Machine Learning

CS229

September 8, 2024

Contents

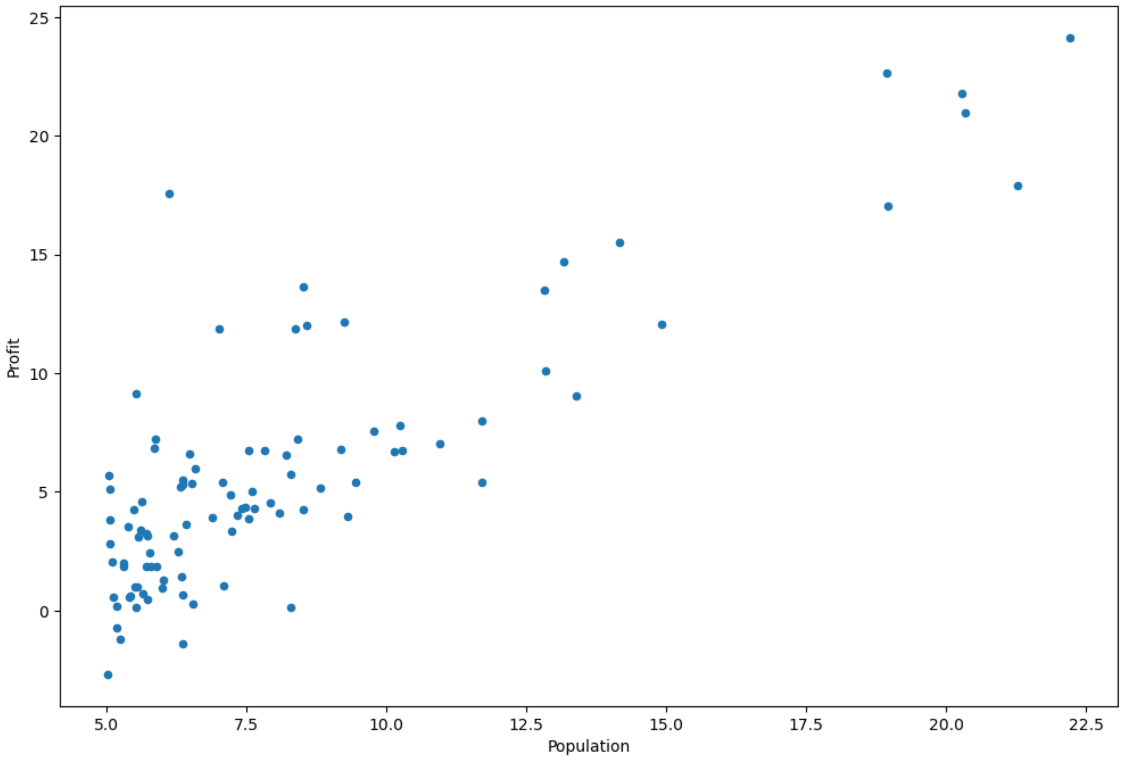
Chapter 1

Linear Regression

1.1 Linear regression with one variable

target: predict profits
data: ex1data1.txt - population(just one feature) | profit

1.1.1 Plotting the data



1.1.2 Gradient Descent

target: fit θ to dataset (θ: to minimize cost $J(\theta)$)
variable: x - feature

Update Equations

cost function:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

hypothesis:

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1 (+ ... + \theta_n x_n)$$

```
def computeCost(X, y, theta):
  inner = np.power(((X * theta.T) - y), 2)
  return np.sum(inner) / (2 * len(X))
```

in batch gradient descent:

$$\theta_j := \theta_j = \alpha \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

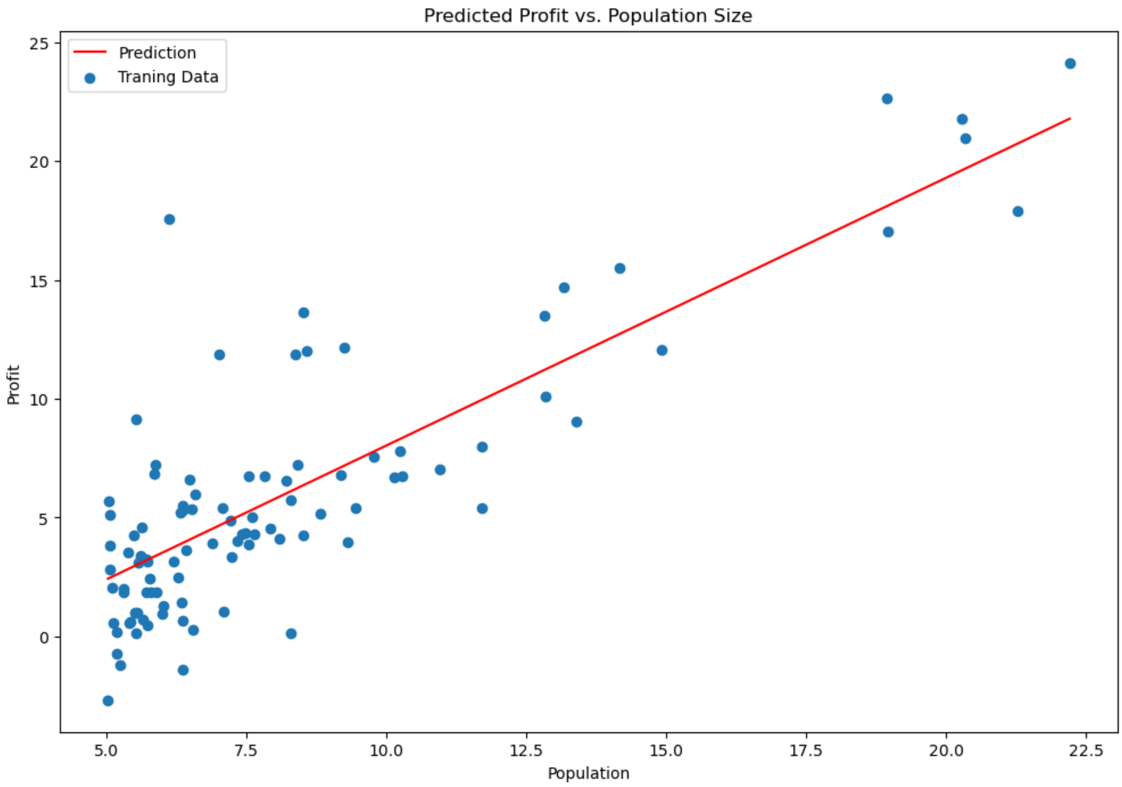
```
def gradientDescent(X, y, theta, alpha, iters):
  temp = np.matrix(np.zeros(theta.shape))
  parameters = int(theta.ravel().shape[1])
  cost = np.zeros(iters)

  for i in range(iters):
    error = (X * theta.T) - y

    for j in range(parameters):
      term = np.multiply(error, X[:,j])
      temp[0,j] = theta[0,j] - ((alpha / len(X)) * np.sum(term))

    theta = temp
    cost[i] = computeCost(X, y, theta)

  return theta, cost
```

1.1.3 Computing the cost $J(\theta)$

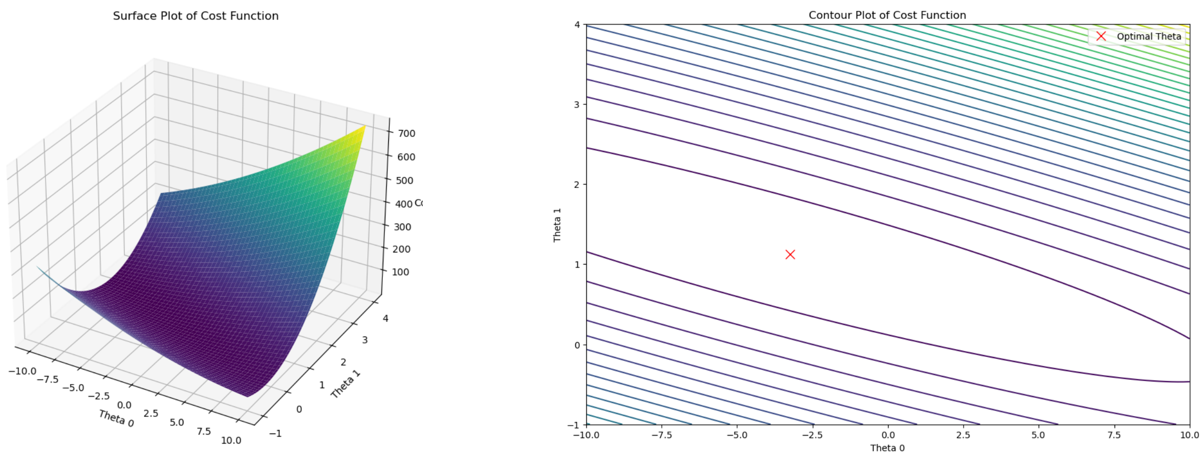
Implementation

```
alpha = 0.01
iters = 1000
Theta, cost = gradientDescent(X, y, theta, alpha, iters)
computeCost(X, y, Theta)
```

1.1.4 Visualizing regression



1.1.5 Visualizing $J(\theta)$



1.2 Linear regression with multiple variables

multiple variables:
target: predict the prices of houses

1.2.1 Feature Normalization

When features differ by orders of magnitude, first performing feature scaling can make gradient descent converge much more quickly.

formula:

$$z = \frac{x - \mu}{\sigma}$$

mean $= \mu$
standard deviation $= \sigma$

results:

	Size	Bedrooms	Price
0	2104	3	399900
1	1600	3	329900
2	2400	3	369000
3	1416	2	232000
4	3000	4	539900

	Size	Bedrooms	Price
0	0.130010	-0.223675	0.475747
1	-0.504190	-0.223675	-0.084074
2	0.502476	-0.223675	0.228626
3	-0.735723	-1.537767	-0.867025
4	1.257476	1.090417	1.595389

alternative: max-min

1.2.2 Gradient Descent

Update Equations

cost function:

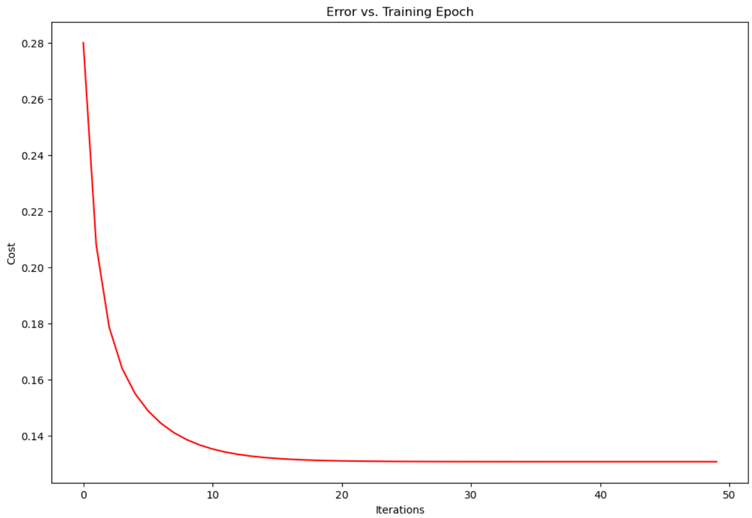
$$J(\theta) = \frac{1}{2m}(X\theta - \vec{y})^T(X\theta - \vec{y})$$

where

$$X = \begin{bmatrix} - & (x^{(1)})^T & - \\ - & (x^{(2)})^T & - \\ & \vdots & \\ - & (x^{(m)})^T & - \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Selecting learning rates

try out different learning rates
find a learning rate that converges quickly - less than 50 iterations



1.2.3 Normal Equations

step1.find θ

closed form solution to linear regression:

$$\theta = (X^T X)^{-1} X^T \vec{y}$$

- does not require any feature scaling
- will get an exact solution in one calculation
- no "loop until convergence" like in gradient descent

step2.predict

make a price prediction for a 1650-square-foot house with 3 bedrooms with θ

Chapter 2

Logistic Regression

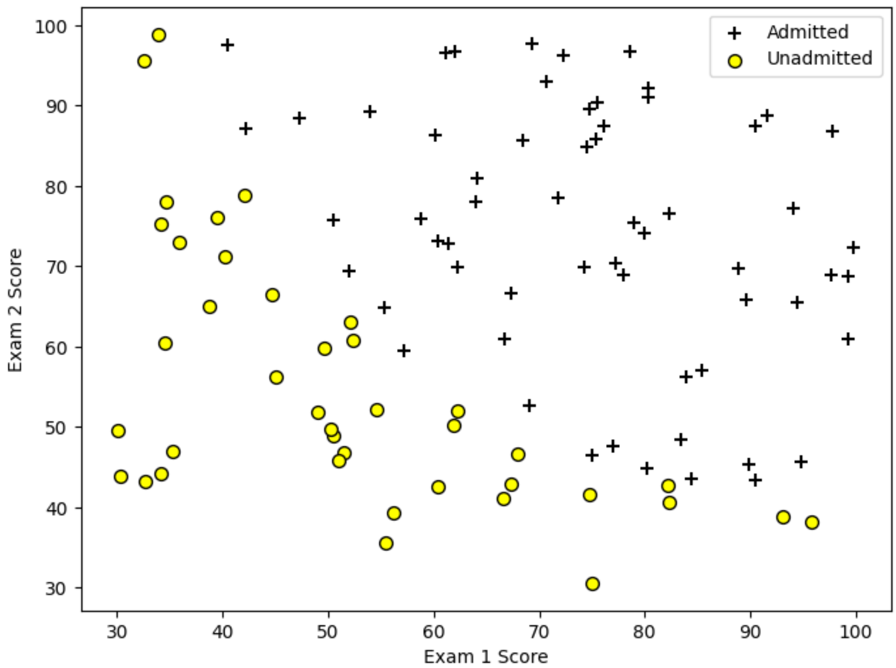
2.1 Logistic Regression

`only able to find a linear decision boundary`

target: predict whether a student gets permitted into a university
- by building a classfication model

data: historical data from previous applicants
(scores on two exams and the admissions decision)
- as a training set for logistic regression

2.1.1 Visualizing the data



2.1.2 Implementation

Sigmoid Function

logistic regression hypothesis:

$$h_\theta(x) = g(\theta^T x)$$

where g is the **sigmoid function:**

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

Cost function and gradient

cost function in logistic regression:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$$

gradient of the cost: (a vector of same length as θ)
the j^{th} element (for j = 0, 1,...,n):

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

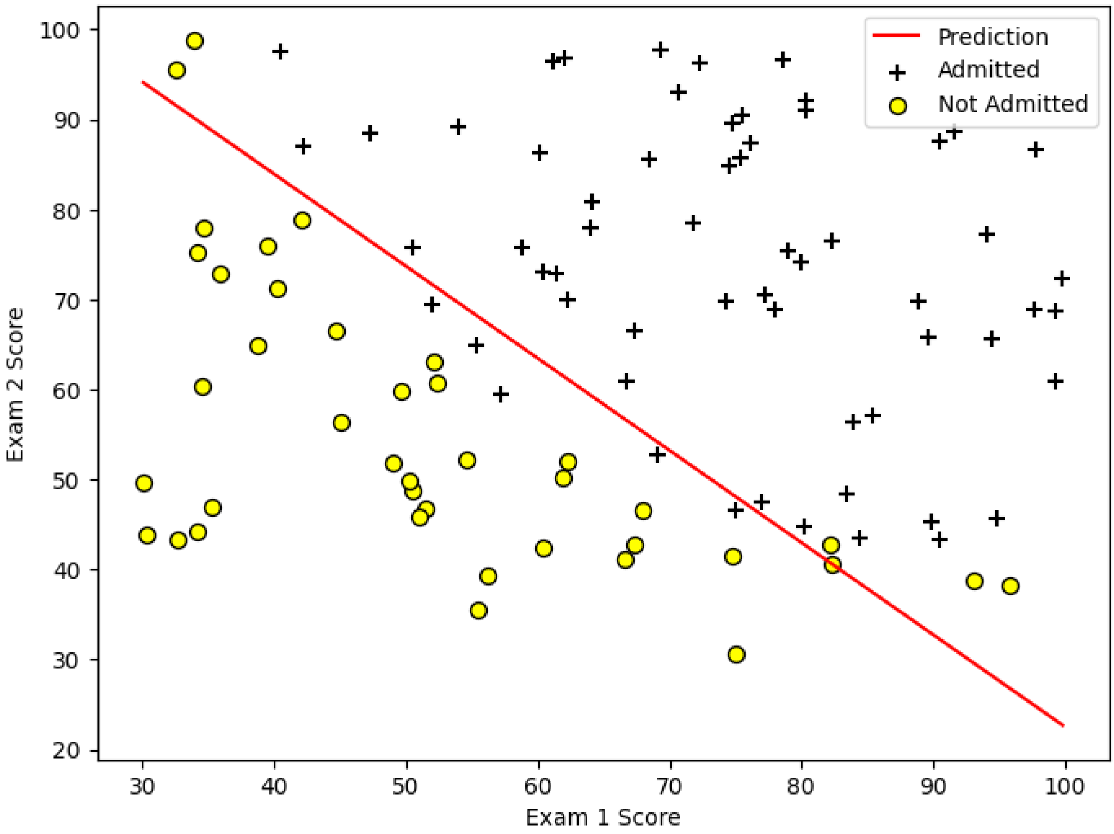
Note: while this gradient looks identical to the linear regression gradient, the formula is actually different because linear and logistic regression have different definitions of $h_\theta(x)$

Learning parameters using fminunc (scipy.optimize.fmin_tnc in Python instead)

```
import scipy.optimize as opt
result = opt.fmin_tnc(func=cost, x0=theta, fprime=gradient, args=(X, y))

theta_min = np.matrix(result[0])
```

2.1.3 Visualing regression

```
x = np.linspace(data['Exam 1'].min(),data['Exam 1'].max(),100)
decision_boundary = -(theta_min[0,1]*x+theta_min[0,0]) / theta_min[0,2]
```



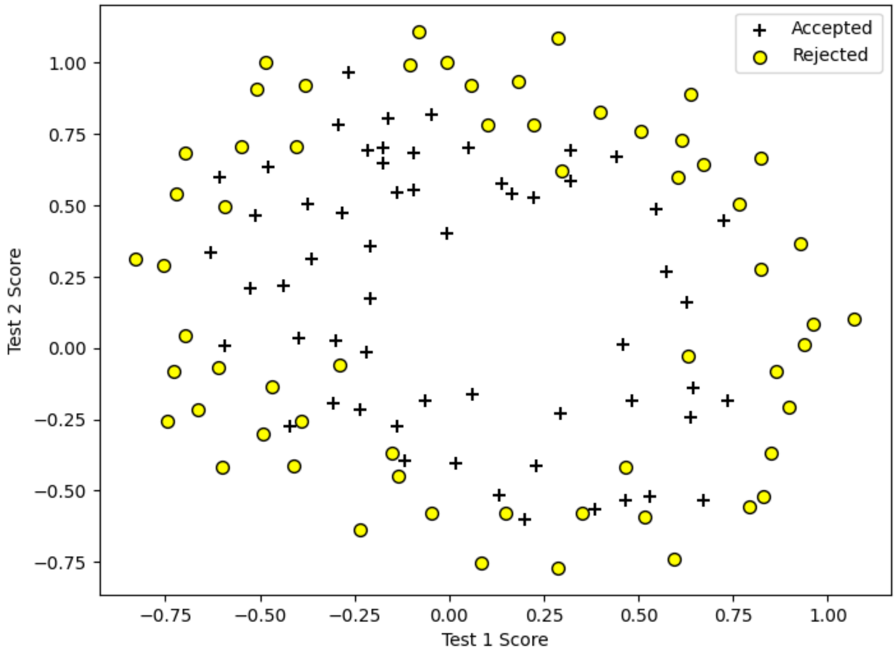
Evaluating logistic regression

After learning the parameters, we can use the model to predic whether a particular student will be admitted. For a student with an Exam 1 score of 45 and an Exam 2 score of 85, you should expect to see an admission probability of 0.776.

```
def predict(theta, X):
  probability = sigmoid(X * theta.T)
  # return [1 if x >= 0.5 else 0 for x in probability]
  return probability

predict(theta_min, [1,45,85])[0,0]
```

2.2 Regularized logistic regression

2.2.1 Visualizing the data



2.2.2 Feature mapping

One way to fit the data better is to create more feature form each data point.âĂŤâĂŤmap the features into all polynomial terms of x_1 and x_2 up to the sixth power.

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

- vector of two features $\rightarrow$ 28-dimensional vector

While the feature mapping allows us to build a more expressive classifier,it also more susceptible to overfitting.

2.2.3 Cost function and gradient

cost function: in logistic regression

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

(**note**: do not regularize θ_0)

the j^{th} element (for j = 0, 1,...,n):

$$\frac{\partial J(\theta)}{\partial\theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad for\ j\ = 1$$

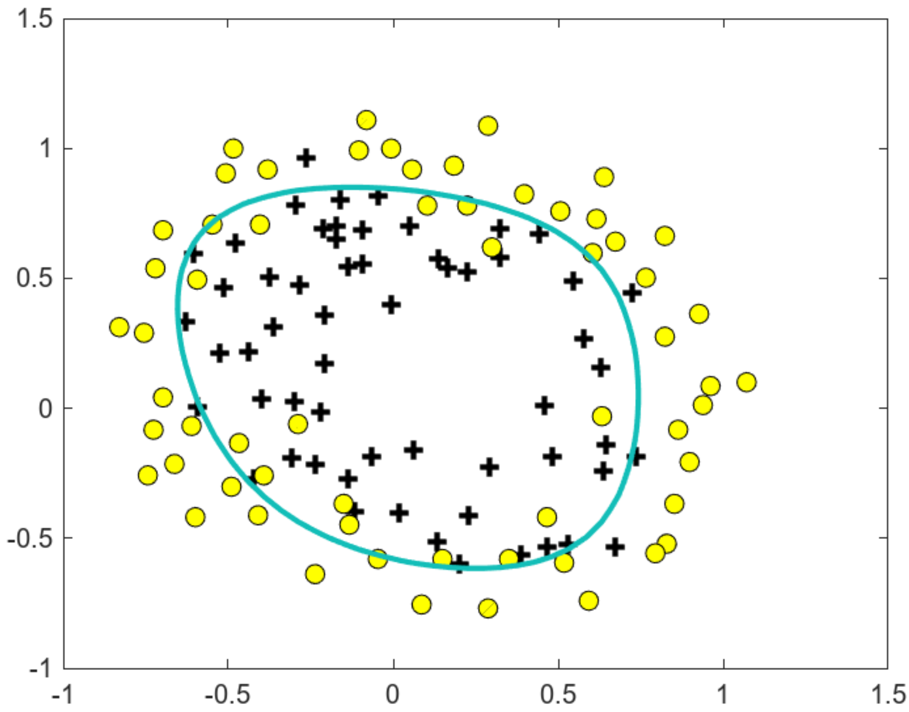
$$\frac{\partial J(\theta)}{\partial\theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j \quad for\ j\ \geq 1$$

Learning parameters using fminunc (scipy.optimize.fmin_tnc in Python instead)

2.2.4 Plotting the decision boundary

```
options = optimset('GradObj', 'on', 'MaxIter', 400);
[theta, J, exit_flag] = fminunc(@(t)(costFunctionReg(t, X_poly, y, lambda))
    , initial_theta, options);
fprintf('Final␣cost:␣%f\n', J);
plotDecisionBoundary(theta, X_poly, y);

function plotDecisionBoundary(theta, X, y)
    plotData(X(:,2:3), y);
    hold on

    u = linspace(-1, 1.5, 50);
    v = linspace(-1, 1.5, 50);
    z = zeros(length(u), length(v));

    for i = 1:length(u)
        for j = 1:length(v)
            z(i,j) = mapFeature(u(i), v(j)) * theta;
        end
    end

    z = z';
    contour(u, v, z, [0, 0], 'LineWidth', 2)
    hold off

end
```

Code Listing 2.1: My MATLAB Code



changes in the dicision boundary as varying *lambda*

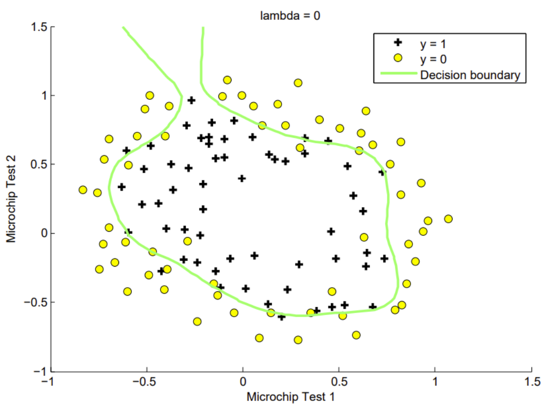


Figure 5: No regularization (Overfitting) ($\lambda = 0$)

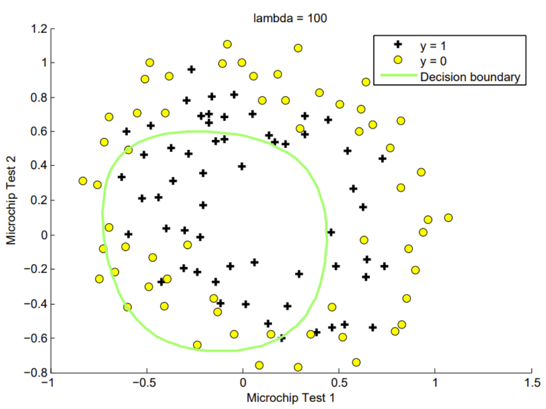


Figure 6: Too much regularization (Underfitting) ($\lambda = 100$)