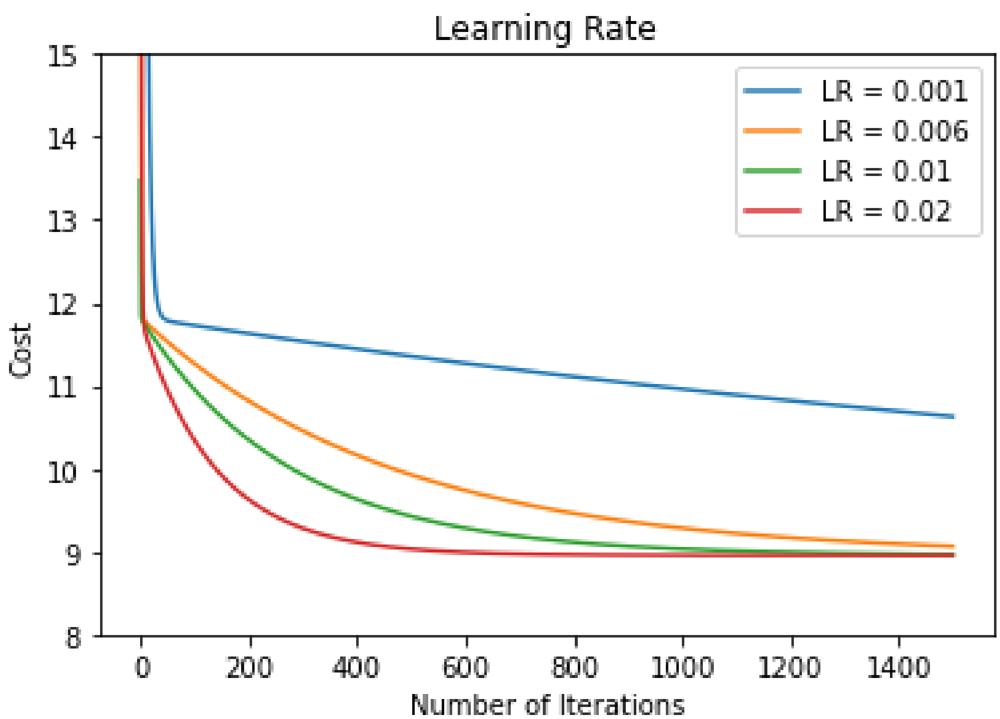
```
# Plot Regression Line for Batch Gradient Descent
xRange = np.linspace(5, 22.5)
yBatch = [coeffBatch[0] + coeffBatch[1]*w  for w in xRange]
plt.scatter(x, y)
plt.plot(xRange, yBatch, 'r-')
plt.title('Linear Regression using Batch Gradient Descent')
plt.show()
```

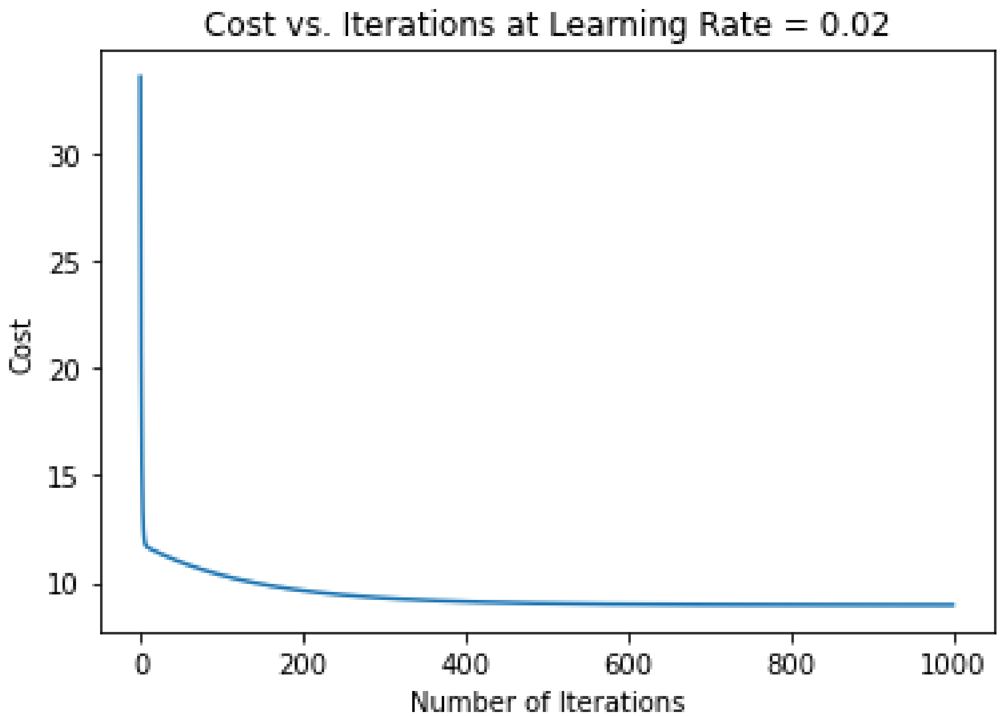


[19]:

```
# Find Optimal Learning Rate
plt.plot(range(1500), batchGD(xB, y.flatten(), theta, alpha=0.001,␣
 ↪iterations=1500)[1])
plt.plot(range(1500), batchGD(xB, y.flatten(), theta, alpha=0.006,␣
 ↪iterations=1500)[1])
plt.plot(range(1500), batchGD(xB, y.flatten(), theta, alpha=0.010,␣
 ↪iterations=1500)[1])
plt.plot(range(1500), batchGD(xB, y.flatten(), theta, alpha=0.020,␣
 ↪iterations=1500)[1])
plt.ylim((8,15))
plt.legend(['LR = 0.001', 'LR = 0.006', 'LR = 0.01', 'LR = 0.02'])
plt.title("Learning Rate")
plt.xlabel("Number of Iterations")
plt.ylabel("Cost")
plt.show()

# Find Optimal Number of Iterations
plt.plot(range(1000), batchGD(xB, y.flatten(), theta, alpha,␣
 ↪iterations=1000)[1])
plt.title("Cost vs. Iterations at Learning Rate = 0.02")
plt.xlabel("Number of Iterations")
plt.ylabel("Cost")
plt.show()
```





4 Stochastic Gradient Descent

Here, we will implement the stochastic gradient descent technique by attempting to minimize the cost function J using one sample at a time. The code for the SGD method is shown below. The returned coefficients are similar to the ones returned from using the normal equation and BGD methods. Underneath the SGD code, a visualization is made which shows the optimal number of iterations, and the learning rate. As for the iteration number, the graph shows that the cost converges around 300-400 iterations, therefore, 400 iterations was chosen as our termination condition. Equally, a few various learning rates were plotted, and the results show that a learning rate of alpha=0.1 was the best. Therefore, our parameters for this model were alpha=0.1 and iter=400. The results for this method are shown below.

[20]:
```
# Implement Stochastic Gradient Descent where One Training Sample is Used

iterations = 400
alpha = 0.1                    # Learning Rate
theta = np.array([0, 0])      # Placeholder for Coefficients

# Cost J = (1/N) * SUM[ (xPred - Y)^2 ]
def costFunction(x, y, theta):
    return np.sum((x.dot(theta) - y)**2) / len(x)

def SGD(x, y, theta, alpha, iterations):
```

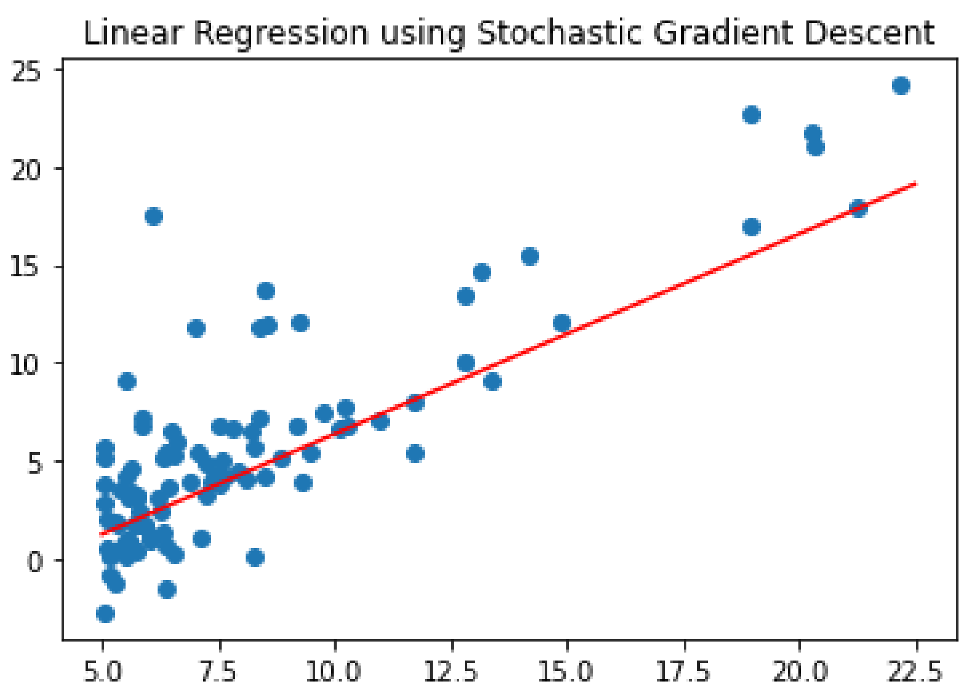
```
    costHistory = [0] * iterations
    for iteration in range(iterations):
      for row in range(len(x)):
        hypothesis = x[row].dot(theta)
        loss = hypothesis - y[row]
        gradient = x[row].T.dot(loss) / len(x)
        theta = theta - alpha*gradient
        costHistory[iteration] = costFunction(x[row], y[row], theta)
    return theta, costHistory

coeffSGD, cost = SGD(xB, y.flatten(), theta, alpha, iterations)
print("Using Stochastic Gradient Descent the Coefficients are: ", coeffSGD[0],␣
 ↪', ', coeffSGD[1])
```

```
Using Stochastic Gradient Descent the Coefficients are:  -3.8080810159300764 ,
1.018478723940661
```

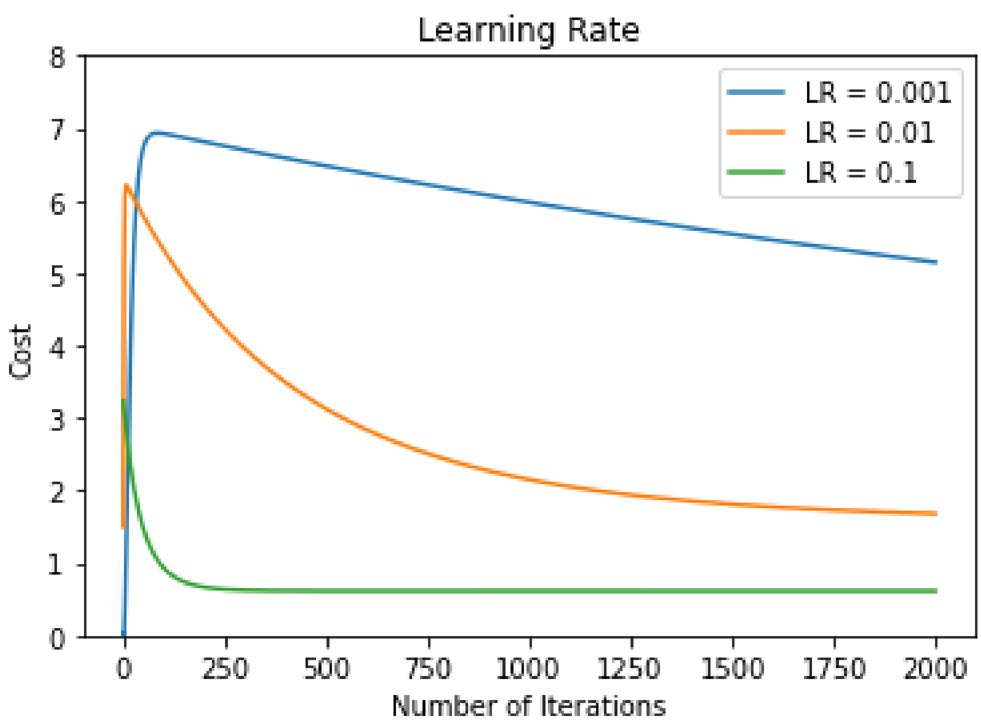
[21]:

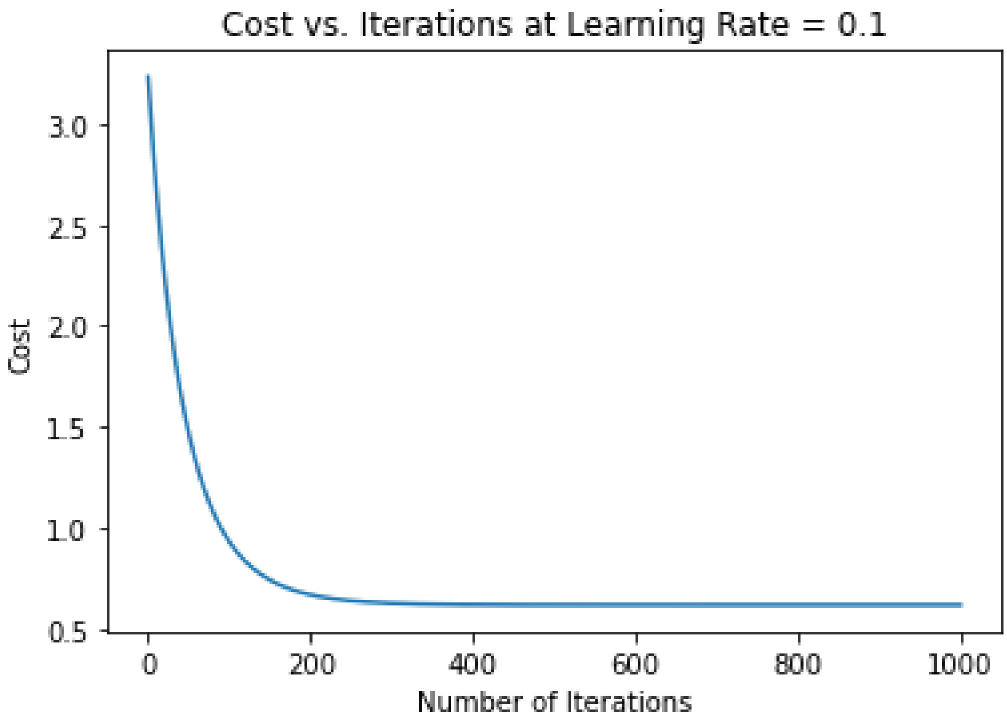
```
# Plot Regression Line for Stochastic Gradient Descent
xRange = np.linspace(5, 22.5)
yStoch = [coeffSGD[0] + coeffSGD[1]*w  for w in xRange]
plt.scatter(x, y)
plt.plot(xRange, yStoch, 'r-')
plt.title('Linear Regression using Stochastic Gradient Descent')
plt.show()
```



[22]:
```
# Find Optimal Learning Rate
plt.plot(range(2000), SGD(xB, y.flatten(), theta, alpha=0.001,␣
 ↪iterations=2000)[1])
plt.plot(range(2000), SGD(xB, y.flatten(), theta, alpha=0.010,␣
 ↪iterations=2000)[1])
plt.plot(range(2000), SGD(xB, y.flatten(), theta, alpha=0.100,␣
 ↪iterations=2000)[1])
plt.ylim((0,8))
plt.legend(['LR = 0.001', 'LR = 0.01', 'LR = 0.1'])
plt.title("Learning Rate")
plt.xlabel("Number of Iterations")
plt.ylabel("Cost")
plt.show()

# Find Optimal Number of Iterations
plt.plot(range(1000), SGD(xB, y.flatten(), theta, alpha, iterations=1000)[1])
plt.title("Cost vs. Iterations at Learning Rate = 0.1")
plt.xlabel("Number of Iterations")
plt.ylabel("Cost")
plt.show()
```





5 Discussion on Batch Gradient Descent vs. SGD

Overall, both the Batch Gradient Descent and Stochastic Gradient Descent methods yielded similar coefficients. It was noticed that the SGD method required less iterations to converge when compared to the BGD method. However, the BGD method became very unstable at higher learning rates, so it is difficult to make a fair comparison. In terms of accuracy, it appears that the SGD method was more accurate all around.