REPORT

Assignment 1:

Gradient-based learning with regularization constraints

MR.CHALERMKIAT CHANAHCHAN

ID: 62070701602

Assignment 1: Gradient-based learning with regularization constraints

Due: 19 September 2019. Please submit your report in PDF to MyLE

Introduction

One of the main problems in machine learning is to select suitable features for the model to learn. Embedded learning is one of feature selection approaches that aim to select suitable features and at the same time fit the model with data. In this assignment, you will work on deriving and implementing the stochastic gradient descent optimization on the logistic regression with ridge regularization. Logistic regression with ridge regularization can be expressed in the following formula,

$$f(x,w) = w_0 + \sum_{i=1}^{n} w_i\, x_i$$

$$H_w(x,w) = \frac{1}{1 + e^{-f(x,w)}}$$

where **x** is the vector of input feature and **w** is the vector of the regression coefficient. Error function for ridge regression can be expressed in the form of

$$\mathrm{E}(w) = \mathrm{L}(w) + \lambda \|w\|_2$$

$$= L(w) + \lambda \sum_{i=1}^{n} w_i^2$$

L(**w**) is the loss/error function of the model and λ is a regularization parameter. Ridge regularization will penalize the use of weight and the outcome of the optimization will be a fitted model with minimal weight. For demonstration, we will use a very simple iris dataset. The report needs to address the following problems/tasks.

Iris dataset: https://www.kaggle.com/arshid/iris-flower-dataset

Raw Dataset: Iris flowers Dataset 150 records

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
...	...	...	...	...	...
50	7	3.2	4.7	1.4	Iris-versicolor
51	6.4	3.2	4.5	1.5	Iris-versicolor
52	6.9	3.1	4.9	1.5	Iris-versicolor
53	5.5	2.3	4	1.3	Iris-versicolor
...	...	...	...	...	...
100	6.3	3.3	6	2.5	Iris-virginica
101	5.8	2.7	5.1	1.9	Iris-virginica
102	7.1	3	5.9	2.1	Iris-virginica
103	6.3	2.9	5.6	1.8	Iris-virginica
...	...	...	...	...	...
149	5.9	3	5.1	1.8	Iris-virginica

Tasks

1. Prepare the data in one-against-the-rest strategy. This can be done by converting the "Species" column into 3 binary columns.
2. Formulate the error function of the logistic regression with ridge regularization criterion.
3. Derive the gradient of the error function by deriving the partial derivative of the error function in Task 2.
4. Implement the gradient descent using all of the dataset in each iteration.
5. Implement the stochastic gradient descent using the subset of dataset in each iteration.
6. Test to see the effect of λ.
7. Test to see the effect of sampling proportion in Task 5.

Task 1: Prepare the data in one-against-the-rest strategy. This can be done by converting the "Species" column into 3 binary columns.

Input vector

X ∈ {sepal_length, sepal_width, petal_length, petal_width}

Output Vector

Y ∈ {Iris-setosa, Iris-versicolor, Iris-virginica}

Adding 3 Binary Columns

	sepal_length	sepal_width	petal_length	petal_width	species	Species_Iris-setosa	Species_Iris-versicolor	Species_Iris-vriginica
0	5.1	3.5	1.4	0.2	Iris-setosa	1	0	0
1	4.9	3	1.4	0.2	Iris-setosa	1	0	0
2	4.7	3.2	1.3	0.2	Iris-setosa	1	0	0
3	4.6	3.1	1.5	0.2	Iris-setosa	1	0	0
…	…	…	…	…	…	…	…	…
50	7	3.2	4.7	1.4	Iris-versicolor	0	1	0
51	6.4	3.2	4.5	1.5	Iris-versicolor	0	1	0
52	6.9	3.1	4.9	1.5	Iris-versicolor	0	1	0
53	5.5	2.3	4	1.3	Iris-versicolor	0	1	0
…	…	…	…	…	…	…	…	…
100	6.3	3.3	6	2.5	Iris-virginica	0	0	1
101	5.8	2.7	5.1	1.9	Iris-virginica	0	0	1
102	7.1	3	5.9	2.1	Iris-virginica	0	0	1
103	6.3	2.9	5.6	1.8	Iris-virginica	0	0	1
…	…	…	…	…	…	…	…	…
149	5.9	3	5.1	1.8	Iris-virginica	0	0	1

Code:

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

data = pd.read_csv('IRIS.csv')

# Task 1 Create 3 Binary Columns
df = pd.get_dummies(data['species'])

binary_data = pd.concat([data,df],axis=1)
print (binary_data)

# Drop Iris-versicolor
binary_data.drop(['species','Iris-versicolor','Iris-virginica'],axis=1,inplace=True)

input_data = binary_data.drop(binary_data.columns[[4]], axis=1)
output_data = binary_data.drop(binary_data.columns[[0,1,2,3]], axis=1)
```

Input vector

X ∈ {sepal_length, sepal_width, petal_length, petal_width}

Output Vector

Y ∈ { 1 (Iris-setosa), 2 (No Iris-setosa)}

	Input Data (X)				Output Data (Y)
	sepal_length	sepal_width	petal_length	petal_width	Species_Iris-setosa
0	5.1	3.5	1.4	0.2	1
1	4.9	3	1.4	0.2	1
2	4.7	3.2	1.3	0.2	1
3	4.6	3.1	1.5	0.2	1
…	…	…	…	…	…
50	7	3.2	4.7	1.4	0
51	6.4	3.2	4.5	1.5	0

52	6.9	3.1	4.9	1.5	0
53	5.5	2.3	4	1.3	0
...	...	...	...	...	...
100	6.3	3.3	6	2.5	0
101	5.8	2.7	5.1	1.9	0
102	7.1	3	5.9	2.1	0
103	6.3	2.9	5.6	1.8	0
...	...	...	...	...	...
149	5.9	3	5.1	1.8	0

Task 2: Formulate the error function of the logistic regression with ridge regularization criterion.

$$f(x, w) = w_0 + \sum_{i=1}^{n} w_i\, x_i$$

$$H_w(x, w) \; = \frac{1}{1 + e^{-f(x,w)}}$$

x is the vector of input feature

w is the vector of the regression coefficient

Formulate the error function:

Implement from Linear Regression Model

$$\begin{aligned} f(x, w) \qquad &= w_0 + \sum_{i=1}^{n} w_i\, x_i \\ &= w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 \\ &= \theta^T x \end{aligned}$$

Then apply the sigmoid function to the $f(x, w)$ then we get Logistic regression model, n is range of input features (n = 4)

$$\begin{aligned} H_w(x, w) \qquad &= \frac{1}{1 + e^{-f(x,w)}} \\ &= \frac{1}{1 + e^{-(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4)}} \\ &= \sigma(\theta^T x) \end{aligned}$$

Find the loss function ($L(w)$) of the Logistic regression model, first we use derivative of $H_w(x, w)$. The answer of it is cost function.

$$Cost\big(h_w(x_i)\big) = \frac{\partial H_w(x_i, w)}{\partial x} = \; -H_w(x_i)(1 - H_w(x_i))x_i$$

Determined when y ∈ {0,1}

$$Cost(h_w(x), \mathrm{y}) = \begin{cases} -log(h_w(x)) \text{ if y } = 1 \\ -log(1 - h_w(x)) \text{ if y } = 0 \end{cases}$$

if y = 1 and

$h_w(x_i) = 1$, then $Cost(h_w(x), \mathrm{y})$ = 0

$h_w(x_i) \to 0$, then $Cost(h_w(x), \mathrm{y}) \longrightarrow \infty$

if y = 0 and

$h_w(x_i) = 0$, then $Cost(h_w(x), \mathrm{y})$ = 0

$h_w(x_i) \to 1$, then $Cost(h_w(x), \mathrm{y}) \longrightarrow \infty$

Then can be written as,

$$\mathrm{Cost}(h_w(x), \mathrm{y}) = -\mathrm{y}\log(h_w(x) - (1-\mathrm{y})\log(1 - h_w(x))$$

So, we can formulate the loss function from $Cost(h_w(x), \mathrm{y})$ as follows,

$$L(w) = \frac{1}{m}\sum_{i=1}^{m} \mathrm{Cost}(h_w(x^{(i)}), y^{(i)})$$

$$L(w) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log(h_w(x^{(i)})) + (1-y^{(i)})\log(1 - h_w(x^{(i)}))$$

Then we put it into error function of the logistic regression with ridge regularization

$$E(w) = L(w) + \lambda \sum_{j=1}^{n} w_j^2$$

$$E(w) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log(h_w(x^{(i)})) + (1-y^{(i)})\log(1 - h_w(x^{(i)})) + \lambda \sum_{j=1}^{n} w_j^2$$

$$\mathrm{E}(w) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log(h_w(x^{(i)})) + (1-y^{(i)})\log(1 - h_w(x^{(i)})) \quad + \lambda(w_1^2 + w_2^2 + w_3^2 + w_4^2)$$

The error function of the logistic regression with ridge regularization:

$$\mathrm{E}(w) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log(h_w(x^{(i)})) \; + \; (1 - y^{(i)}) \log(1 \; - h_w(x^{(i)})) \\ + \lambda(w_1^2 + w_2^2 + w_3^2 + w_4^2)$$

Task 3: Derive the gradient of the error function by deriving the partial derivative of the error function in Task 2.

$$\frac{\partial}{\partial w}E(w) = \frac{\partial}{\partial w}\left[-\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log(h_w(x^{(i)})) + (1-y^{(i)})\log(1-h_w(x^{(i)})) + \lambda\sum_{j=1}^{n} w_j^2\right]$$

$$\frac{\partial}{\partial w}E(w) = -\frac{1}{m}\sum_{j=1}^{n}\frac{\partial}{\partial w}\left[y^{(i)}\log(h_w(x^{(i)})) + (1-y^{(i)})\log(1-h_w(x^{(i)}))\right] + \lambda\sum_{j=1}^{n}\frac{\partial}{\partial w}w_j^2$$

$$\frac{\partial}{\partial w}E(w) = -\frac{1}{m}\sum_{j=1}^{n}\left[y^{(i)}\frac{\partial}{\partial w}\log(h_w(x^{(i)})) + (1-y^{(i)})\frac{\partial}{\partial w}\log(1-h_w(x^{(i)}))\right] + \lambda\sum_{j=1}^{n}\frac{\partial}{\partial w}w_j^2$$

From differential of log is, $\frac{\partial}{\partial z}log(z) = \frac{1}{z}$

And differential of sigmoid function is,

$$\frac{\partial}{\partial z}H(z) = \frac{\partial}{\partial w}\left(\frac{1}{1+e^{-z}}\right)$$

$$\frac{\partial}{\partial z}H(z) = -1(1+e^{-z})^{-2}\frac{\partial}{\partial z}(1+e^{-z})$$

$$\frac{\partial}{\partial z}H(z) = -1(1+e^{-z})^{-2}(\frac{\partial}{\partial z}(1) + \frac{\partial}{\partial z}(e^{-z}))$$

$$\frac{\partial}{\partial z}H(z) = -1(1+e^{-z})^{-2}(0 + (-e^{-z}))$$

$$\frac{\partial}{\partial z}H(z) = -\frac{1}{(1+e^{-z})^2}(-e^{-z})$$

$$\frac{\partial}{\partial z}H(z) = -\left(\frac{1}{1+e^{-z}}\right)\left(\frac{-e^{-z}}{1+e^{-z}}\right)$$

$$\frac{\partial}{\partial z}H(z) = \left(\frac{1}{1+e^{-z}}\right)\left(\frac{e^{-z}}{1+e^{-z}}\right)$$

Adding 1 minus 1 (1 – 1 = 0, the value of formula does not change) in formula

$$\frac{\partial}{\partial z}H(z) = \left(\frac{1}{1+e^{-z}}\right)\left(\frac{e^{-z}+(1-1)}{1+e^{-z}}\right)$$

$$\frac{\partial}{\partial z}H(z) = H(z)\left(\frac{e^{-z}+1-1}{1+e^{-z}}\right)$$

$$\frac{\partial}{\partial z}H(z) = H(z)\left(\frac{e^{-z}+1}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right)$$

$$\frac{\partial}{\partial z}H(z) = H(z)\big(1 - H(z)\big)$$

Using **differential of log** and **differential of sigmoid function** in the partial derivative of the error function

$$\frac{\partial}{\partial w}E(w) = -\frac{1}{m}\sum_{j=1}^{n}\left[y^{(i)}\frac{1}{h_w(x^{(i)})}\frac{\partial}{\partial w}(h_w(x^{(i)})) \ + \ (1-y^{(i)})\ \frac{1}{(1\ - h_w(x^{(i)}))}\frac{\partial}{\partial w}(1\ - h_w(x^{(i)}))\right] + 2\lambda\sum_{j=1}^{n} w_j$$

$$\frac{\partial}{\partial w}E(w) = -\frac{1}{m}\sum_{j=1}^{n}\left[y^{(i)}\frac{1}{h_w(x^{(i)})}h_w(x^{(i)})(1-\ h_w(x^{(i)}))\frac{\partial}{\partial w}(x^{(i)}) \ + \ (1 - y^{(i)})\ \frac{1}{(1\ - h_w(x^{(i)}))}\left(\frac{\partial}{\partial w}(1)\ - \frac{\partial}{\partial w}h_w(x^{(i)}))\right)\right] + 2\lambda\sum_{j=1}^{n} w_j$$

$$\frac{\partial}{\partial w}E(w) = -\frac{1}{m}\sum_{j=1}^{n}\left[y^{(i)}\frac{1}{h_w(x^{(i)})}h_w(x^{(i)})(1-\ h_w(x^{(i)}))(x^{(i)}) \ + \ (1 - y^{(i)})\ \frac{1}{(1\ - h_w(x^{(i)}))}\left(0 - (h_w(x^{(i)}))(1-\ h_w(x^{(i)}))\frac{\partial}{\partial w}(x^{(i)})\ \right)\right] + 2\lambda\sum_{j=1}^{n} w_j$$

$$\frac{\partial}{\partial w}E(w) = -\frac{1}{m}\sum_{j=1}^{n}\left[y^{(i)}\frac{1}{\cancel{h_w(x^{(i)})}}\cancel{h_w(x^{(i)})}(1-\ h_w(x^{(i)}))(x^{(i)}) \ + \ (1 - y^{(i)})\ \frac{1}{\cancel{(1\ - h_w(x^{(i)}))}}\left(-(h_w(x^{(i)}))\cancel{(1-\ h_w(x^{(i)}))}(x^{(i)})\right)\right] + 2\lambda\sum_{j=1}^{n} w_j$$

$$\frac{\partial}{\partial w}E(w) = -\frac{1}{m}\sum_{j=1}^{n}\left[y^{(i)}(1-\ h_w(x^{(i)}))(x^{(i)}) \ + \ (1-y^{(i)})\left(\ -(h_w(x^{(i)}))(x^{(i)})\right)\right] + 2\lambda\sum_{j=1}^{n} w_j$$

$$\frac{\partial}{\partial w}E(w) = -\frac{1}{m}\sum_{j=1}^{n}\left[\left(y^{(i)}(1-\ h_w(x^{(i)})) \ + \ (1-y^{(i)})(-h_w(x^{(i)}))\right)(x^{(i)})\ \right] + 2\lambda\sum_{j=1}^{n} w_j$$

$$\frac{\partial}{\partial w}E(w) = -\frac{1}{m}\sum_{j=1}^{n}\left[\left(y^{(i)} - \cancel{y^{(i)}h_w(x^{(i)})} \ + \ (-h_w(x^{(i)})) + \cancel{y^{(i)}h_w(x^{(i)})}\right)(x^{(i)})\ \right] + 2\lambda\sum_{j=1}^{n} w_j$$

$$\frac{\partial}{\partial w}E(w) = -\frac{1}{m}\sum_{j=1}^{n}\left[\left(y^{(i)} + \ (-h_w(x^{(i)}))\right)(x^{(i)})\ \right] + 2\lambda\sum_{j=1}^{n} w_j$$

$$\frac{\partial}{\partial w}E(w) = \frac{1}{m}\sum_{j=1}^{n}\left[h_w(x^{(i)}) - y^{(i)}\right](x^{(i)}) + 2\lambda\sum_{j=1}^{n} w_j$$

The partial derivative of the error function in Task 2:

$$\frac{\partial}{\partial w}E(w) = \frac{1}{m}\sum_{j=1}^{n}\left[h_w(x^{(i)}) - y^{(i)}\right](x^{(i)}) + 2\lambda\sum_{j=1}^{n} w_j$$

Task 4: Implement the gradient descent using all of the dataset in each iteration.

Code:

```python
def weightInitialization(n_features):
    w = np.zeros((1,n_features))
    w_0 = 0
    return w_0, w

def sigmoid_func(z):
    return 1 / (1 + np.exp(-z))

def model_optimize(w_0, w, X, Y,lambda_value):
    m = X.shape[0]
    #Prediction
    final_result = sigmoid_func(np.dot(w,X.T)+w_0)
    Y_T = Y.T
    # Normal Cost Function
    cost = (-1/m)*(np.sum((Y_T*np.log(final_result)) + ((1-Y_T)*(np.log(1-final_result)))))
    regularize = (lambda_value/(2*m)) * (np.dot(w.T,w)[0][0])
    cost = cost + regularize
    cost = cost + regularize

    #Gradient calculation
    dw = (1/m)*(np.dot(X.T, (final_result-Y.T).T))
    dw_0 = (1/m)*(np.sum(final_result-Y.T))

    #Adding Reguralize Cost Function to Gardient
    dw = dw + (w*(lambda_value/m)).T
    grads = {"dw": dw, "dw_0": dw_0}
    return grads, cost, regularize
def model_predict(w_0, w, X, Y,lambda_value, learning_rate, no_iterations):
    costs = []
    reg_costs = []
    for i in range(no_iterations):
        #
        grads, cost, reg = model_optimize(w_0,w,X,Y,lambda_value)
        #
        dw = grads["dw"]
        dw_0 = grads["dw_0"]
        #weight update
        w = w - learning_rate * (dw.T)
        w_0 = w_0 - (learning_rate * dw_0)

        if (i % 100 == 0):
            costs.append(cost)
            reg_costs.append(reg)
    #final parameters
    coeff = {"w": w, "w_0": w_0}
    gradient = {"dw": dw, "dw": dw_0}

    return coeff, gradient, costs, reg_costs

#Logistic Regession Model
input_data = binary_data.drop(binary_data.columns[[4]], axis=1)
output_data = binary_data.drop(binary_data.columns[[0,1,2,3]], axis=1)

#Get number of features
number_features = input_data.shape[1]
print('Number of Features', number_features)
w_0, w = weightInitialization(number_features)
print ("Weight 0 : ",w_0)
print ("Weight N : ",w)
input_arr = input_data.to_numpy()
output_arr = output_data.to_numpy()

#Set Lambda
lambda_value = 1
#Gradient Descent
coeff, gradient, costs,reg_costs = model_predict(w_0,w, input_arr, output_arr,lambda_value, learning_r
ate=0.01,no_iterations=10)
#Final prediction
w = coeff["w"]
w_0 = coeff["w_0"]
print('Optimized weights', w)
print('Optimized intercept',w_0)

#Plot costs function each iteration
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title('Cost reduction over time')
plt.show()
```

Initial weight:

weights = [0, 0, 0, 0]

intercept = 0

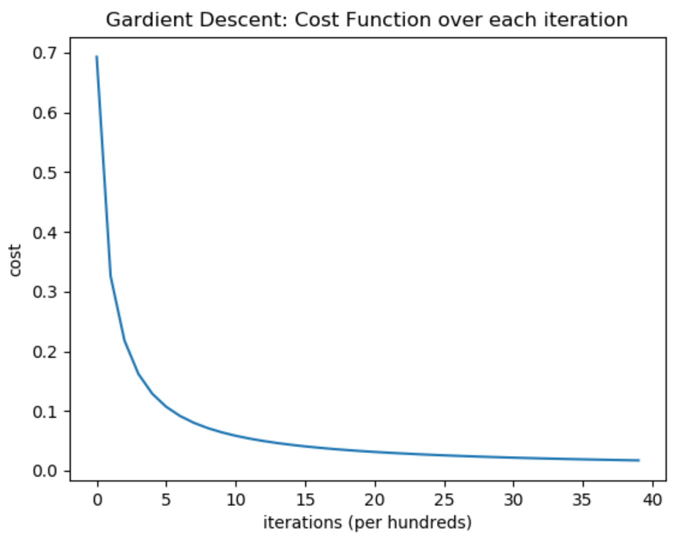
Learning Rate = 0.01

$\lambda = 0$

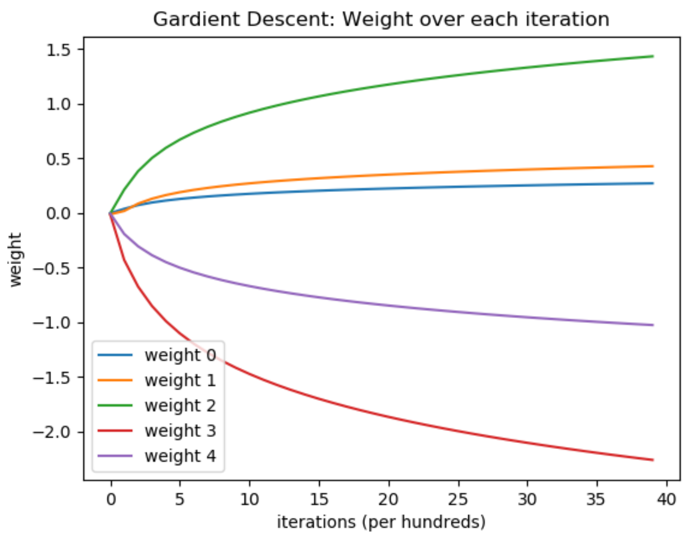
Optimized weight:

Optimized weights = [0.43010461, 1.44132531, -2.27212597, -1.03131427]

Optimized intercept = 0.2726775534196986



Cost Function over each iteration, Learning Rate = 0.01, λ = 1



Weight Function over each iteration, Learning Rate = 0.01, λ = 1

Task 5: Implement the stochastic gradient descent using the subset of dataset in each iteration.

Pseudocode for stochastic gradient descent (SGD):

```
- Choose an initial vector of solution
- Repeat until an approximated minimum is reached: {
    - Randomly sample data (from a larger pool) into the learning dataset
    - Calculate stochastic gradient
          $\theta = \theta - \alpha \frac{\partial L(w)}{\partial x}$
}
```

Code:

```python
def model_predict_SGD(w_0, w, X, Y,lambda_value,learning_rate, no_iterations):
    costs = []
    reg_costs = []
    m = X.shape[0]
    for i in range(no_iterations):
        cost_temp = 0
        for j in range(m):
            #Random Sampling
            rand_index = np.random.randint(0,m)
            X_rand = X[rand_index,:].reshape(1,X.shape[1])
            # print("X_rand: {0}".format(X_rand))
            Y_rand = Y[rand_index].reshape(1,1)
            # print("Y_rand: {0}".format(Y_rand))
            grads, cost, reg = model_optimize(w_0,w,X_rand,Y_rand,lambda_value)

            dw = grads["dw"]
            dw_0 = grads["dw_0"]
            #weight update
            w = w - learning_rate * (dw.T)
            w_0 = w_0 - (learning_rate * dw_0)
            cost_temp = cost

        if (i % 100 == 0):
            costs.append(cost_temp)
            reg_costs.append(reg)
            #print("Cost after %i iteration is %f" %(i, cost))

    #final parameters
    coeff = {"w": w, "w_0": w_0}
    gradient = {"dw": dw, "dw": dw_0}

    return coeff, gradient, costs, reg_costs
```

```
#Logistic Regession Model
input_data = binary_data.drop(binary_data.columns[[4]], axis=1)
output_data = binary_data.drop(binary_data.columns[[0,1,2,3]], axis=1)

#Get number of features
number_features = input_data.shape[1]
print('Number of Features', number_features)
w_0, w = weightInitialization(number_features)
print ("Weight 0 : ",w_0)
print ("Weight N : ",w)
input_arr = input_data.to_numpy()
output_arr = output_data.to_numpy()

#Set Lambda
lambda_value = 1

#Stochastic Gradient Descent
coeff, gradient, costs,reg_costs = model_predict_SGD(w_0,w, input_arr, output_arr,lambda_value, learni
ng_rate=0.01,no_iterations=4000)
#Final prediction
w = coeff["w"]
w_0 = coeff["w_0"]
print('Optimized weights', w)
print('Optimized intercept',w_0)
# # #
final_hypothesis = sigmoid_func(np.dot(w,input_arr.T)+w_0)
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title('Stochastic Gradient Descent: Cost Function over each iteration')
plt.show()
```

Initial weight:

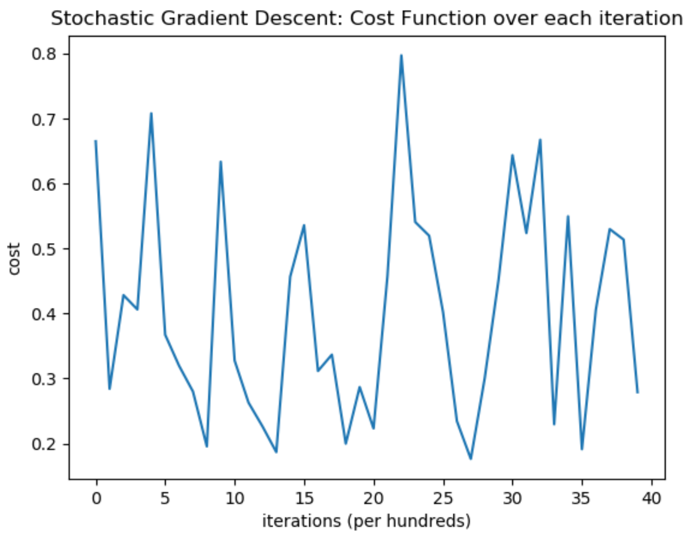
weights = [0, 0, 0, 0]

intercept = 0

Learning Rate = 0.01

$\lambda = 0$

Output:



Cost Function over each iteration, Learning Rate = 0.01, λ = 1

Task 6: Test to see the effect of λ.

Testing on λ = 1:

Initial weight:

weights = [0, 0, 0, 0]

intercept = 0

Learning Rate = 0.01

λ = 1

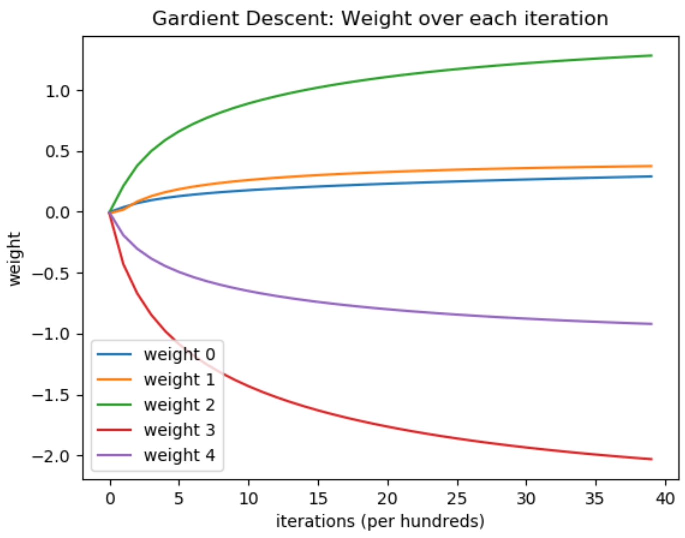
Optimized weight:

Optimized weights = [0.37679841, 1.28811695, -2.03875309, -0.92364375]

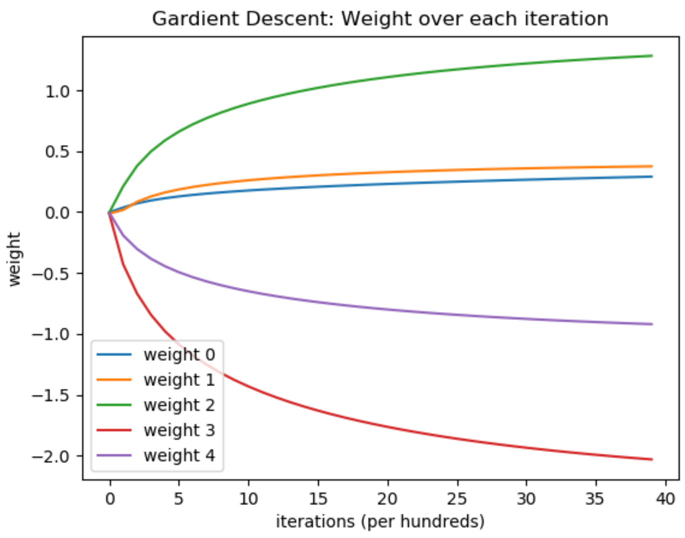
Optimized intercept = 0.2726775534196986

Regularization Cost each 100 Iteration =

```
[0.0,                   0.0008734739060217522, 0.00229017134031385,
0.0037279904077559926, 0.005068320651172439,  0.006295759895919654,
0.007418103902982904,  0.008447454326962844,  0.00939547011697645,
0.01027229446871948,   0.01108649239537152,   0.011845248074871091,
0.012554600603475076,  0.013219655776098092,  0.013844761044916136,
0.014433645764251265,  0.014989532525079355,  0.015515225488559945,
0.016013180736898114,  0.01648556263448669,   0.016934289293004882,
0.017361069513475033,  0.017767433022003506,  0.018154755393317883,
0.018524278736883254,  0.018877128979071164,  0.019214330391929085,
0.01953681787975601,   0.019845447427928202,  0.020141005036104366,
0.0204242143940589,    0.02069574350849213,   0.020956210449935068,
0.02120618835782552,   0.021446209817121968,  0.021676770700032526,
0.02189833355049836,   0.022111330576161516,  0.022316166302038202,
0.022513219931518384]
```



Cost Function over each iteration, Learning Rate = 0.01, λ = 1



Weight Function over each iteration, Learning Rate = 0.01, λ = 1

Testing on λ = 0.1:

Initial weight:

weights = [0, 0, 0, 0]

intercept = 0

Learning Rate = 0.01

λ = 0.1

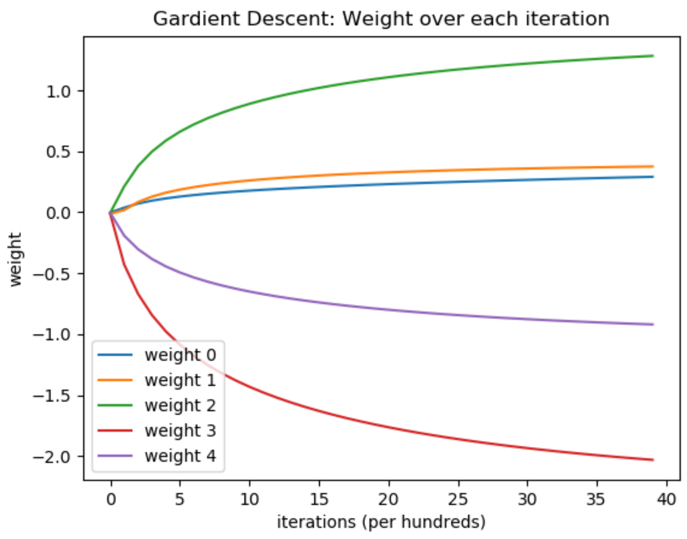
Optimized weight:

Optimized weights = [0.37679841, 1.28811695, -2.03875309, -0.92364375]

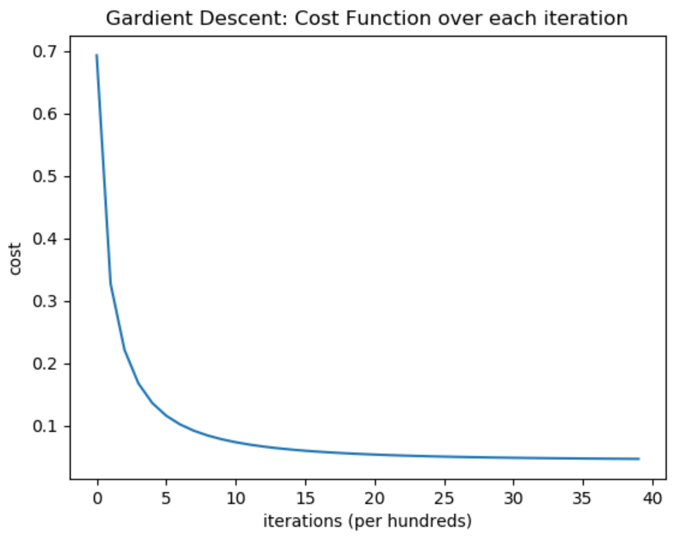
Optimized intercept = 0.29343805723514027

Regularization Cost each 100 Iteration =

```
[0.0,                   0.0008734739060217522, 0.00229017134031385,
0.0037279904077559926, 0.005068320651172439,  0.006295759895919654,
0.007418103902982904,  0.008447454326962844,  0.00939547011697645,
0.01027229446871948,   0.01108649239537152,   0.011845248074871091,
0.012554600603475076,  0.013219655776098092,  0.013844761044916136,
0.014433645764251265,  0.014989532525079355,  0.015515225488559945,
0.016013180736898114,  0.01648556263448669,   0.016934289293004882,
0.017361069513475033,  0.017767433022003506,  0.018154755393317883,
0.018524278736883254,  0.018877128979071164,  0.019214330391929085,
0.01953681787975601,   0.019845447427928202,  0.020141005036104366,
0.0204242143940589,    0.02069574350849213,   0.020956210449935068,
0.02120618835782552,   0.021446209817121968,  0.021676770700032526,
0.02189833355049836,   0.022111330576161516,  0.022316166302038202,
0.022513219931518384]
```



Cost Function over each iteration, Learning Rate = 0.01, λ = 0.1



Weight Function over each iteration, Learning Rate = 0.01, λ = 0.1

Testing on λ = 10:

Initial weight:

weights = [0, 0, 0, 0]

intercept = 0

Learning Rate = 0.01

λ = 10

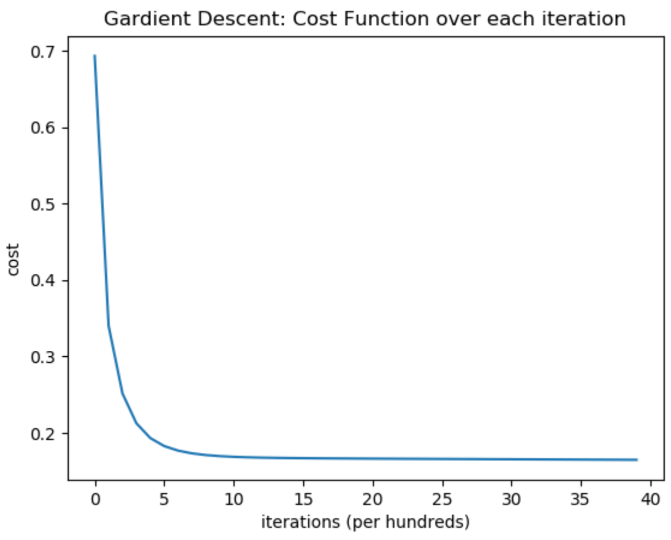
Optimized weight:

Optimized weights = [0.17662097, 0.74468777, -1.24544124, -0.55731151]

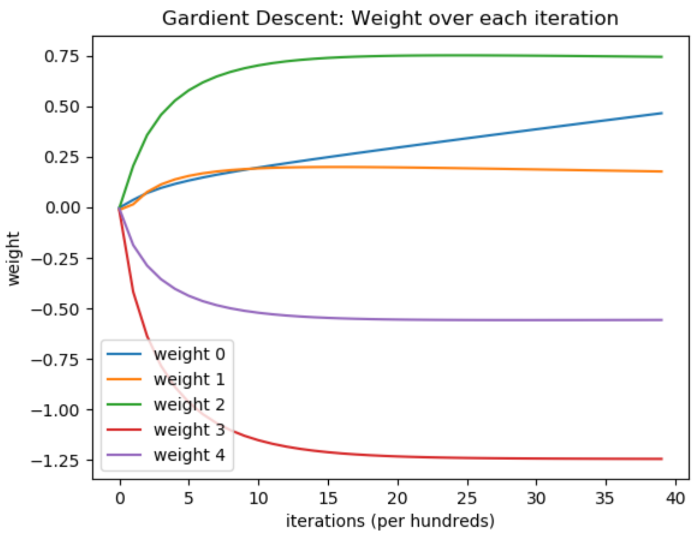
Optimized intercept = 0.47478466664893965

Regularization Cost each 100 Iteration =

```
[0.0,                   0.008286982077434826, 0.020653778417247543,
0.03205033067854311,    0.04160389047321929,  0.049410341629835125,
0.05573927019619207,    0.06085868969386058,  0.06499815294019765,
0.06834585494562426,    0.07105386230742455,  0.07324453044164099,
0.0750162811065065,     0.07644838129417979,  0.07760478103009945,
0.07853716495622279,    0.07928737406556495,  0.07988933033451584,
0.08037057050607398,    0.08075347222842491,  0.08105623726879027,
0.08129368215342646,    0.08147787553631866,  0.08161865311758486,
0.0817240344023929,     0.08180056053793126,  0.08185356853256638,
0.08188741408119551,    0.08190565279501488,  0.08191118771341349,
0.08190638944805755,    0.08189319408908788,  0.0818731830253183,
0.08184764804397163,    0.0818176444416125,   0.08178403436584256,
0.0817475221928624,     0.08170868341004084,  0.0816679881999492,
0.08162582070075812]
```



Cost Function over each iteration, Learning Rate = 0.01, λ = 10



Weight Function over each iteration, Learning Rate = 0.01, λ = 10

Task 7: Test to see the effect of sampling proportion in Task 5.

- **Adding 2 Random Sample**

Testing on λ = 1:

Initial weight:

weights = [0, 0, 0, 0]

intercept = 0

Learning Rate = 0.01

λ = 10

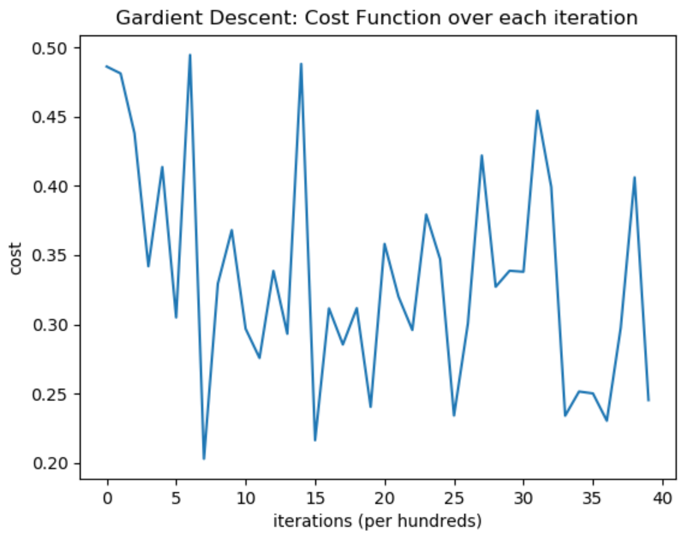
Optimized weight:

Optimized weights = [-0.21887377, 0.11067022, -0.61494334, -0.25176759]

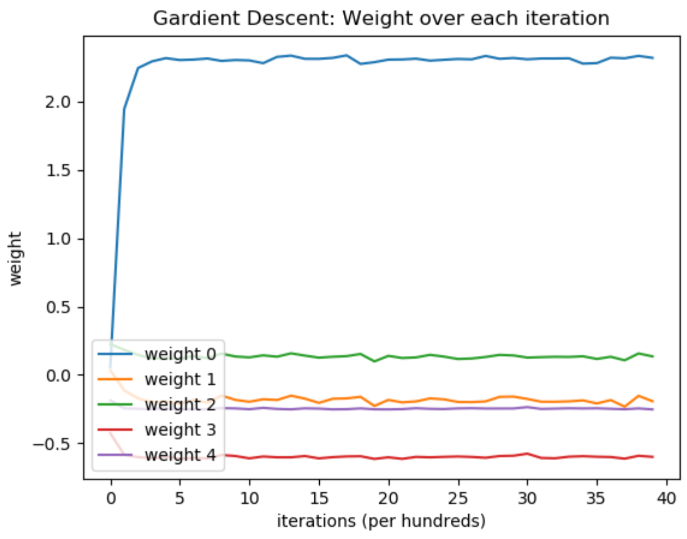
Optimized intercept = 2.315927051530593

Regularization Cost each 100 Iteration =

```
[0.06708358949597486,  0.11512615127425095,  0.11911042046052894,
0.12044892407009192,   0.118799997639671,    0.12307643119939661,
0.12488127733660377,   0.12320206004683244,  0.11325840517369945,
0.11713716861232534,   0.12145362287955862,  0.11535533479398047,
0.12112628728943248,   0.11999721760443953,  0.11531793172603244,
0.12252576080791064,   0.11653861177026781,  0.11518800514398145,
0.11669773812306036,   0.12487388118203016,  0.12083409824903144,
0.12253041478023434,   0.11674140775487653,  0.11898755484358593,
0.11909324147360661,   0.11642126098976105,  0.11711396245405348,
0.12135808540395461,   0.11608331175518721,  0.11550051635359995,
0.10998878160070044,   0.12327481942277972,  0.12338424456198846,
0.11741081799822552,   0.11570580172444003,  0.11811408394431283,
0.11772060363221493,   0.12462397194955194,  0.11428415077741116,
0.11867898598184456]
```



Cost Function over each iteration, Learning Rate = 0.01, λ = 10c



Weight Function over each iteration, Learning Rate = 0.01, λ = 10

Appendix

Code:

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


data = pd.read_csv('IRIS.csv')

# Task 1 Create 3 Binary Columns
df = pd.get_dummies(data['species'])

binary_data = pd.concat([data,df],axis=1)
print (binary_data)

# Drop Iris-versicolor
binary_data.drop(['species','Iris-versicolor','Iris-virginica'],axis=1,inplace=True)
print(binary_data.head())
print(binary_data.tail())

#Logistic Regession
input_data = binary_data.drop(binary_data.columns[[4]], axis=1)
output_data = binary_data.drop(binary_data.columns[[0,1,2,3]], axis=1)

print (input_data.shape[1])
print (output_data.shape)

def weightInitialization(n_features):
    w = np.zeros((1,n_features))
    w_0 = 0
    return w_0, w

def sigmoid_func(z):
    return 1 / (1 + np.exp(-z))

def model_optimize(w_0, w, X, Y,lambda_value):
    m = X.shape[0]
    #Prediction
    final_result = sigmoid_func(np.dot(w,X.T)+w_0)
    Y_T = Y.T
    # Normal Cost Function
    cost = (-1/m)*(np.sum((Y_T*np.log(final_result)) + ((1-Y_T)*(np.log(1-final_result)))))
    # Debugging on Regularize Cost:
    # print(len(w))
    # print("Weight 1 : {0} Power^2 : {1}".format(w[0][0],w[0][0]**2))
    # print("Weight 2 : {0} Power^2 : {1}".format(w[0][1],w[0][1]**2))
    # print("Weight 3 : {0} Power^2 : {1}".format(w[0][2],w[0][2]**2))
    # print("Weight 4 : {0} Power^2 : {1}".format(w[0][3],w[0][3]**2))
    # print("Sum : {0}".format(w[0][0]**2 + w[0][1]**2 + w[0][2]**2 + w[0][3]**2))
    # print(m)
    # regularize = (lambda_value/(2*m)) * (w[0][0]**2 + w[0][1]**2 + w[0][2]**2 + w[0][3]**2)
    # regularize = (lambda_value/(2*m)) * (np.dot(w,w.T)[0][0])
    # print("Regularize cost: {0}".format(regularize))
    # print("Cost: {0}".format(cost))
    # print((np.dot(w,w.T)))
    regularize = (lambda_value/(2*m)) * (np.dot(w,w.T)[0][0])
    cost = cost + regularize

    #Gradient calculation
    dw = (1/m)*(np.dot(X.T, (final_result-Y.T).T))
    dw_0 = (1/m)*(np.sum(final_result-Y.T))
    #Debugging Gardeint Descent
    # print("Lambda / m : {0}".format(lambda_value/m))
    # print("Vector W : {0}".format(w))
    # print("Vector W[0] : {0}".format(w[0]))
    # print("Vector W * (lambda /m ) : {0}".format(w*(lambda_value/m)))
    # print("Vector Derivertive of weight: {0}".format(dw))
    # print("Vector W * (lambda /m ).T: {0}".format((w*(lambda_value/m)).T))

    #Adding Reguralize Cost Function to Gardient
    dw = dw + (w*(lambda_value/m)).T



    grads = {"dw": dw, "dw_0": dw_0}

    return grads, cost, regularize


def model_predict(w_0, w, X, Y,lambda_value, learning_rate, no_iterations):
    costs = []
    reg_costs = []
    w0_ = []
    w_ = []
    for i in range(no_iterations):
        #
        grads, cost, reg = model_optimize(w_0,w,X,Y,lambda_value)
        #
        dw = grads["dw"]
        dw_0 = grads["dw_0"]
        #weight update
        w = w - learning_rate * (dw.T)
        w_0 = w_0 - (learning_rate * dw_0)

        if (i % 100 == 0):
```

```python
            costs.append(cost)
            reg_costs.append(reg)
            w0_.append(w_0)
            w_.append(w)

            #print("Cost after %i iteration is %f" %(i, cost))

    #final parameters
    coeff = {"w": w, "w_0": w_0}
    gradient = {"dw": dw, "dw": dw_0}

    return coeff, gradient, costs, reg_costs, w0_, w_

def model_predict_SGD(w_0, w, X, Y,lambda_value,learning_rate, no_iterations):
    costs = []
    reg_costs = []
    m = X.shape[0]
    sampling_X = []
    sampling_Y = []
    w_ = []
    w0_ = []
    for i in range(no_iterations):
        cost_temp = 0
        sampling_X = np.zeros(shape=(2,4))
        sampling_Y = np.zeros(shape=(2,1))
        for j in range(m):
            #Random Sampling
            rand_index = np.random.randint(0,m)
            sampling_X[0] = X[rand_index,:].reshape(1,X.shape[1])
            sampling_Y[0] = Y[rand_index].reshape(1,1)
            rand_index = np.random.randint(0,m)
            sampling_X[1] = X[rand_index,:].reshape(1,X.shape[1])
            sampling_Y[1] = Y[rand_index].reshape(1,1)

            # print("X_rand: {0}".format(X_rand))
            # print("Y_rand: {0}".format(Y_rand))
            # print(sampling_X)
            grads, cost, reg = model_optimize(w_0,w,sampling_X ,sampling_Y, lambda_value)

            dw = grads["dw"]
            dw_0 = grads["dw_0"]
            #weight update
            w = w - learning_rate * (dw.T)
            w_0 = w_0 - (learning_rate * dw_0)
            cost_temp = cost

        if (i % 100 == 0):
            costs.append(cost_temp)
            reg_costs.append(reg)
            w0_.append(w_0)
            w_.append(w)

            #print("Cost after %i iteration is %f" %(i, cost))

    #final parameters
    coeff = {"w": w, "w_0": w_0}
    gradient = {"dw": dw, "dw": dw_0}

    return coeff, gradient, costs, reg_costs, w0_, w_

#Logistic Regession Model
input_data = binary_data.drop(binary_data.columns[[4]], axis=1)
output_data = binary_data.drop(binary_data.columns[[0,1,2,3]], axis=1)

#Get number of features
number_features = input_data.shape[1]
print('Number of Features', number_features)
w_0, w = weightInitialization(number_features)
print ("Weight 0 : ",w_0)
print ("Weight N : ",w)
input_arr = input_data.to_numpy()
output_arr = output_data.to_numpy()

#Set Lambda
lambda_value = 1
#Gradient Descent
# coeff, gradient, costs,reg_costs, weight_0, weight = model_predict(w_0,w, input_arr, output_arr,lamb
da_value, learning_rate=0.01,no_iterations=4000)

#Stochastic Gradient Descent
coeff, gradient, costs,reg_costs, weight_0, weight = model_predict_SGD(w_0,w, input_arr, output_arr,la
mbda_value, learning_rate=0.01,no_iterations=4000)
#Final prediction
w = coeff["w"]
w_0 = coeff["w_0"]
print('Optimized weights', w)
print('Optimized intercept',w_0)
# # #
final_hypothesis = sigmoid_func(np.dot(w,input_arr.T)+w_0)
# print(reg_costs)
# print(costs)


plt.figure(1)
# Plot Cost Function
plt.plot(costs)
plt.ylabel('cost')
```

```python
plt.xlabel('iterations (per hundreds)')
plt.title('Gardient Descent: Cost Function over each iteration')

print("Regularization Costs:")
print(reg_costs,sep='\n')

# print(weight[0][0][0])
# print("Weight[0] : {0}".format(weight[0]))
# print("Length of weight: {0} X {1}".format(len(weight),len(weight[0])))

re_weight = []
for j_ in range(len(weight[0][0])):
    weight_ = []
    weight_.clear
    for i_ in range(len(weight)):
        weight_.append(weight[i_][0][j_])
    re_weight.append(weight_)

# print(len(re_weight))
# print("Length of Re_Weight[0]: {0}".format(len(re_weight[0])))
plt.figure(2)
plt.plot(weight_0)
plt.plot(re_weight[0])
plt.plot(re_weight[1])
plt.plot(re_weight[2])
plt.plot(re_weight[3])
plt.ylabel('weight')
plt.xlabel('iterations (per hundreds)')
plt.title('Gardient Descent: Weight over each iteration')
plt.legend(['weight 0', 'weight 1', 'weight 2', 'weight 3','weight 4'], loc='lower left')
plt.show()
```

References:

- https://machinelearningmedium.com/2017/09/15/regularized-logistic-regression/
- https://www.ritchieng.com/logistic-regression/
- https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc
- https://github.com/SSaishruthi/LogisticRegression_Vectorized_Implementation/blob/master/Logistic_Regression.ipynb
- https://github.com/sachinruk/deepschool.io/tree/master/DL-Keras_Tensorflow
- https://gist.github.com/sagarmainkar/41d135a04d7d3bc4098f0664fe20cf3c
- CPE603 Deep Learning, Lecture 1-5, Asst. Prof. Dr. Santitham Prom-on, Department of Computer Engineering, Faculty of Engineering King Mongkut's University of Technology Thonburi