Assignment 2: Linear and Logistic Regression

Introduction

In this exercise, you will implement linear regression and logistic regression variants and see how it works on some simple datasets. Before the tasks most important equations are explained again. You might wanna use the lecture slides / notes as well.

As in assignment 1 please submit your solutions with the grading system (you need to be either inside the HM network or connected to eduVPN to reach the online submission service). Additionally please upload a PDF of the notebook to the Moddle course with all result cells visible (see last section of this notebook).

The previous environment (python venv) used for assignment 1 should also work here to execute everything in this second assignment. See assignemnt 1 folder for the requirements.txt which can be installed with "pip install -r exercise-requirements.txt".

In [1]:
```
# used for manipulating directory paths
import os

# Scientific and vector computation for python
import numpy as np

# Plotting library
from matplotlib import pyplot

import utils

grader=utils.Grader()

# SET YOUR Authentication Token. To get the token login to http://evalml.
AUTH_TOKEN = "cd4a9aa7782d5a61a41c6dd48746e426d1f9a09a"
grader.setToken(AUTH_TOKEN)
```

Submission and Grading

After completing each part of the assignment you are asked to submit your results

In this second assignment we will cover linear regression with one feature dimension and logistic regression with two feature dimensions, as well as cost functions and parameter learning.

Required Exercises

| Task | Part |Submitted Function | Points |---------|:- |:- | :-:
| 1 | Compute cost for linear regression | `computeCost` | 20 | | 2 | Closed form solution linear regression | `normalEqn` | 15 | | 3 | Gradient descent linear regression

| `gradientDescent` | 20 | 4 | Sigmoid Function | `sigmoid` | 5
| 5 | Compute cost for logistic regression | `costFunctionLog` | 20
| 6 | Gradient descent logistic regression | `gradientDescentLog` | 15
| 7 | Predict Function logistic regression | `predict` | 5 | | | Total Points | | 100

You are allowed to submit your solutions multiple times. Correct results are not updated anymore.

At the end of each section, we have a cell which contains code for submitting the solutions thus far to the grader. Execute the cell to see your score up to the current section. For all your work to be submitted properly, you must execute those cells at least once. They must also be re-executed everytime the submitted function is updated.

1 Linear regression

In this first chapter of the assignment you will implement linear regression using a one-dimensional input and a one-dimensional target variable.

The file `Data/studenthours.txt` contains the dataset for our linear regression problem. The dataset contains 100 2d data points. This toy dataset provides a relation between study hours on the machine learning topic and gained course grades (Note: fictive dataset!). The first column are the student hours and the second column is the grade gained at the end.

The dataset is loaded from the data file into the variables `X` and `y`:

In [2]:
```
# Read comma separated data
dataLinReg = np.loadtxt(os.path.join('Data', 'studenthours.txt'), delimit
XLin, yLin = dataLinReg[:, 0], dataLinReg[:, 1]

NLin = yLin.size  # number of training examples
```

The objective of linear regression is to minimize the cost function

$$C(w) = \sum_{i=1}^{N} \left(h_w(x_i) - y_i \right)^2$$

where the hypothesis $h_w(x)$ is given by the linear model

$$h_w(x) = w^T x = w_0 + w_1 x_1$$

Recall that the parameters of your model are the w_k values. These are the values you will adjust to minimize cost $C(w)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

$$w_k = w_k - \alpha \sum_{i=1}^{N} \left(h_w(x_i) - y_i \right) x_{ik} \qquad \text{simultaneously update } w_k \text{ for all } k$$

With each step of gradient descent, your parameters w_k come closer to the optimal values that will achieve the lowest cost C(w).

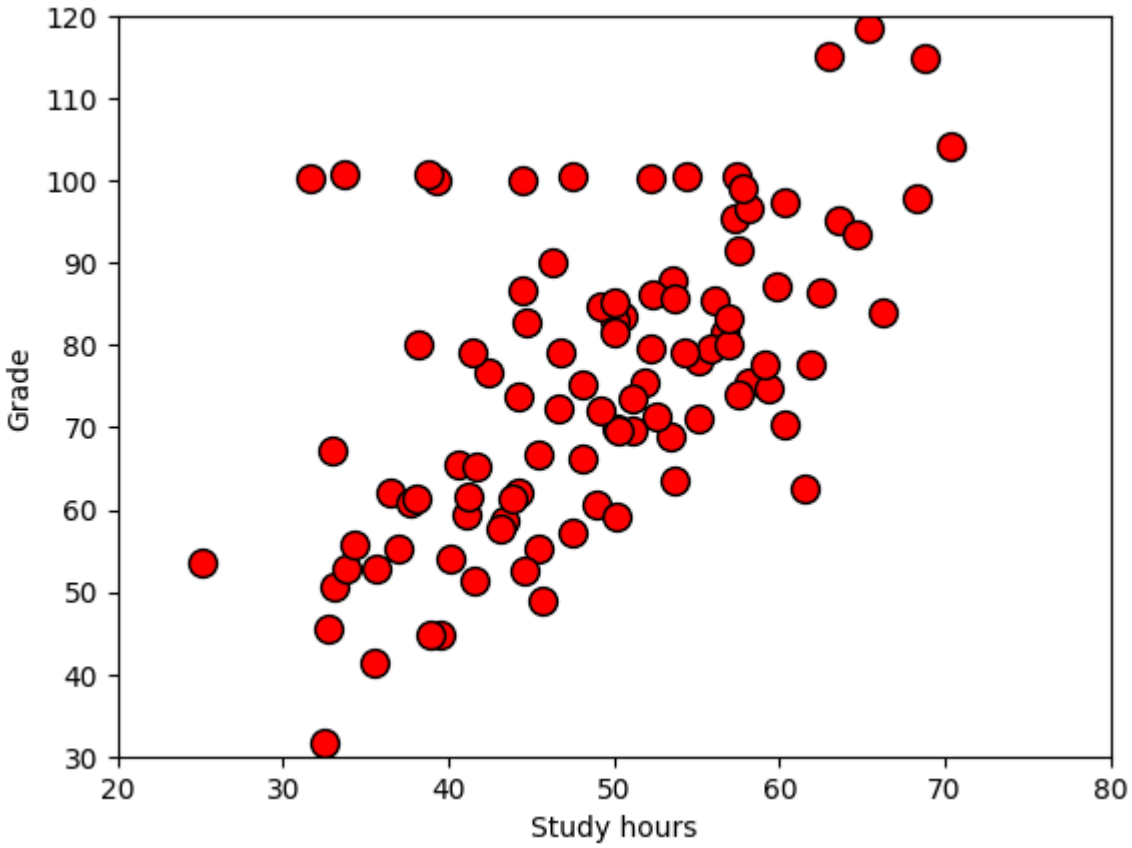
Note on implementation: We store each example as a row in the the X matrix (i.e. the Design matrix) in a Python `numpy` array. To take into account the intercept term (w_0), we add an additional first column to X and set it to all ones.

1.1 Plotting the Data

Before starting on any task, it is often a good idea 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 (student hours and grades). Many other problems that you will encounter in real life are multi-dimensional and cannot be plotted on a 2-d plot.

In [3]:
```
def plotData(x, y):
    """
    Plots the data points x and y into a new figure.

    Parameters
    ----------
    x : array_like
        Data point values for x-axis.

    y : array_like
        Data point values for y-axis. Note x and y should have the same s

    Instructions
    ------------
    Plot the training data into a figure using the "figure" and "plot"
    functions. Set the axes labels using the "xlabel" and "ylabel" functi
    Assume the study hours and grades have been passed in as the x
    and y arguments of this function.

    Hint
    ----
    You can use the 'ro' option with plot to have the markers
    appear as red circles. Furthermore, you can make the markers larger b
    using plot(..., 'ro', ms=10), where `ms` refers to marker size. You
    can also set the marker edge color using the `mec` property.
    """
    fig = pyplot.figure()  # open a new figure

    pyplot.plot(x, y, 'ro', ms=10, mec='k')
    pyplot.xlim([20,80])
    pyplot.ylim([30,120])
    pyplot.ylabel('Grade')
    pyplot.xlabel('Study hours')
```

In [4]:
```
plotData(XLin, yLin)
```



1.2 Implementation

1.2.1 Adding the intercept term

We have already set up the data for linear regression. In the following cell, we add another dimension to our data to accommodate the w_0 intercept term. Do NOT execute this cell more than once (otherwise you wil add more and more ones to the top of the vector...).

In [5]:
```
# Add a column of ones to X. The numpy function stack joins arrays along
# The first axis (axis=0) refers to rows (training examples)
# and second axis (axis=1) refers to columns (features).
XLin = np.stack([np.ones(NLin), XLin], axis=1)
```

1.2.1 Computing the cost $C(w)$

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

Your next task is to complete the code for the function `computeCost` which computes $C(w)$. As you are doing this, remember that the variables X and y are not scalar values. X is a matrix whose rows represent the examples from the training set and y is a vector whose each element represent the value at a given row of X.

In [6]:
```
def computeCost(X, y, w):
    """
    Compute cost for linear regression. Computes the cost of using w as t
    parameter for linear regression to fit the data points in X and y.

    Parameters
    ----------
    X : array_like
        The input dataset of shape (N x M+1), where N is the number of ex
        and M is the number of features. We assume a vector of one's alre
        appended to the features so we have M+1 columns.

    y : array_like
        The values of the function at each data point. This is a vector o
        shape (N, ).

    w : array_like
        The parameters for the regression function. This is a vector of
        shape (M+1, ).

    Returns
    -------
    C : float
        The value of the regression cost function.

    Instructions
    ------------
    Compute the cost of a particular choice of w.
    You should set J to the cost.
    """

    # initialize some useful values
    N = y.size  # number of training examples

    # You need to return the following variables correctly
    C = 0.0

    # ====================== YOUR CODE HERE =====================
    C = np.sum((np.dot(X, w) - y)**2)


    # ===========================================================

    return C
```

Once you have completed the function, the next step will run `computeCost` two times using two different initializations of w. You will see the cost printed to the screen.

In [7]:
```
C = computeCost(XLin, yLin, w=np.array([0.0, 0.0]))
print('With w = [0, 0] \nCost computed = %.2f' % C)
print('Expected cost value (approximately) 603923.07\n')

# further testing of the cost function
C = computeCost(XLin, yLin, w=np.array([-1, 2]))
print('With w = [-1, 2]\nCost computed = %.2f' % C)
print('Expected cost value (approximately) 73516.93')
```

```
With w = [0, 0]
Cost computed = 603923.07
Expected cost value (approximately) 603923.07

With w = [-1, 2]
Cost computed = 73516.93
Expected cost value (approximately) 73516.93
```

You should now submit your solutions by executing the following cell.

In [8]:
```
# appends the implemented function to the grader object
grader.setFunc("computeCost", computeCost)
newfunc = grader.grade()
```

```
Submitting Solutions | Programming Exercise
                              Cost lin regr |  20 /  20 | Nice work!
                                  Normal Eq |   0 /  15 | wrong
                                GD lin regr |   0 /  20 | wrong
                                    Sigmoid |   0 /   5 | wrong
                             Cost log regr. |   0 /  20 | wrong
                                GD log regr |   0 /  15 | wrong
                                    Predict |   0 /   5 | wrong
                                            --------------------------------
                                            |  20 / 100 |

                              Cost lin regr |  20 /  20 | Nice work!
                                  Normal Eq |   0 /  15 | wrong
                                GD lin regr |   0 /  20 | wrong
                                    Sigmoid |   0 /   5 | wrong
                             Cost log regr. |   0 /  20 | wrong
                                GD log regr |   0 /  15 | wrong
                                    Predict |   0 /   5 | wrong
                                            --------------------------------
                                            |  20 / 100 |
```

1.2.2 Closed-form solution

Remember the closed-form, analytical solution to linear regression (see lecture slides for derivation):

$$\bm{w} = \left(\bm{X}^T\bm{X}\right)^{-1}\bm{X}^T\bm{y}$$

This gives you directly an exact solution without the need using an iterative method like gradient descent.

Let's reload the data and add the intercept term again to be included implicitely in the feature vector.

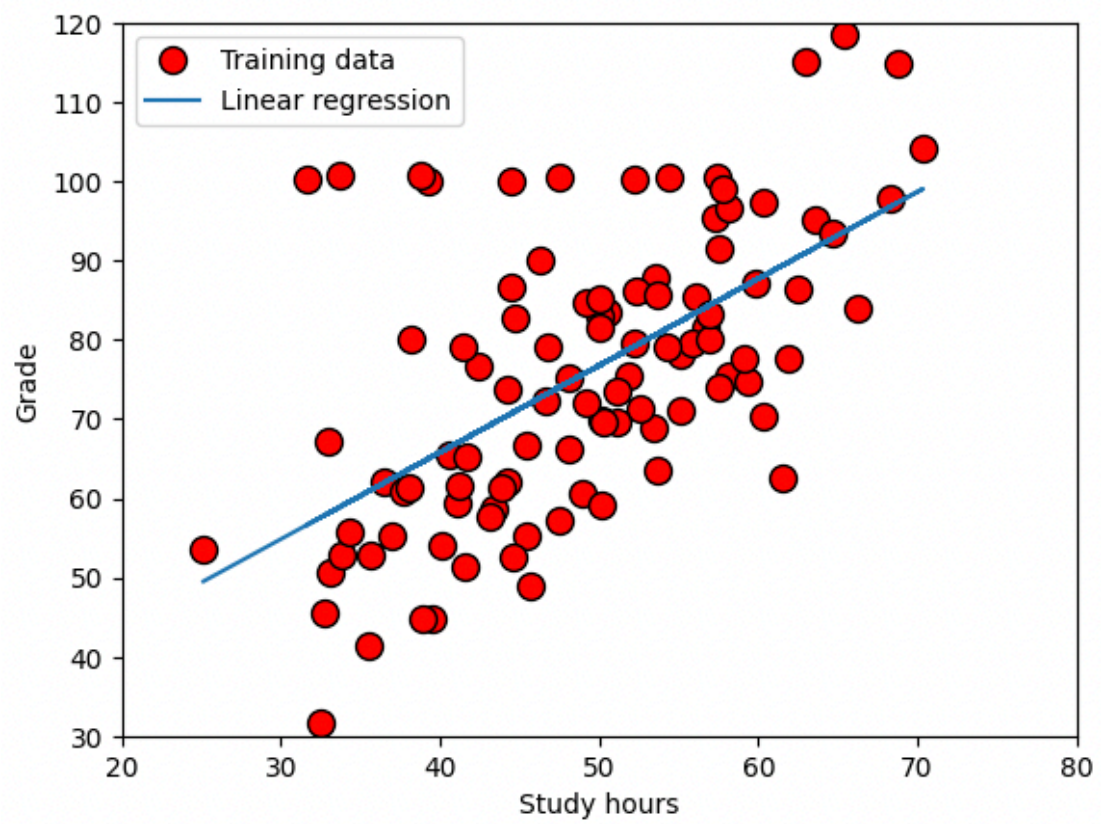
In [9]:
```
# Load data
data = np.loadtxt(os.path.join('Data', 'studenthours.txt'), delimiter=','
XLin, yLin = data[:, 0], data[:, 1]
NLin = yLin.size
XLin = np.stack([np.ones(NLin), XLin], axis=1)
```

Complete the code for the function `normalEqn` below to use the formula above to calculate w.

> Note the we can use matrix multiplication, rather than explicit summation or looping. This is also called code vectorization.

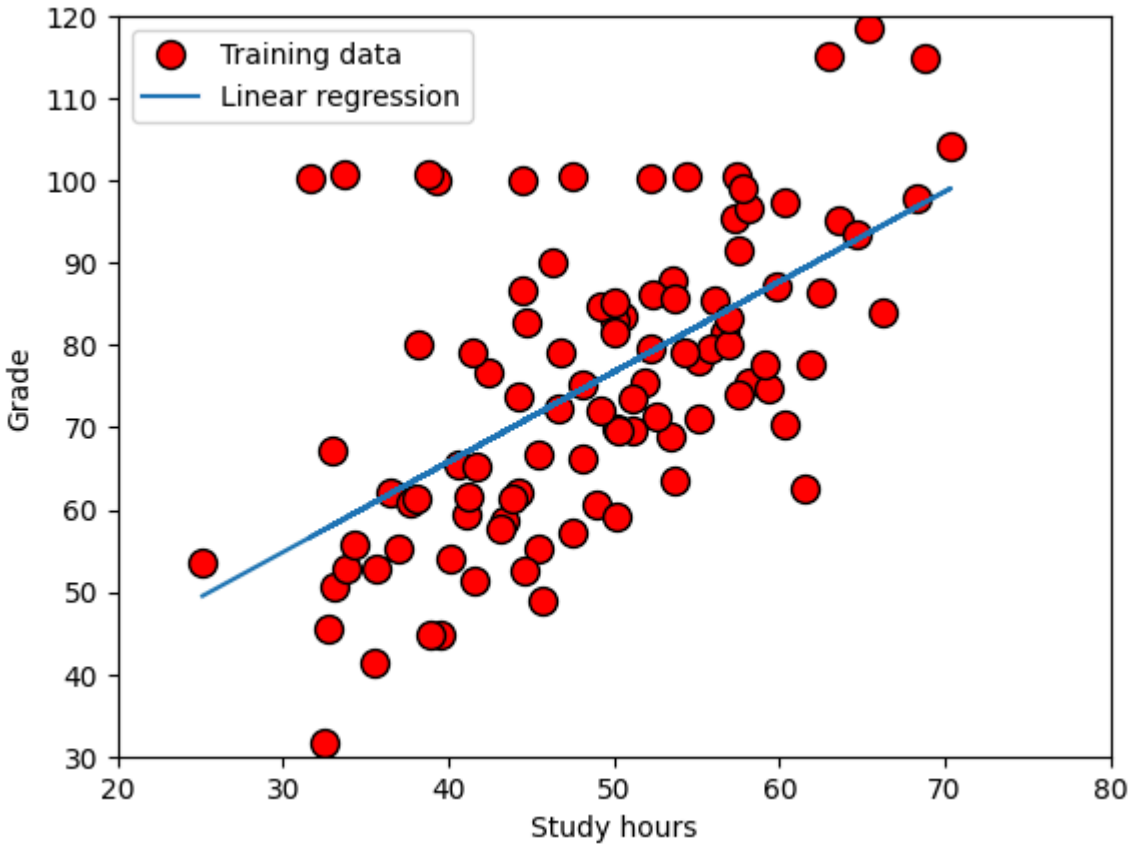
In [10]:
```
def normalEqn(X, y):
    """
    Computes the closed-form solution to linear regression using the norm

    Parameters
    ----------
    X : array_like
        The dataset of shape (N x M+1).

    y : array_like
        The value at each data point. A vector of shape (N, ).

    Returns
    -------
    w : array_like
        Estimated linear regression parameters. A vector of shape (M+1, )

    Instructions
    ------------
    Complete the code to compute the closed form solution to linear
    regression and put the result in w.

    """
    w = np.zeros(X.shape[1])

    # ===================== YOUR CODE HERE ============================
    # part 1 of the derivation of the normal equation (X^T*X)^-1
    p1 = np.linalg.inv(np.dot(np.transpose(X), X))
    # part 2 of the derivation of the normal equation X^T*y
    p2 = np.dot(np.transpose(X), y)
    w = (np.dot(p1, p2))

    # ================================================================
    return w
```

Now you can plot the found linear model. This should look similar to this one:



In [11]:
```
# plot the linear fit
plotData(XLin[:, 1], yLin)
w = normalEqn(XLin, yLin)
pyplot.plot(XLin[:, 1], np.dot(XLin, w), '-')
pyplot.ylabel('Grade')
pyplot.xlabel('Study hours')
pyplot.xlim([20,80])
pyplot.ylim([30,120])
pyplot.legend(['Training data', 'Linear regression'])
```

Out[11]:
```
<matplotlib.legend.Legend at 0x107c0c0e0>
```



Two more checks for you: 1) Compare the weight parameters. 2) Given the model parameters w you would like to predict the grade based on an input of 60 study hours. This should result in grade of 88.

In [12]:
```
# Calculate the parameters from the normal equation
w_neq = normalEqn(XLin, yLin)

# Display normal equation's result
print('w computed from the normal equations: {:s} (should be [21.95116112

# Estimate the Grade when investing 60 Study Hours
# ====================== YOUR CODE HERE ======================

# You should change this and add the correc term for predicting a grade b
hours = 0

# ============================================================

print('Predicted Grade when investing 60 Study Hours (using normal equati
```

```
w computed from the normal equations: [21.95116112  1.09652181] (should be
[21.95116112  1.09652181])
Predicted Grade when investing 60 Study Hours (using normal equations): 0
(should be 88)
```

In [13]:
```
# adds the implemented function to the grader object
grader.setFunc("normalEqn", normalEqn)
grader.grade()
```

```
Submitting Solutions | Programming Exercise
                              Cost lin regr |  20 /  20 | Nice work!
                                  Normal Eq |  15 /  15 | Nice work!
                                GD lin regr |   0 /  20 | wrong
                                    Sigmoid |   0 /   5 | wrong
                             Cost log regr. |   0 /  20 | wrong
                                GD log regr |   0 /  15 | wrong
                                    Predict |   0 /   5 | wrong
                                  --------------------------------
                                            |  35 / 100 |
```

1.2.3 Gradient Descent

In this part, you will fit the linear regression parameters w to our dataset using gradient descent.

The objective of linear regression is to minimize the cost function (note constant $1/N$ already removed)

$$C(w) = \sum_{i=1}^{N} \left(h_w(x_i) - y_i\right)^2$$

where the hypothesis $h_w(x)$ is given by the linear model

$$h_w(x) = w^T x = w_0 + w_1 x_1$$

Recall that the parameters of your model are the w_j values. These are the values you will adjust to minimize cost $C(w)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

$$w_k = w_k - \alpha \sum_{i=1}^{N} \left(h_w(x_i) - y_i\right) x_{ik} \qquad \text{simultaneously update } w_k \text{ for all } k$$

With each step of gradient descent, your parameters w_j come closer to the optimal values that will achieve the lowest cost C(w).

Implementation Note: We store each example as a row in the the X matrix in Python `numpy`. To take into account the intercept term (w_0), we add an additional first column to X and set it to all ones. This allows us to treat w_0 as simply another 'feature'.

Next, you will complete a function which implements gradient descent. The loop structure has been written for you, and you only need to supply the updates to w 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 $C(w)$ is parameterized by the vector w, not X and y. That is, we minimize the value of $C(w)$ by changing the values of the vector w, not by changing X or y. Refer to lecture slides.

The starter code for the function `gradientDescent` calls `computeCost` on every iteration and saves the cost to a `python` list. Assuming you have implemented gradient descent and `computeCost` correctly, your value of $C(w)$ should never increase, and should converge to a steady value by the end of the algorithm. Maybe you could add another cell to check this behavior in a plot, where you ploting the loss over performed iterations (e.g. using matplotlib).

In [14]:
```
def gradientDescent(X, y, w, alpha, num_iters):
    """
    Performs gradient descent to learn `w`. Updates w by taking `num_iter
    gradient steps with learning rate `alpha`.

    Parameters
    ----------
    X : array_like
        The input dataset of shape (N x M+1).

    y : array_like
        Value at given features. A vector of shape (N, ).

    w : array_like
        Initial values for the linear regression parameters.
        A vector of shape (M+1, ).

    alpha : float
        The learning rate.

    num_iters : int
        The number of iterations for gradient descent.

    Returns
    -------
    w : array_like
        The learned linear regression parameters. A vector of shape (M+1,

    C_history : list
        A python list for the values of the cost function after each iter

    Instructions
    ------------
    Peform a single gradient step on the parameter vector w.

    While debugging, it can be useful to print out the values of
    the cost function (computeCost) and gradient here.
    """
    # Initialize some useful values
    N = y.shape[0]  # number of training examples

    # make a copy of w, to avoid changing the original array, since numpy
    # are passed by reference to functions
    w = w.copy()

    C_history = [] # Use a python list to save cost in every iteration

    for i in range(num_iters):

        # ==================== YOUR CODE HERE ============================
```

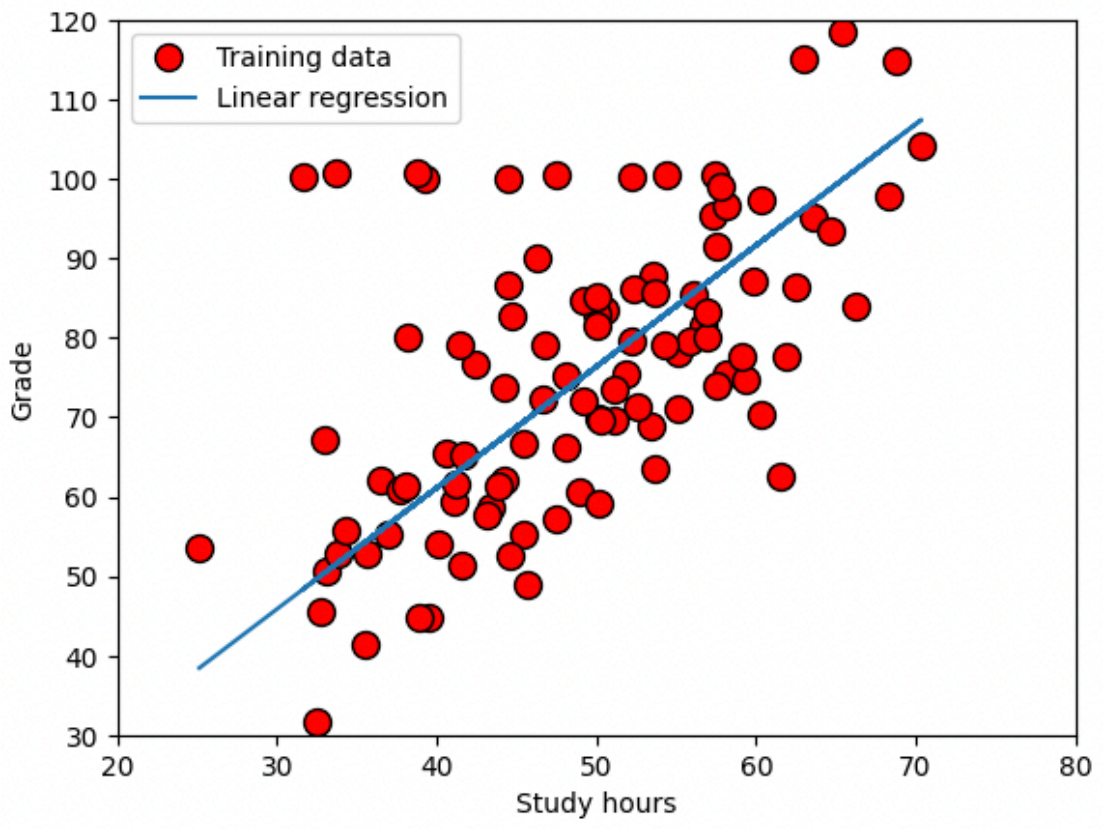
```
        for k in range (len(w)):
            w[k] = w[k] - alpha * np.sum((np.dot(X,w) - y) * X[:,k])

        # ============================================================

        # save the cost J in every iteration
        C_history.append(computeCost(X, y, w))

    return w, C_history
```

After you are finished call the implemented `gradientDescent` function and print the computed w. We initialize the w parameters to 0 and the learning rate α to 0.000001. Execute the following cell to check your code.

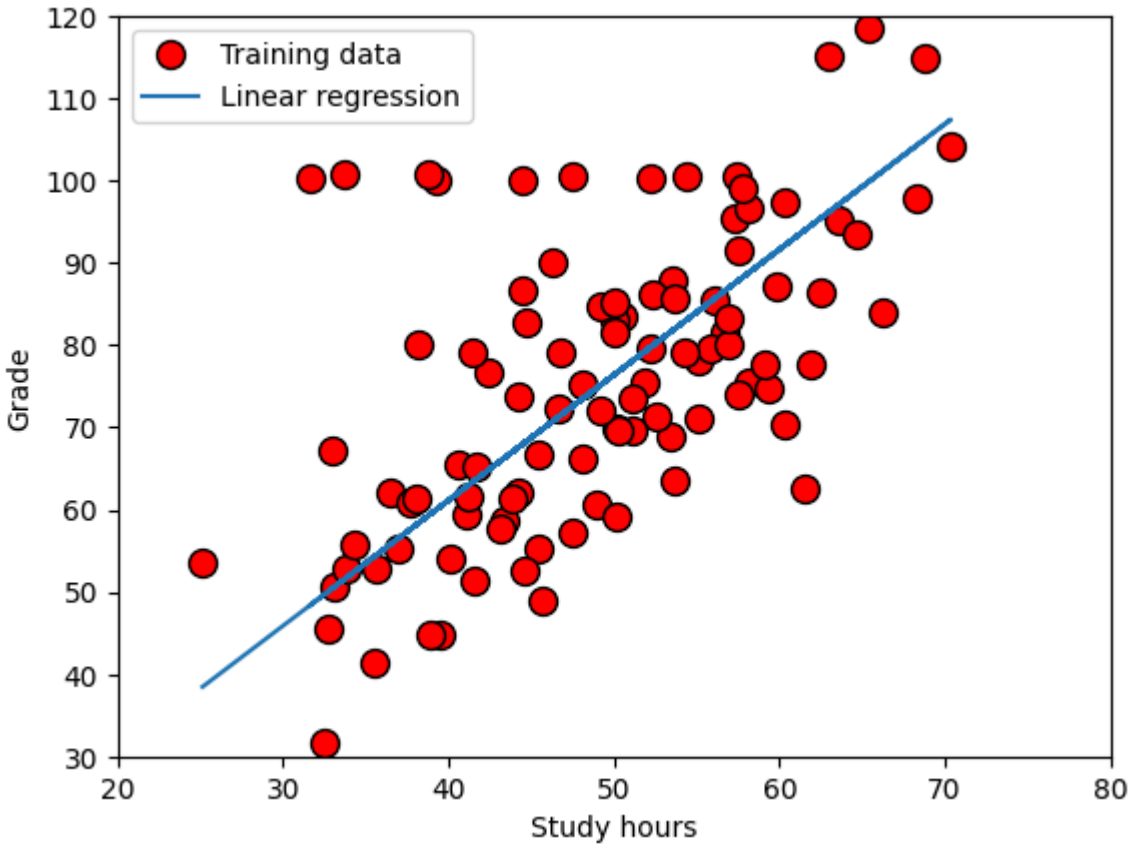
In [15]:
```
# Load data
data = np.loadtxt(os.path.join('Data', 'studenthours.txt'), delimiter=','
XLin, yLin = data[:, 0], data[:, 1]
NLin = yLin.size
XLin = np.stack([np.ones(NLin), XLin], axis=1)

# initialize fitting parameters
w = np.zeros(2)

# some gradient descent settings
iterations = 1000

# Note: The learnind rate should be quite small since we ignored the cons
alpha = 0.000001

w, C_history = gradientDescent(XLin ,yLin, w, alpha, iterations)
print('Computed cost is {}'.format(computeCost(XLin, yLin, w)))
print('w found by gradient descent: [{:.4f}, {:.4f}]'.format(*w))
print('Expected w values (approximately): [0.1126, 1.5258]')
```

```
Computed cost is 22350.28714226391
w found by gradient descent: [0.1126, 1.5258]
Expected w values (approximately): [0.1126, 1.5258]

w found by gradient descent: [0.1126, 1.5258]
Expected w values (approximately): [0.1126, 1.5258]
```

We will use your final parameters to plot the linear fit. The results should look like the following figure.



In [16]:
```
# plot the linear fit
plotData(XLin[:, 1], yLin)
pyplot.plot(XLin[:, 1], np.dot(XLin, w), '-')
pyplot.xlim([20,80])
pyplot.ylim([30,120])
pyplot.legend(['Training data', 'Linear regression']);
```



Your final values for w will also be used to make predictions on grades for 40 and 60 study hours spent.

In [17]:
```
# Predict values for population sizes of 40 and 60 study hours
predict1 = np.dot([1, 40], w)
print('For 40 study hours, we predict a grade of {:.2f} (expected: ~61.14

predict2 = np.dot([1, 60], w)
print('For 60 study hours, we predict a grade of {:.2f} (expected: ~91.66
```

```
For 40 study hours, we predict a grade of 61.14 (expected: ~61.14)

For 60 study hours, we predict a grade of 91.66 (expected: ~91.66)
```

You should now submit your solutions by executing the next cell.

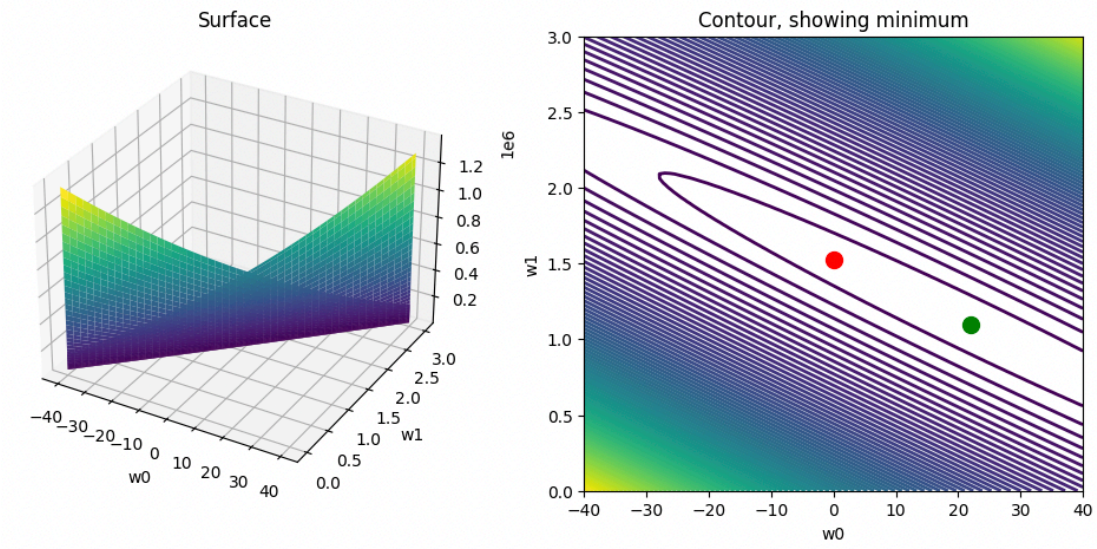
In [18]:
```
# appends the implemented function to the grader object
grader.setFunc("gradientDescent", gradientDescent)
grader.grade()
```

```
Submitting Solutions | Programming Exercise
                            Cost lin regr |  20 /  20 | Nice work!
                                Normal Eq |  15 /  15 | Nice work!
                              GD lin regr |  20 /  20 | Nice work!
                                  Sigmoid |   0 /   5 | wrong
                            Cost log regr. |   0 /  20 | wrong
                               GD log regr |   0 /  15 | wrong
                                   Predict |   0 /   5 | wrong
                                  ---------------------------------
                                            |  55 / 100 |
```

1.3 Visualizing $C(w)$

To understand the cost function $C(w)$ better, you will now plot the cost over a 2-dimensional grid of w_0 and w_1 values.

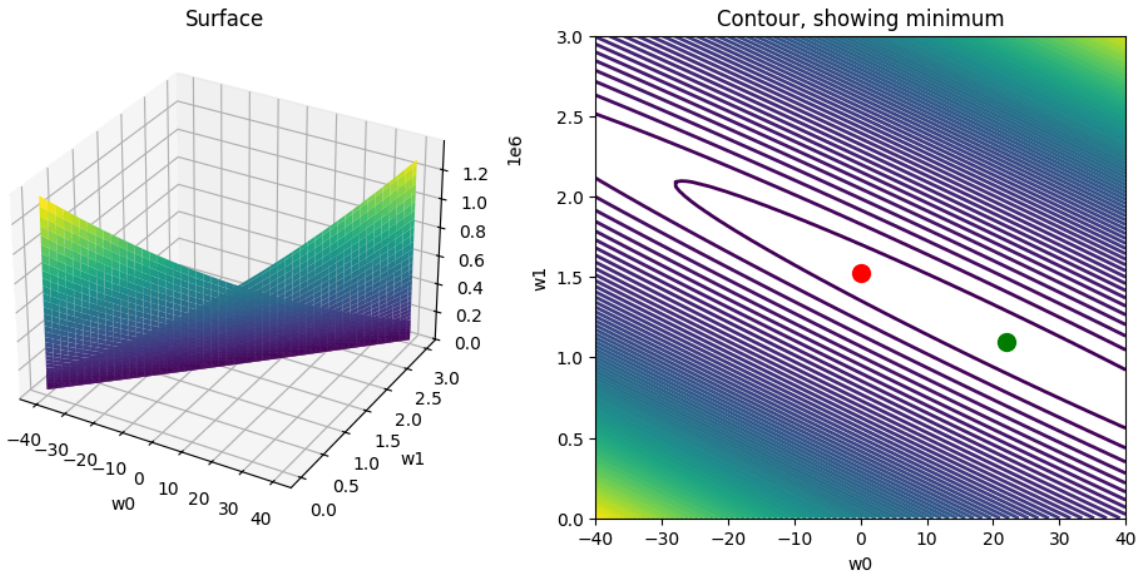
In the next cell, the code is already written to compute $C(w)$ over a grid of values using the `computeCost` function. After executing the following cell, you will have a 2-D array of $C(w)$ values. Then, those values are used to produce surface and contour plots of $C(w)$ using the matplotlib `plot_surface` and `contourf` functions. The plots should look something like the following:



The purpose of these graphs is to show you how $C(w)$ varies with changes in w_0 and w_1. The cost function $C(w)$ is bowl-shaped and has a global minimum (although the cost function became really flat around this point). This optimal minimum (global minimum, geen point) has been computed using the closed-form solution. Each step of gradient descent moves closer to this this optimal minimum, but since the gradient gets very small at the end it is hard to reach to optimal minimum using gradient descent (red point).

In [19]:
```
# grid over which we will calculate J
w0_vals = np.linspace(-40, 40, 100)
w1_vals = np.linspace(0, 3, 100)

# initialize J_vals to a matrix of 0's
J_vals = np.zeros((w0_vals.shape[0], w1_vals.shape[0]))

# Fill out J_vals
for i, w0 in enumerate(w0_vals):
    for j, w1 in enumerate(w1_vals):
        J_vals[i, j] = computeCost(XLin, yLin, [w0, w1])

# Because of the way meshgrids work in the surf command, we need to
# transpose J_vals before calling surf, or else the axes will be flipped
J_vals = J_vals.T

# surface plot
fig = pyplot.figure(figsize=(12, 5))
ax = fig.add_subplot(121, projection='3d')
ax.plot_surface(w0_vals, w1_vals, J_vals, cmap='viridis')
pyplot.xlabel('w0')
pyplot.ylabel('w1')
pyplot.title('Surface')

# contour plot
# Plot J_vals as 15 contours spaced logarithmically between 0.01 and 100
ax = pyplot.subplot(122)
pyplot.contour(w0_vals, w1_vals, J_vals, linewidths=2, cmap='viridis', le
pyplot.xlabel('w0')
pyplot.ylabel('w1')
pyplot.plot(w[0], w[1], 'ro', ms=10, lw=2)
pyplot.plot(w_neq[0], w_neq[1], 'go', ms=10, lw=2)
```

```
pyplot.title('Contour, showing minimum')
pass
```



You should now submit your solutions.

2 Logistic Regression

In this part of the exercise, you will build a logistic regression model to predict whether Suppose you are the product manager of a microchip factory and you have meassured for some microchips two different features. From these two features, you would like to determine whether the microchips should be accepted or rejected. To help you make the decision, you have a dataset of feature meassures on microchips produced in the past, from which you can build a logistic regression model.

The following cell will load the data and corresponding labels:

In [20]:
```
from utils import mapFeature

# Load Data
# note this data is already normalized
data = np.loadtxt(os.path.join('Data', 'microchips.txt'), delimiter=',')

# The first two columns contains the X values and the third column
# contains the label (y).

sel = np.ones(data.shape[0])

# You might wanna select only part of data
# sel = data[:, 0]<0.5

XLog = data[sel==1, :2]
yLog = data[sel==1, 2]
```

2.1 Visualizing the data

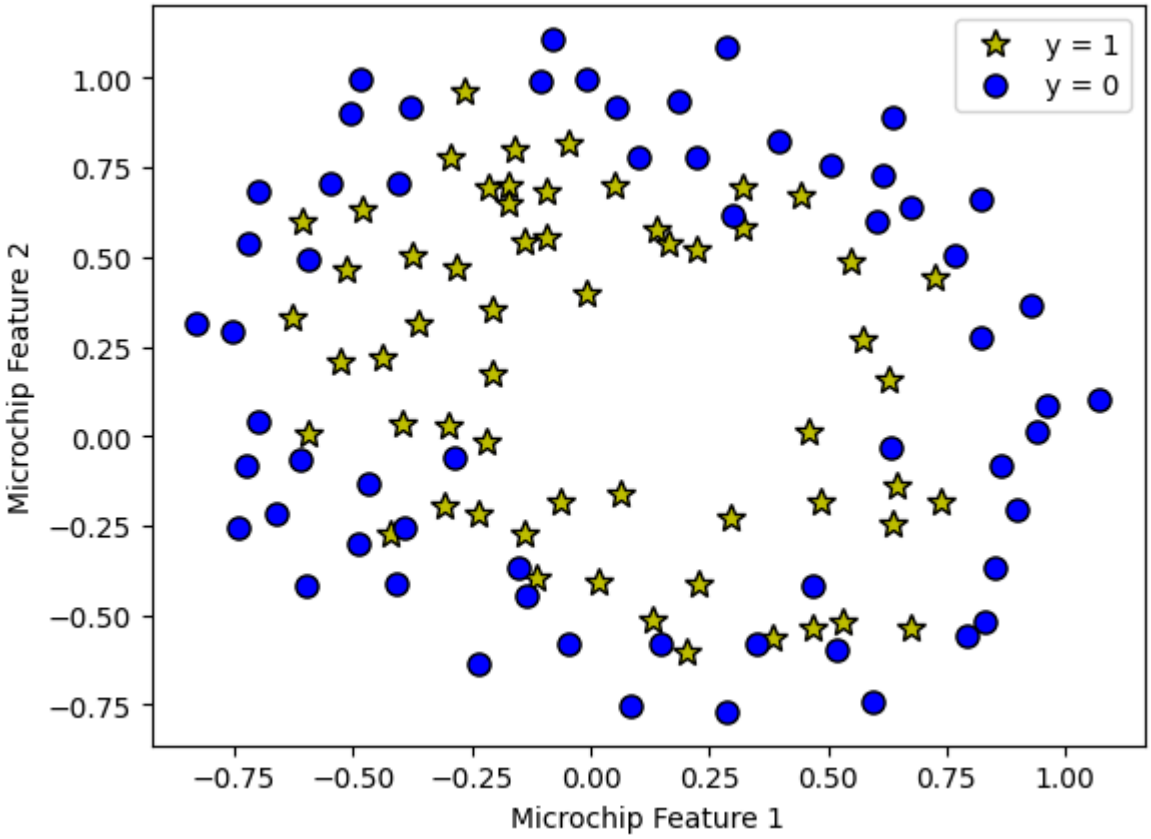
Before starting to implement the learning algorithm, it is again good to visualize the data if possible. We display the data on a 2-dimensional plot by calling the function

`plotData`.

In [21]:
```
def plotData(X, y):
    """
    Plots the data points X and y into a new figure. Plots the data
    points with * for the positive examples and o for the negative exampl

    Parameters
    ----------
    X : array_like
        An Nx2 matrix representing the dataset.

    y : array_like
        Label values for the dataset. A vector of size (N, ).

    Instructions
    ------------
    Plot the positive and negative examples on a 2D plot, using the
    option 'k*' for the positive examples and 'ko' for the negative examp
    """
    # Create New Figure
    fig = pyplot.figure()

    pos = y == 1
    neg = y == 0

    # Plot Examples
    pyplot.plot(X[pos, 0], X[pos, 1], 'k*',  mfc='y', lw=2, ms=10)
    pyplot.plot(X[neg, 0], X[neg, 1], 'ko', mfc='b', ms=8, mec='k', mew=1
```

Now, we call the implemented function to display the loaded data:

In [22]:
```
plotData(XLog, yLog)
# Labels and Legend
pyplot.xlabel('Microchip Feature 1')
pyplot.ylabel('Microchip Feature 2')

# Specified in plot order
pyplot.legend(['y = 1', 'y = 0'], loc='upper right')
pass
```



2.2 Implementation

2.2.1 The sigmoid function

Before you start with the actual cost function, recall that the logistic regression hypothesis is defined as:

$$h_w(x) = g(w^T x)$$

where function g is the sigmoid function. The sigmoid function is defined as:

$$g(z) = \frac{1}{1 + e^{-z}}$$

Your first step is to implement this function `sigmoid` so it can be called by the following functions. When you are finished, try testing a few values by calling `sigmoid(x)` in a new cell. For large positive values of `x`, the sigmoid should be close to 1, while for large negative values, the sigmoid should be close to 0. Evaluating `sigmoid(0)` should give you exactly 0.5. Your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid function on every element.

In [23]:
```
def sigmoid(z):
    """
    Compute sigmoid function given the input z.

    Parameters
    ----------
    z : array_like
        The input to the sigmoid function. This can be a 1-D vector
```

```
        or a 2-D matrix.

    Returns
    -------
    g : array_like
        The computed sigmoid function. g has the same shape as z, since
        the sigmoid is computed element-wise on z.

    Instructions
    ------------
    Compute the sigmoid of each value of z (z can be a matrix, vector or
    """
    # convert input to a numpy array
    z = np.array(z)

    # You need to return the following variables correctly
    g = np.zeros(z.shape)

    # ====================== YOUR CODE HERE ======================
    g = 1 / (1 + np.exp(-z))

    # =============================================================
    return np.float64(g)
```

The following cell evaluates the sigmoid function at `z=0`. You should get a value of 0.5. You can also try different values for `z` to experiment with the sigmoid function.

In [24]:
```
# Test the implementation of sigmoid function here
z = 0
g = sigmoid(z)
print('g(', z, ') = ', g)
```

```
g( 0 ) =  0.5
```

Execute the following cell to grade your solution to the first part of this exercise.

In [25]:
```
# adds the implemented function to the grader object
grader.setFunc("sigmoid", sigmoid)
grader.grade()
```

```
Submitting Solutions | Programming Exercise
                              Cost lin regr |  20 /  20 | Nice work!
                                  Normal Eq |  15 /  15 | Nice work!
                                GD lin regr |  20 /  20 | Nice work!
                                    Sigmoid |   5 /   5 | Nice work!
                             Cost log regr. |   0 /  20 | wrong
                                GD log regr |   0 /  15 | wrong
                                    Predict |   0 /   5 | wrong
                                  ---------------------------------
                                            |  60 / 100 |
```

2.2.2 Feature Transform

As you noticed our dataset is highly non-linear. One way to tackle this is to use base functions to transform the features and to modify the feature space. In the function `mapFeature` defined in the file `utils.py`, we will map the features into all polynomial terms of x_1 and x_2 up to the eight power:

$$\text{mapFeature}(x) = \begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1 x_2 & x_2^2 & x_1^3 & \dots & x_1 x_2^5 & x_2^6 \end{bmatrix}^T$$

Note: By this basis expension we can also experess dependencies between feature dimensions and do not work on each feature independentely. As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 48-dimensional vector. A logistic regression classifier trained on this higher-dimension feature vector will have a more complex decision boundary and will appear nonlinear when drawn in our 2-dimensional plot.
Note: While the feature mapping allows us to build a more expressive classifier, it also more susceptible to overfitting.

In [26]:
```
XLog = utils.mapFeature(XLog[:, 0], XLog[:, 1])
print(XLog.shape)
```

```
(118, 45)
```

2.2.3 Cost function and gradient

Now you will implement code to compute the cost function and gradient for logistic regression. But this time (compared to above regression implementation) we would like to add a L2 regularization term. Complete the code for the function `costFunctionLog` below to return the cost and gradient.

Recall that the Ridge Regression has the following form:

$$C(w) = \sum_{i=1}^{N} \left[-y_i \log\left(h_w\left(x_i\right)\right) - \left(1 - y_i\right) \log\left(1 - h_w\left(x_i\right)\right) \right] + \lambda \sum_{k=1}^{M} w_k^2$$

Note that the parameter w_0 should ot get reularizes, since this parameter has no effect on a specific feature dimension. The gradient of the cost function thus becomes::

$$\frac{\partial C(w)}{\partial w_0} = \sum_{i=1}^{N} \left(h_w\left(x_i\right) - y_i\right) x_i \qquad \text{for } k = 0$$

$$\frac{\partial C(w)}{\partial w_j} = \left(\sum_{i=1}^{N} \left(h_w\left(x_i\right) - y_i\right) x_i \right) + \lambda w_k \qquad \text{for } k \geq 1$$

In [27]:
```
def costFunctionLog(X, y,w, lambda_regularization=0):
    """
    Compute cost and gradient for logistic regression with regularization

    Parameters
    w : array_like
        Logistic regression parameters. A vector with shape (M, ). M is
        the number of features including any intercept. If we have mapped
        our initial features into polynomial features, then n is the tota
        number of polynomial features.

    X : array_like
        The data set with shape (N x M). m is the number of examples, and
```

```
        n is the number of features (i.e. after feature mapping if used).

    y : array_like
        The data labels. A vector with shape (N, ).

    lambda_regularization : float
        The regularization / weight decay parameter.

    Returns
    -------
    C : float
        The computed value for the regularized cost function.

    grad : array_like
        A vector of shape (M, ) which is the gradient of the cost
        function with respect to w, at the current values of w.

    Instructions
    ------------
    Compute the cost `C` of a particular choice of w.
    Compute the partial derivatives and set `grad` to the partial
    derivatives of the cost w.r.t. each parameter in w.
    """
    # Initialize some useful values
    N = y.size  # number of training examples

    # You need to return the following variables correctly
    C = 0
    grad = np.zeros(w.shape)

    # ====================== YOUR CODE HERE ======================
    h = sigmoid(X@w) # or sigmoid(X@w)

    #C = (np.sum(-y * np.log(h) - (1 - y) * np.log(1 - h)) +
    #     lambda_regularization * np.sum(w[1:] ** 2))

    C = (-1)* (y.dot(np.log(h)) + (1-y).dot(np.log(1-h))) + lambda_regula

    # Compute the gradient
    error = h - y # N dimensionen
    #grad[0] = np.sum(np.dot(X[:, 0], error))
    #grad[1:] = np.sum((np.dot(np.transpose(X[:, 1:]), error))) + (lambda

    grad = (X.T.dot(error))
    grad[1:] += (lambda_regularization) * w[1:]

    # ============================================================

    return np.float64(C), np.float64(grad)
```

Use below tests to check if your cost and gradient gets correctly computed:

In [28]:
```
# Initialize fitting parameters
initial_w = np.zeros(XLog.shape[1])

# Set regularization parameter lambda to 1
# DO NOT use `lambda` as a variable name in python
# because it is a python keyword
lambda_regularization = 1
```

```
# Compute and display initial cost and gradient for regularized logistic
# regression
cost, grad = costFunctionLog(XLog, yLog, initial_w, lambda_regularization
print('Using a weight decay parameter of {:.8f}\n'.format(lambda_regulari
print('Cost at initial w (zeros): {:.3f}'.format(cost))
print('Expected cost (approx)       : 81.791\n')

print('Gradient at initial w (zeros) - first five values only:')
print('\t[{:.4f}, {:.4f}, {:.4f}, {:.4f}, {:.4f}]'.format(*grad[:5]))
print('Expected gradients (approx) - first five values only:')
print('\t[1.0000, 2.2170, 0.0092, 5.9407, 1.3572]\n')


# Compute and display cost and gradient
# with all-ones w and lambda = 10
lambda_regularization = 0.1
test_w = np.ones(XLog.shape[1])
cost, grad = costFunctionLog(XLog, yLog, test_w, lambda_regularization)

print('------------------------------\n')
print('Using a weight decay parameter of {:.8f}\n'.format(lambda_regulari
print('Cost at test w    : {:.2f}'.format(cost))
print('Expected cost (approx): 282.29\n')

print('Gradient at initial w (zeros) - first five values only:')
print('\t[{:.4f}, {:.4f}, {:.4f}, {:.4f}, {:.4f}]'.format(*grad[:5]))
print('Expected gradients (approx) - first five values only:')
print('\t[40.9838, 9.2252, 13.1992, 16.9120, 0.9340]')
```

```
Using a weight decay parameter of 1.00000000

Cost at initial w (zeros): 81.791
Expected cost (approx)       : 81.791

Gradient at initial w (zeros) - first five values only:
        [1.0000, 2.2170, 0.0092, 5.9407, 1.3572]
Expected gradients (approx) - first five values only:
        [1.0000, 2.2170, 0.0092, 5.9407, 1.3572]

------------------------------

Using a weight decay parameter of 0.10000000

Cost at test w    : 282.29
Expected cost (approx): 282.29

Gradient at initial w (zeros) - first five values only:
        [40.9838, 9.2252, 13.1992, 16.9120, 0.9340]
Expected gradients (approx) - first five values only:
        [40.9838, 9.2252, 13.1992, 16.9120, 0.9340]
```

You should now submit your solutions.

In [29]:
```
# appends the implemented function to the grader object
grader.setFunc("costFunctionLog", costFunctionLog)
grader.grade()
```

```
Submitting Solutions | Programming Exercise
                              Cost lin regr |  20 /  20 | Nice work!
                                  Normal Eq |  15 /  15 | Nice work!
                                GD lin regr |  20 /  20 | Nice work!
                                    Sigmoid |   5 /   5 | Nice work!
                             Cost log regr. |  20 /  20 | Nice work!
                                GD log regr |   0 /  15 | wrong
                                    Predict |   0 /   5 | wrong
                                ------------------------------
                                            |  80 / 100 |
```

2.2.4 Gradient Descent

In this part, you will fit the logistic regression parameters w to our dataset using gradient descent. This time our function gets an additioanl attribute (λ) to control the regularization force (weight decay).

In [30]:
```
def gradientDescentLog(X, y, w, alpha, num_iters, lambda_regularization=1
    """
    Performs gradient descent to learn `w`. Updates w by taking `num_iter
    gradient steps with learning rate `alpha`.

    Parameters
    ----------
    X : array_like
        The input dataset of shape (N x M+1).

    y : array_like
        Value at given features. A vector of shape (N, ).

    w : array_like
        Initial values for the linear regression parameters.
        A vector of shape (M+1, ).

    alpha : float
        The learning rate.

    num_iters : int
        The number of iterations for gradient descent.

    lambda_regularization : float
        Weight decay parameter. Weight factor for regularization term.

    Returns
    -------
    w : array_like
        The learned linear regression parameters. A vector of shape (M+1,

    C_history : list
        A python list for the values of the cost function after each iter

    Instructions
    ------------
    Peform a single gradient step on the parameter vector w.

    While debugging, it can be useful to print out the values of
    the cost function (computeCost) and gradient here.
```

```
    """
    # Initialize some useful values
    m = y.shape[0]  # number of training examples

    # make a copy of w, to avoid changing the original array, since numpy
    # are passed by reference to functions
    w = w.copy()

    C_history = [] # Use a python list to save cost in every iteration

    cost = 0
    for i in range(num_iters):

        # ===================== YOUR CODE HERE ============================
        cost, grad = costFunctionLog(X, y, w, lambda_regularization)

        #for k in range(len(w)):
        w = w - alpha * grad

        # =================================================================

        # save the cost J in every iteration
        C_history.append(cost)

    return np.float64(w), np.float64(C_history)
```

In [31]:
```
# initialize fitting parameters
w = np.zeros(XLog.shape[1])

# some gradient descent settings
iterations = 10000
alpha = 0.1

lambda_regularization = 0.01
w, C_history = gradientDescentLog(XLog ,yLog, w, alpha, iterations, lambd
print('w found by gradient descent (first six): [{:.4f}, {:.4f}, {:.4f},
print('Expected w values (approximately): [3.7118, 1.8446, 4.7629, -5.763
```

```
w found by gradient descent (first six): [3.7118, 1.8446, 4.7629, -5.7630,
-6.6632, -5.2784]
Expected w values (approximately): [3.7118, 1.8446, 4.7629, -5.7630, -6.66
32, -5.2784]
```

You should now submit your solutions.

In [32]:
```
# appends the implemented function to the grader object
grader.setFunc("gradientDescentLog", gradientDescentLog)
grader.grade()
```

```
Submitting Solutions | Programming Exercise
                              Cost lin regr |  20 /  20 | Nice work!
                                  Normal Eq |  15 /  15 | Nice work!
                                GD lin regr |  20 /  20 | Nice work!
                                    Sigmoid |   5 /   5 | Nice work!
                             Cost log regr. |  20 /  20 | Nice work!
                                GD log regr |  15 /  15 | Nice work!
                                    Predict |   0 /   5 | wrong
                                  ----------------------------------
                                            |  95 / 100 |
```

2.3 Evaluating logistic regression

After having learned the paramater we would like to use them to predict the class of a new sample given its features. Your task is to complete the code in function `predict`. The predict function will produce “1” or “0” predictions given a dataset and a learned parameter vector w.

In [33]:
```
def predict(w, X):
    """
    Predict whether the label is 0 or 1 using learned logistic regression
    Computes the predictions for X using a threshold at 0.5
    (i.e., if sigmoid(w.T*x) >= 0.5, predict 1)

    Parameters
    ----------
    w : array_like
        Parameters for logistic regression. A vecotor of shape (M+1, ).

    X : array_like
        The data to use for computing predictions. The rows is the number
        of points to compute predictions, and columns is the number of
        features.

    Returns
    -------
    p : array_like
        Predictions and 0 or 1 for each row in X.

    Instructions
    ------------
    Complete the following code to make predictions using your learned
    logistic regression parameters.You should set p to a vector of 0's an
    """
    m = X.shape[0] # Number of training examples

    # You need to return the following variables correctly
    p = np.zeros(m)

    # ====================== YOUR CODE HERE ======================
    p = sigmoid(np.dot(X, w))
    for i in range(len(p)):
        if p[i] >= 0.5:
            p[i] = 1
        else:
            p[i] = 0
    p = p.astype(int)
    # ============================================================

    return p
```

After you have completed the code in `predict`, we proceed to report the training accuracy of your classifier by computing the percentage of examples classified correctly.

In [34]:
```
#  Predict probability for a student with score 45 on exam 1
#  and score 85 on exam 2
# Compute accuracy on our training set
p = predict(w, XLog)
print('Train Accuracy: {:.2f} %'.format(np.mean(p == yLog) * 100))
print('Expected accuracy (approx): 83.90 %')
```

```
Train Accuracy: 83.90 %
Expected accuracy (approx): 83.90 %
```

You should now submit your solutions.

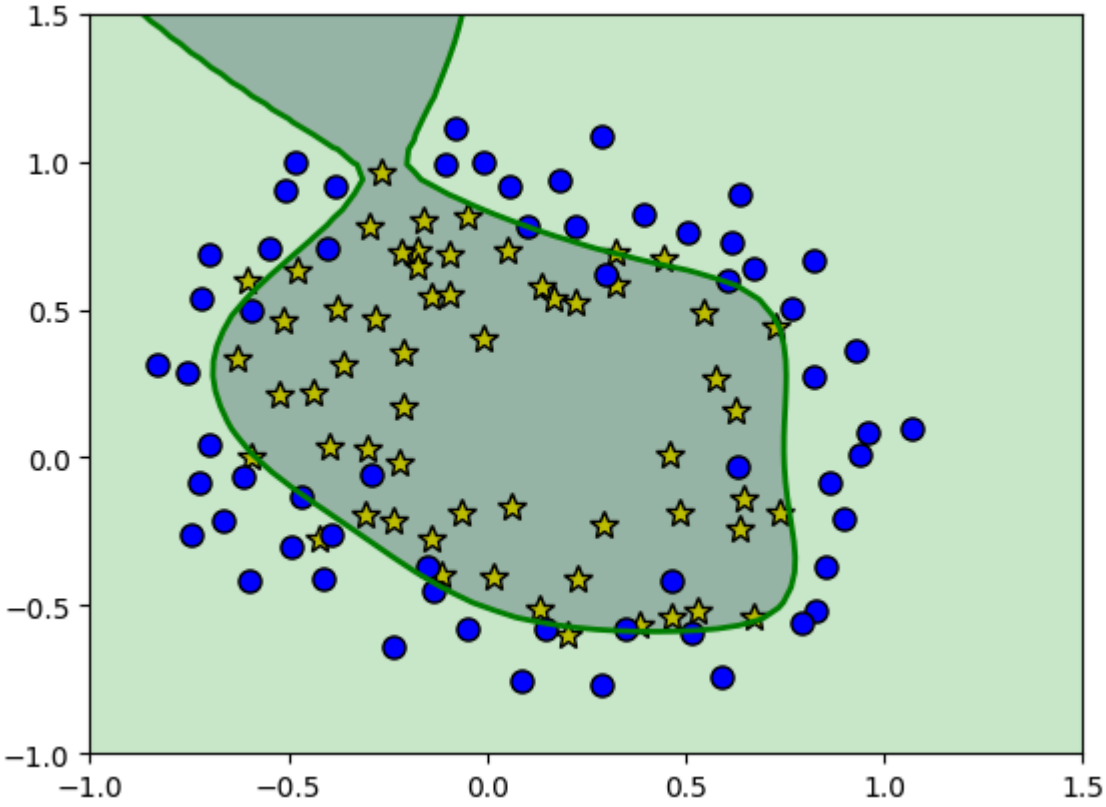
In [35]:
```
# appends the implemented function to the grader object
grader.setFunc("predict", predict)
grader.grade()
```

```
Submitting Solutions | Programming Exercise
                              Cost lin regr |  20 /  20 | Nice work!
                                  Normal Eq |  15 /  15 | Nice work!
                                GD lin regr |  20 /  20 | Nice work!
                                    Sigmoid |   5 /   5 | Nice work!
                             Cost log regr. |  20 /  20 | Nice work!
                                GD log regr |  15 /  15 | Nice work!
                                    Predict |   5 /   5 | Nice work!
                                  ------------------------------
                                            | 100 / 100 |
```

Plot the decision boundary of the unregularized version.

In [36]:
```
# Lean parameters (unregularized)
w = np.zeros(XLog.shape[1])
# some gradient descent settings
iterations = 10000
alpha = 0.1
lambda_regularization = 0
w, C_history = gradientDescentLog(XLog ,yLog, w, alpha, iterations, lambd

#Evaluate
p = predict(w, XLog)
print('Train Accuracy: {:.2f} %'.format(np.mean(p == yLog) * 100))

# Plot Boundary
utils.plotDecisionBoundary(plotData, w, XLog, yLog)
```

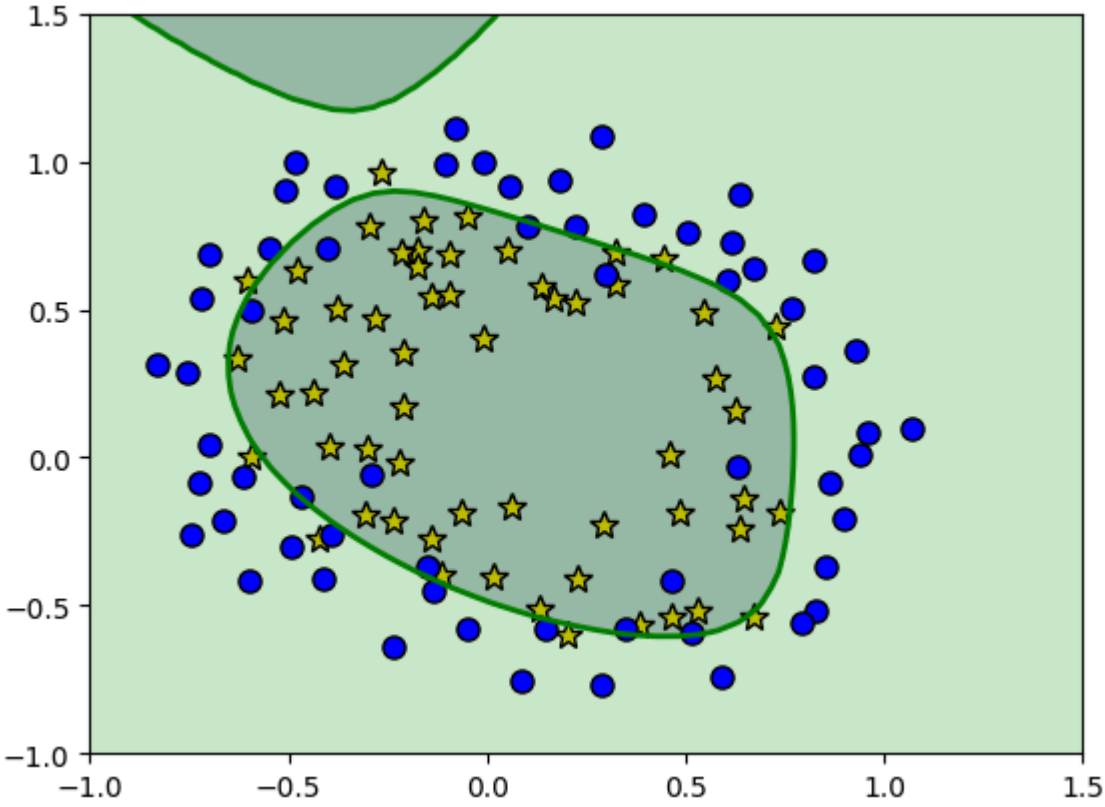
```
Train Accuracy: 87.29 %
```



Plot the decision boundary of the unregularized version. Notice that the training accuracy goes slieghtly down, but the shape of the contour looks more reasonable. Especially for more complex models this regularization leads to a better generalization. In general the importance of the prior (i.e. the regularization) decreases when you have enough and data.

In [37]:
```
# Lean parameters (unregularized)
w = np.zeros(XLog.shape[1])
# some gradient descent settings
iterations = 10000
alpha = 0.1
lambda_regularization = 0.01
w, C_history = gradientDescentLog(XLog ,yLog, w, alpha, iterations,lambda

#Evaluate
p = predict(w, XLog)
print('Train Accuracy: {:.2f} %'.format(np.mean(p == yLog) * 100))

# Plot Boundary
utils.plotDecisionBoundary(plotData, w, XLog, yLog)
```

```
Train Accuracy: 83.90 %
```



3 Uploading your PDF

Well done. You are now ready with this assignment. Please make sure that ALL code cells are executed and create a PDF from the notebook: Use the File-->Print... function in Juypter Lab (not via the Browser, and not export as PDF in Jupyter Lab). Upload this PDF to Moodle in the assignment section and in the correct assignment number.

4 References

Microchip Dataset (Logisitc Regression) from Andrew Ng, Machine Learning CS229/Coursera
Notebook has been inspired from Andrew Ng, Machine Learning CS229/Coursera