Problem Set 2: Classification

To run and solve this assignment, one must have a working IPython Notebook installation. The easiest way to set it up for both Windows and Linux is to install Anaconda (https://www.continuum.io/downloads). Then save this file to your computer (use "Raw" link on gist\github), run Anaconda and choose this file in Anaconda's file explorer. Use `Python 3` version. Below statements assume that you have already followed these instructions. If you are new to Python or its scientific library, Numpy, there are some nice tutorials here (https://www.learnpython.org/) and here (http://www.scipy-lectures.org/).

To run code in a cell or to render Markdown (https://en.wikipedia.org/wiki/Markdown)+LaTeX (https://en.wikipedia.org/wiki/LaTeX) press `Ctr+Enter` or `[>|]` (like "play") button above. To edit any code or text cell [double]click on its content. To change cell type, choose "Markdown" or "Code" in the drop-down menu above.

If a certain output is given for some cells, that means that you are expected to get similar results in order to receive full points (small deviations are fine). For some parts we have already written the code for you. You should read it closely and understand what it does.

Total: 85 points (35 (Q1) + 20 (Q2) + 30 (Q3)).

1. Logistic Regression

In this part of the exercise, you will build a logistic regression model to predict whether a student gets admitted into a university.

Suppose that you are the administrator of a university department and you want to determine each applicant’s chance of admission based on their results on two exams. You have historical data from previous applicants in *ex2data1.txt* that you can use as a training set for logistic regression. For each training example, you have the applicant’s scores on two exams and the admissions decision.

Your task is to build a classification model that estimates an applicant’s probability of admission based on the scores from those two exams. This outline and code framework will guide you through the exercise.

1.1 Implementation

In [376]:
```
import sys
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
print('Tested with:')
print('Python', sys.version)
print({x.__name__: x.__version__ for x in [np, matplotlib]})
```

```
Tested with:
Python 3.7.0 (default, Aug 14 2018, 19:12:50) [MSC v.1900 32 bit (Intel)]
{'numpy': '1.15.1', 'matplotlib': '2.2.3'}
```

1.1.1 Visualizing the data

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. This first part of the code will load the data and display it on a 2-dimensional plot by calling the function plotData. The axes are the two exam scores, and the positive and negative examples are shown with different markers.

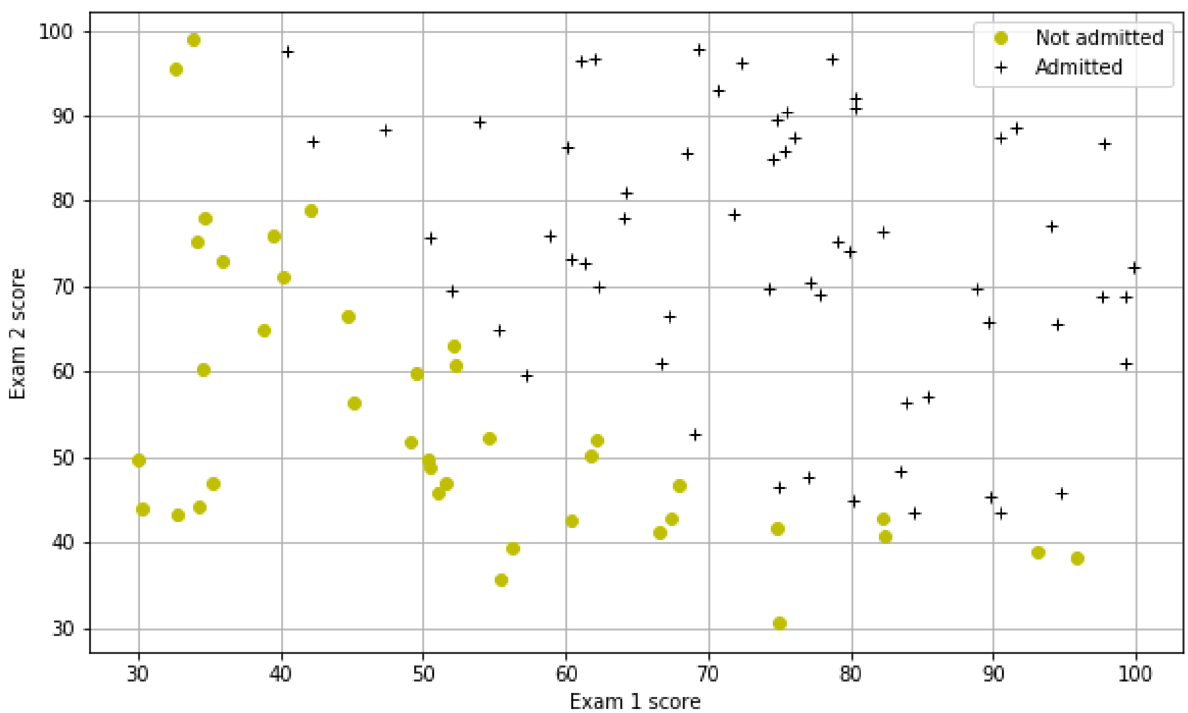
In [377]:
```
##############################################################################
#####
# Try to fit your code and comments into 80 charecters because
# - it is guaranteed to look as intened on any screen size
# - it encourages you to write "flater" logic that is easier to reason abou
t
# - it encourages you to decompose logic into comprehansible blocks.
#
# Try to avoid reassinging/mutating variables because when you encounter an
# unexplainable bugs (you will) it is easier to have the whole history
# of values to reason about.
#
# Using %debug magic to run pdb might be useful for debugging.
# It is just like gdb but for Python.
```

In [378]:
```
# it is good to isolate logical parts to avoid variables leaking into the
# global scope and messing up your logic later in weird ways

def read_classification_csv_data(fn, add_ones=False):
    # read comma separated data
    data = np.loadtxt(fn, delimiter=',')
    X_, y_ = data[:, :-1], data[:, -1, None]  # a fast way to keep last dim

    # printing statistics of data before working with it might have saved
    # hundreds hours of of my time, do not repeat my errors :)
    print(X_.shape, X_.min(), X_.max(), X_.dtype)
    print(y_.shape, y_.min(), y_.max(), y_.dtype)
    # aha, y is float! this is not what we expected
    # what might go wrong with further y == 0 checks?
    # A: floating point equality comparison, that's what!

    # insert the column of 1's into the "X" matrix (for bias)
    X = np.insert(X_, X_.shape[1], 1, axis=1) if add_ones else X_
    y = y_.astype(np.int32)
    return X, y

X_data, y_data = read_classification_csv_data('ex2data1.txt', add_ones=True
)
print(X_data.shape, X_data.min(), X_data.max(), X_data.dtype)
print(y_data.shape, y_data.min(), y_data.max(), y_data.dtype)
```

```
(100, 2) 30.05882244669796 99.82785779692128 float64
(100, 1) 0.0 1.0 float64
(100, 3) 1.0 99.82785779692128 float64
(100, 1) 0 1 int32
```

In [379]:

```
# how does the *X[y.ravel()==1, :2].T trick work?
# https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-li
sts

def plot_data(X, y, labels, markers, xlabel, ylabel, mSize = 6, figsize=(10
, 6), ax=None):
    if figsize is not None:
        plt.figure(figsize=figsize)

    ax = ax or plt.gca()
    for label_id, (label, marker) in enumerate(zip(labels, markers)):
        ax.plot(*X[y.ravel()==label_id, :2].T, marker, label=label, markers
ize = mSize)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    plt.legend()
    ax.grid(True)

student_plotting_spec = {
    'X': X_data,
    'y': y_data,
    'xlabel': 'Exam 1 score',
    'ylabel': 'Exam 2 score',
    'labels': ['Not admitted', 'Admitted'],
    'markers': ['yo', 'k+'],
    'figsize': (10, 6)
}

plot_data(**student_plotting_spec)
plt.show()
```



1.1.2 [5pts] Sigmoid function

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

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

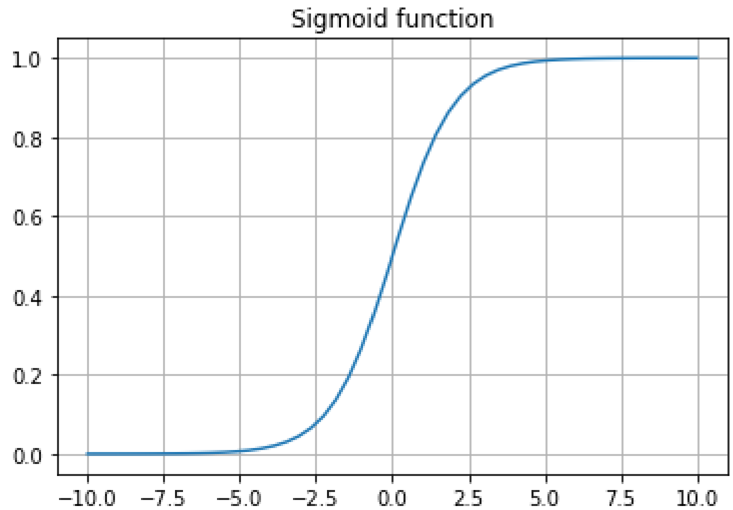
where function g is the sigmoid function. The sigmoid function is defined as:

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

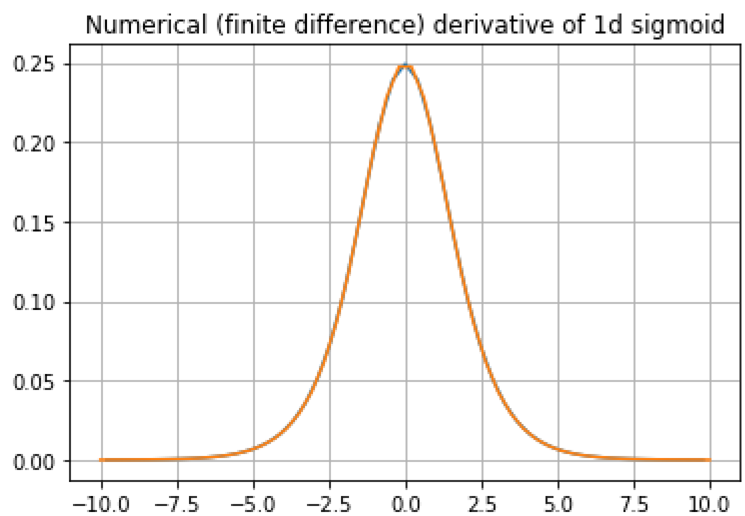
Your first step is to implement/find a sigmoid function so it can be called by the rest of your program. Your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid function on every element.

When you are finished, (a) plot the sigmoid function, and (b) test the function with a scalar, a vector, and a matrix. For scalar large positive values of x, the sigmoid should be close to 1, while for scalar large negative values, the sigmoid should be close to 0. Evaluating sigmoid(0) should give you exactly 0.5.

In [380]:
```
# check out scipy.special for great variaty of vectorized functions
# remember that sigmoid is the inverse of logit function
# maybe worth checking out scipy.special.logit first


def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def check_that_sigmoid_f(f):
    # don't use np.arange with float step because it works as
    # val_{i+1} = val_i + step while val_i < end
    # what might do wrong with float precision?
    x_test = np.linspace(-10, 10, 50)
    sigm_test = f(x_test)
    plt.plot(x_test, sigm_test)
    plt.title("Sigmoid function")
    plt.grid(True)
    plt.show()

    # why should analytical_diff almost== finite_diff for sigmoid?
    analytical_diff = sigm_test*(1-sigm_test)
    finite_step = x_test[1]-x_test[0]
    finite_diff = np.diff(sigm_test) / finite_step
    print(x_test.shape, finite_diff.shape)
    plt.plot(x_test[:-1]+finite_step/2, finite_diff)
    plt.plot(x_test, analytical_diff)
    plt.title("Numerical (finite difference) derivative of 1d sigmoid")
    plt.grid(True)
    plt.show()


check_that_sigmoid_f(sigmoid)
```


Sigmoid function
1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

```
(50,) (49,)
```


Numerical (finite difference) derivative of 1d sigmoid
0.25
0.20
0.15
0.10
0.05
0.00
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

1.1.3 [15pts] Cost function and gradient

Now you will implement the cost function and gradient for logistic regression. Complete the code in the functions *hyposesis_function* and *binary_logistic_loss* below to return the value of the hypothesis function and the cost, respectively. Recall that the cost function in logistic regression is

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

and the gradient of the cost is a vector of the same length as θ where the j^{th} element (for $j = 0, 1, \ldots, n$) is defined as follows:

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

where m is the number of points and n is the number of features. Note that while this gradient looks identical to the linear regression gradient, the formula is actually different because linear and logistic regression have different definitions of $h_\theta(x)$.

What should be the value of the loss for $\theta = \bar{0}$ regardless of input? Why? Make sure your code also outputs this value.

In [381]:
```
# we are trying to fit a function that would return a
# "probability of "

# hyposesis_function describes parametric family of functions that we are
# going to pick our "best fitting function" from. It is parameterized by
# real-valued vector theta, i.e. we are going to pick
#    h_best = argmin_{h \in H} logistic_loss_h(x, y, h)
# but because there exist a bijection between theta's and h's it is
# eqvivalent to choosing
#    theta_best = argmin_{theta \in H} logistic_loss_theta(x, y, theta)


def hyposesis_function(x, theta):

    theta = theta.reshape((-1, 1))
    hyp = np.dot(x, theta)
    return sigmoid(hyp)

# negative log likelihood of observing sequence of integer
# y's given probabilities y_pred's of each Bernoulli trial
# recommentation: convert both variables to float's
# or weird sign stuff might happen like -1*y != -y for uint8
# use np.mean and broadcasting
def binary_logistic_loss(y, y_pred):
    assert y_pred.shape == y.shape
    # or weird sign stuff happens! like -1*y != -y
    y, y_pred = y.astype(np.float64), y_pred.astype(np.float64)
    # When y_pred = 0, log(0) = -inf,
    # we could add a small constant to avoid this case
    CONSTANT = 0.000001
    y_pred = np.clip(y_pred, 0+CONSTANT, 1-CONSTANT)
    inv_y_pred = np.clip(1 - y_pred, 0+CONSTANT, 1-CONSTANT)
    cost = -(1 / y.shape[0]) * np.sum(y * np.log(y_pred) + (1 - y) * np.log
(inv_y_pred))
    return cost



def logistic_loss_theta_grad(x, y, h, theta):
    theta = theta.reshape((-1, 1))
    y_pred = h(x, theta)
    point_wise_grads = (y_pred - y)*x
    grad = np.mean(point_wise_grads, axis=0)[:, None]
    assert grad.shape == theta.shape
    return np.ndarray.flatten(grad)


def logistic_loss_theta(x, y, h, theta):
    return binary_logistic_loss(y, h(x, theta))
```

In [382]:
```
# Check that with theta as zeros, cost is about 0.693:
theta_init = np.zeros((X_data.shape[1], 1))
print(logistic_loss_theta(X_data, y_data, hyposesis_function, theta_init))
print(logistic_loss_theta_grad(X_data, y_data, hyposesis_function, theta_in
it))
```

```
0.6931471805599453
[-12.00921659 -11.26284221  -0.1        ]
```

1.1.4 Learning parameters using *scipy.optimize*

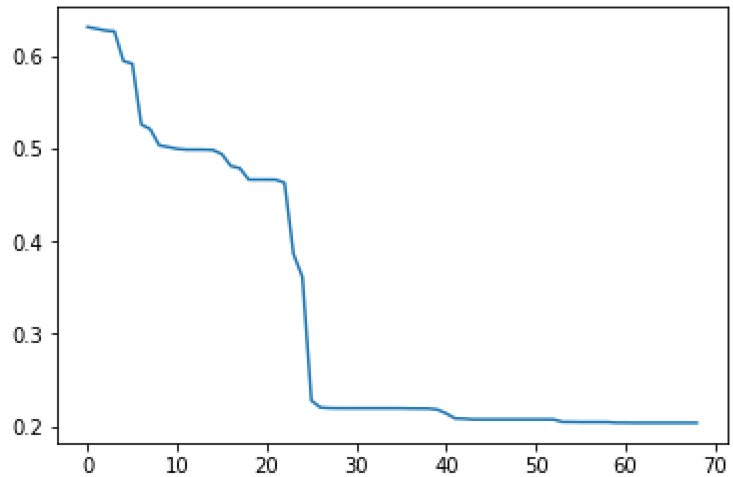
In the previous assignment, you found the optimal parameters of a linear regression model by implementing gradient descent. You wrote a cost function and calculated its gradient, then took a gradient descent step accordingly. This time, instead of taking gradient descent steps, you will use a scipy.optimize built-in function called *scipy.optimize.minimize*. In this case, we will use the [*conjugate gradient algorithm (https://docs.scipy.org/doc/scipy/reference/optimize.minimize-cg.html)*](https://docs.scipy.org/doc/scipy/reference/optimize.minimize-cg.html).

The final θ value will then be used to plot the decision boundary on the training data, as seen in the figure below.

In [383]:
```
import scipy.optimize
from functools import partial
```

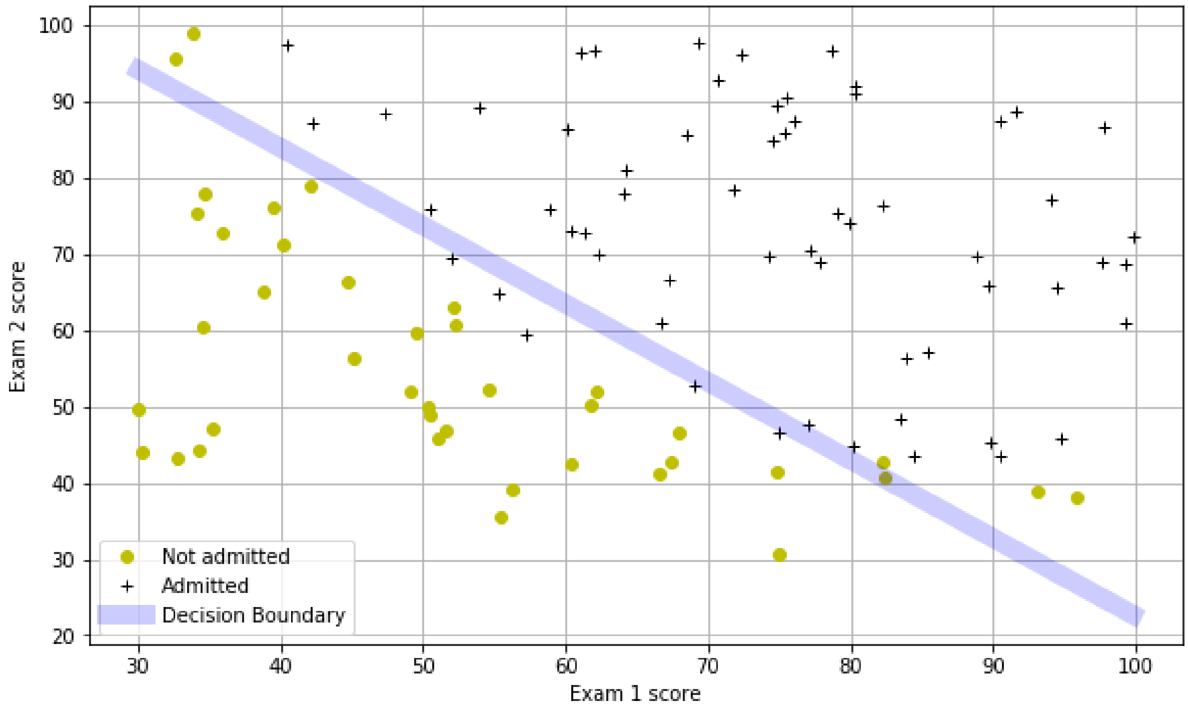
In [384]:
```
def optimize(theta_init, loss, loss_grad, max_iter=10000, print_every=1000,
optimizer_fn=None, show=False):
    theta = theta_init.copy()
    opt_args = {'x0': theta_init, 'fun': loss, 'jac': loss_grad, 'options':
{'maxiter': max_iter}}


    loss_curve = []
    def scipy_callback(theta):
        f_value = loss(theta)
        loss_curve.append(f_value)

    if optimizer_fn is None:
        optimizer_fn = partial(scipy.optimize.minimize, method='CG', callba
ck=scipy_callback)



    opt_result = optimizer_fn(**opt_args)

    if show:
        plt.plot(loss_curve)
        plt.show()

    return opt_result['x'].reshape((-1, 1)), opt_result['fun']
```

In [385]:
```
theta_init = np.zeros((3, 1))
loss = partial(logistic_loss_theta, X_data, y_data, hyposesis_function)
loss_grad = partial(logistic_loss_theta_grad, X_data, y_data, hyposesis_fun
ction)
theta, best_cost = optimize(theta_init, loss, loss_grad, show=True)
print(best_cost)
```




```
0.20349770186378272
```

In [386]:
```
# Plotting the decision boundary: two points, draw a line between
# Decision boundary occurs when h = 0, or when
# theta_0*x1 + theta_1*x2 + theta_2 = 0
# y=mx+b is replaced by x2 = (-1/theta1)(theta2 + theta0*x1)

line_xs = np.array([np.min(X_data[:,0]), np.max(X_data[:,0])])
line_ys = (-1./theta[1])*(theta[2] + theta[0]*line_xs)
plot_data(**student_plotting_spec)
plt.plot(line_xs, line_ys, 'b-', lw=10, alpha=0.2, label='Decision Boundar
y')
plt.legend()
plt.show()
```



1.1.5 [15pts] Evaluating logistic regression

After learning the parameters, you can use the model to predict whether a particular student will be admitted.

(a) [5 pts] Show that 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.

Another way to evaluate the quality of the parameters we have found is to see how well the learned model predicts on our training set.

(b) [10 pts] In this part, your task is to complete the code in *makePrediction*. The predict function will produce “1” or “0” predictions given a dataset and a learned parameter vector θ. After you have completed the code, the script below will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct. You should also see a Training Accuracy of 89.0.

In [387]:
```
# 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.
check_data = np.array([[45., 85., 1]])
print(check_data.shape)
print(hyposesis_function(check_data, theta))
```

```
(1, 3)
[[0.77627808]]
```

In [388]:
```
# use hyposesis function and broadcast compare operator
def predict(x, theta):
    return hyposesis_function(x, theta) > 0.5

def accuracy(x, y, theta):
    return np.mean(predict(x, theta) == y)

print(accuracy(X_data, y_data, theta))
```

```
0.89
```

2. Regularized logistic regression

In this part of the exercise, you will implement regularized logistic regression to predict whether microchips from a fabrication plant pass quality assurance (QA). During QA, each microchip goes through various tests to ensure it is functioning correctly. Suppose you are the product manager of the factory and you have the test results for some microchips on two different tests. From these two tests, you would like to determine whether the microchips should be accepted or rejected. To help you make the decision, you have a dataset of test results on past microchips in *ex2data2.txt*, from which you can build a logistic regression model.

2.1 Visualizing the data

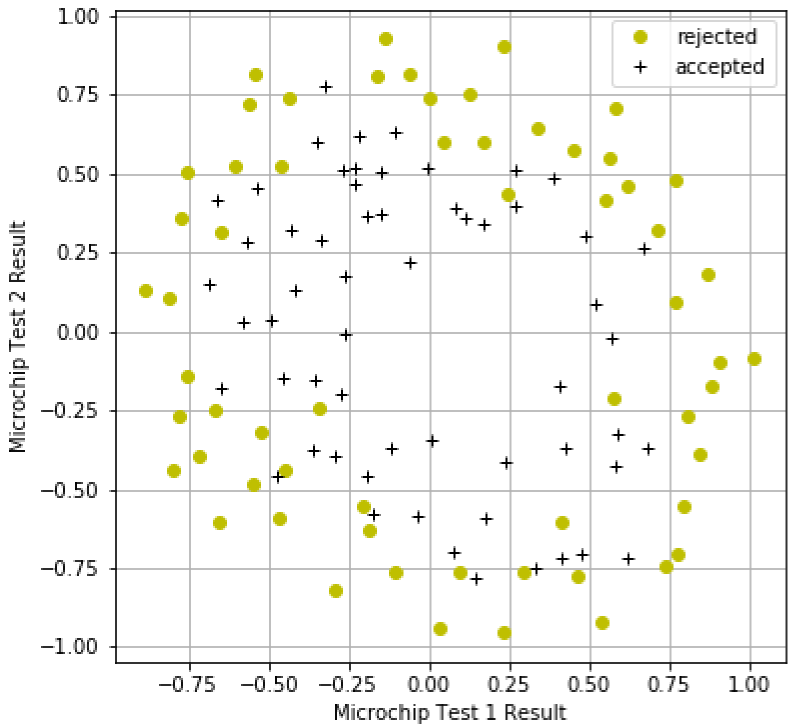
Similar to the previous parts of this exercise, plotData is used to generate the figure below, where the axes are the two test scores, and the positive (y = 1, accepted) and negative (y = 0, rejected) examples are shown with different markers.

The figure below shows that our dataset cannot be separated into positive and negative examples by a straight line. Therefore, a straightforward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

In [389]:
```
X_data_, y_data = read_classification_csv_data('ex2data2.txt')
X_data = X_data_ - X_data_.mean(axis=0)[None, :]
print(X_data.shape, X_data.min(), X_data.max(), X_data.dtype)
print(y_data.shape, y_data.min(), y_data.max(), y_data.dtype)
```

```
(118, 2) -0.83007 1.1089 float64
(118, 1) 0.0 1.0 float64
(118, 2) -0.9528415593220338 1.0161210915254237 float64
(118, 1) 0 1 int32
```

In [390]:
```
chip_plotting_spec = {
    'X': X_data,
    'y': y_data,
    'xlabel': 'Microchip Test 1 Result',
    'ylabel': 'Microchip Test 2 Result',
    'labels': ['rejected', 'accepted'],
    'markers': ['yo', 'k+'],
    'figsize': (6, 6)
}

plot_data(**chip_plotting_spec)
plt.show()
```



2.2 Nonlinear feature mapping

One way to fit the data better is to create more features from each data point. In *mapFeature* below, we will map the features into all polynomial terms of x_1 and x_2 up to the sixth power as follows:

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

As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 28-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. While the feature mapping allows us to build a more expressive classifier, it is also more susceptible to overfitting. In the next parts of the exercise, you will implement regularized logistic regression to fit the data and also see for yourself how regularization can help combat the overfitting problem.

Either finite dimentional (or even infinite-dimentional, as you would see in the SVM leacture and the corresponding home assingment) feature mappings are usually denoted by Φ and therefore our hyposesis is now that the Bernoulli probability of chip matfunctioning might be described as

$$p_i = \sigma(\Phi(x_i)^T \theta)$$

In [391]:
```
from itertools import combinations_with_replacement

def polynomial_feature_map(X_data, degree=20, show_me_ur_powers=False):
    assert len(X_data.shape) == 2
    group_size = X_data.shape[1]
    assert group_size == 2
    # hm.. how to get all ordered pairs (c, d) of non-negative ints
    # such that their sum is c + d <= dergee?
    # it is eqvivalent to getting all groups of integers (a, b) such that
    # 0 <= a <= b <= degree and definintg c = a, d = b - a
    # their sum is below degree, both are >= 0
    # then feature_i = (x_0 ^ c) * (x_1 ^ d)
    comb_iterator = combinations_with_replacement(range(degree+1), group_si
ze)
    not_quite_powers = np.array(list(comb_iterator))
    powers_bad_order = not_quite_powers.copy()
    powers_bad_order[:, 1] -= not_quite_powers[:, 0]
    # let's reoder them so that lower power monomials come first
    rising_power_idx = np.argsort(powers_bad_order.sum(axis=1))
    powers = powers_bad_order[rising_power_idx]
    if show_me_ur_powers is True:
        print(powers.T)
        print('total power per monomial', powers.sum(axis=1))
    X_with_powers = np.power(X_data[:, :, None], powers.T[None])
    # tu tu power rangers (with replacement)
    X_poly = np.prod(X_with_powers, axis=1)
    return X_poly

X_pf = polynomial_feature_map(X_data, show_me_ur_powers=True)
print(X_pf.shape)
```

```
[[ 0  0  1  0  2  1  0  2  3  1  4  2  1  3  0  5  3  0  2  4  1  0  6  2
   5  4  3  1  4  0  1  2  6  7  3  5  2  3  4  6  8  5  7  0  1  4  7  3
   6  9  8  0  2  1  5  9  3  5  7  8  4 10  6  1  0  2  0  9  1  4  5  2
  10  7  8  6 11  3  7  0  1  9 12  4  6  2  8 11 10  3  5  6 13  1  5  7
   9  8  4 12  2 10  0 11  3  0 12 11 14 10  9  7  6  8 13  4  2  3  1  5
   5  4  9  0 15 14  7  1 10  8  3  2 11  6 12 13  8  3 11  2 14  9 12 13
  10 16  4  0  6  5  1 15  7 12  5  2 13 11  6 15  3 10  7  0  1 16 14  8
   4  9 17  5 12  2  4  8 14 13 16 11  9  6 15  0  1 10  3 18  7 17 15 13
  14 17 19 18 16  2 12  3  1  4  5  0  7  8  6  9 10 11 11 18  3  1 17 10
   4 16 13  5  0 19 15 12  6  9  7 14  2  8 20]
 [ 0  1  0  2  0  1  3  1  0  2  0  2  3  1  4  0  2  5  3  1  4  6  0  4
   1  2  3  5  3  7  6  5  1  0  4  2  6  5  4  2  0  3  1  8  7  5  2  6
   3  0  1  9  7  8  4  1  7  5  3  2  6  0  4  9 10  8 11  2 10  7  6  9
   1  4  3  5  0  8  5 12 11  3  0  8  6 10  4  1  2  9  7  7  0 12  8  6
   4  5  9  1 11  3 13  2 10 14  2  3  0  4  5  7  8  6  1 10 12 11 13  9
  10 11  6 15  0  1  8 14  5  7 12 13  4  9  3  2  8 13  5 14  2  7  4  3
   6  0 12 16 10 11 15  1  9  5 12 15  4  6 11  2 14  7 10 17 16  1  3  9
  13  8  0 13  6 16 14 10  4  5  2  7  9 12  3 18 17  8 15  0 11  1  4  6
   5  2  0  1  3 17  7 16 18 15 14 19 12 11 13 10  9  8  9  2 17 19  3 10
  16  4  7 15 20  1  5  8 14 11 13  6 18 12  0]]
total power per monomial [ 0  1  1  2  2  2  3  3  3  3  4  4  4  4  4  5
5  5  5  5  5  6  6  6
  6  6  6  6  7  7  7  7  7  7  7  7  8  8  8  8  8  8  8  8  8  9  9  9
  9  9  9  9  9  9  9 10 10 10 10 10 10 10 10 10 10 10 11 11 11 11 11 11
 11 11 11 11 11 11 12 12 12 12 12 12 12 12 12 12 12 12 12 13 13 13 13 13
 13 13 13 13 13 13 13 13 13 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 16 16 16 16 16 16 16 16
 16 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17
 17 17 17 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 19 19
 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 20 20 20 20 20 20
 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20]
(118, 231)
```

2.3 Cost function and gradient

Now you will implement code to compute the cost function and gradient for regularized logistic regression. Recall that the regularized cost function in logistic regression is:

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

Note that you should not regularize the parameter θ_0 (Why not? Think about why that would be a bad idea).

The gradient of the cost function is a vector where the j element is defined as follows (you should understand how to obtain this expression):

$$\frac{\partial J(\theta)}{\partial \theta_0} \ = \ \frac{1}{m} \ \sum_{i=1}^{m} \ (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \qquad \text{for} \quad j = 0$$

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

2.3.1 [10pts] Implementing regularized logistic regression

Re-implement computeCost with regularization.

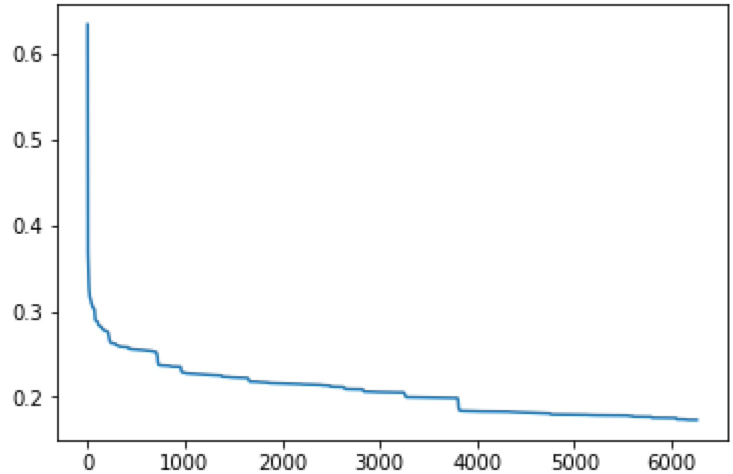
In [392]:
```
# Cost function, default lambda (regularization) 0
def logistic_loss_theta_w_reg(x, y, h, theta, lambda_=0.0):

    m = y.shape[0]
    thetaNew = np.copy(theta)
    thetaNew = thetaNew[2:]
    pred = h(x, theta)
    CONSTANT = 0.000001
    pred = np.clip(pred, 0+CONSTANT, 1-CONSTANT)
    return ((-1/m)*np.sum(y*np.log(pred)+(1-y)*np.log(1-pred))) + (lambda_/
(2*m))*np.sum(thetaNew**2)

def logistic_loss_theta_w_reg_grad(x, y, h, theta, lambda_=0.0):
    m = y.shape[0]
    theta2 = np.copy(theta)
    theta2[0] = 0
    grad = (1/m) * np.sum((h(x, theta) - y) * x, axis=0) + (lambda_/m)
    return grad
```

Once you are done, you will call your cost function using the initial value of θ (initialized to all zeros). You should see that the cost is about 0.693.

In [393]:
```
theta_init = np.zeros((X_pf.shape[1], 1))
print(logistic_loss_theta_w_reg(X_pf, y_data, hyposesis_function, theta_ini
t))
print(logistic_loss_theta_w_reg_grad(X_pf, y_data, hyposesis_function, thet
a_init))

loss = partial(logistic_loss_theta_w_reg, X_pf, y_data, hyposesis_function)
loss_grad = partial(logistic_loss_theta_w_reg_grad, X_pf, y_data, hyposesis
_function)
theta, best_cost = optimize(theta_init, loss, loss_grad, max_iter=10000, pr
int_every=0, show=True)
print('best loss', best_cost)
print('best acc', accuracy(X_pf, y_data, theta))
```

```
0.6931471805599454
[ 8.47457627e-03 -1.47393694e-03  1.83238652e-02  3.79204875e-02
  4.83116870e-02  8.14194228e-03  2.74281128e-03 -2.78183863e-03
  1.02502549e-02  2.53327158e-03  3.62206863e-02  1.14745376e-02
 -5.13006494e-04 -7.37665567e-04  2.96926570e-02  9.66227378e-03
  1.26710268e-03  5.83282025e-04 -7.50311428e-04 -2.66858278e-03
  1.96894912e-03  2.22460292e-02  2.61849067e-02  5.14792073e-03
 -1.80202003e-03  5.56307799e-03 -1.21167516e-03 -2.15775754e-03
 -7.77390798e-04 -9.97158003e-04  1.71369098e-03 -6.27268614e-04
 -2.22430409e-03  9.08105468e-03  8.49610910e-04  1.01196534e-03
  2.91086360e-03 -1.05308072e-03  2.03649298e-03  3.01185527e-03
  1.95553205e-02 -8.02388158e-04 -1.74538564e-03  1.70850907e-02
 -2.24318631e-03 -4.29158120e-04  7.52146991e-04  5.96787640e-04
 -6.12485717e-04  8.44926264e-03 -1.72286617e-03 -1.85306005e-03
 -5.83990046e-04  1.55326398e-03  5.26695294e-04 -1.48962270e-03
 -7.85026484e-04 -5.21056659e-04 -5.06525809e-04  1.75113347e-03
  9.94803149e-04  1.52391383e-02  9.72566668e-04 -2.00909851e-03
  1.35100928e-02  1.88656843e-03 -2.25032182e-03  5.34461772e-04
  1.41933459e-03 -2.83959262e-04  3.08317958e-04 -5.37087646e-04
 -1.32506623e-03  3.34412182e-04 -4.25516303e-04 -2.90240458e-04
  7.90050084e-03  4.44349403e-04 -2.76713749e-04  1.09436748e-02
 -1.73758499e-03 -3.21363567e-04  1.24079315e-02  5.71999634e-04
  4.32732207e-04  1.33553760e-03  5.12654909e-04 -1.23938660e-03
  1.06907659e-03 -5.80663410e-04 -3.33795768e-04 -1.60990627e-04
  7.47484721e-03  1.29042510e-03  1.93080059e-04  1.76990626e-04
  2.09326473e-04 -1.86973776e-04 -2.07021597e-04 -1.04620838e-03
 -4.86587418e-04 -2.80191940e-04 -2.38498748e-03  3.75865346e-04
  3.50244399e-04  9.03209550e-03  6.77123643e-04 -2.04647384e-04
  1.05371383e-02  2.87124296e-04 -1.56428071e-04 -1.58474755e-04
  2.27964051e-04  2.12965399e-04 -1.04058930e-03  3.66985247e-04
  1.00233823e-03 -4.39251806e-04 -1.49331666e-03 -2.20523830e-04
  1.30451628e-04 -1.61176751e-04  1.04119828e-04 -2.37488815e-03
  7.16946632e-03 -8.61198695e-04  9.95414889e-05  1.16421308e-03
 -1.16709110e-04 -9.51174611e-05  2.87275083e-04 -4.35866834e-04
  1.29923762e-04 -9.85544357e-05 -1.80712606e-04  2.66827110e-04
  1.04689809e-04 -3.41331526e-04 -9.15817603e-05  7.82822860e-04
  4.43095889e-04 -8.27204171e-05  1.66859455e-04 -1.30791364e-04
  1.13406018e-04  9.29796292e-03  2.54559409e-04  7.56219770e-03
  1.34573718e-04 -1.52363835e-04 -1.28640877e-03 -8.94740987e-04
 -9.49435171e-05 -7.17147424e-05  9.45717643e-05 -3.87371653e-04
  8.04167511e-05  6.17684344e-05 -6.57250165e-05  1.93699340e-04
  2.41471250e-04 -5.65355937e-05  5.97572413e-05 -2.28744025e-03
  1.04308827e-03 -7.42326893e-04 -1.16041532e-04 -5.25177896e-05
 -1.30965247e-04  5.49236357e-05  6.96814247e-03 -1.10125780e-04
  6.33934609e-05  6.28673866e-04  1.86987655e-04  5.73885016e-05
  9.92359449e-05 -5.45334072e-05  2.99854418e-04 -4.58889618e-05
 -4.58753639e-05  8.65567605e-05 -8.41661983e-05  6.40173082e-03
 -1.11333815e-03  5.30023157e-05 -2.71775416e-04  8.48180067e-03
 -5.99572151e-05 -7.92108723e-04  4.98326029e-05  3.67822386e-05
 -4.37812881e-05  1.45169685e-04  6.85268784e-03 -6.68120055e-04
 -7.49083298e-05 -3.42482223e-04 -3.35943102e-05  2.05842933e-04
  9.29517573e-04 -1.09218531e-04  7.25918800e-05 -2.16073377e-03
  3.84950427e-05 -3.11509638e-05 -4.72248742e-05  3.04852627e-05
 -2.94487967e-05  3.13101492e-05 -2.41290980e-05  2.10657777e-04
 -2.20772412e-04 -9.68124950e-04 -5.48182719e-05  2.74189830e-05
  1.43302573e-04  5.99640102e-05 -2.63125037e-05 -8.28445778e-05
  5.46567327e-03 -7.22028416e-04 -3.27481656e-05  2.86020127e-05
```

```
 5.96304276e-05 -2.67109736e-05 -4.00056524e-05  3.64708123e-05
 5.15053246e-04  3.41810853e-05  7.95432475e-03]
```




```
best loss 0.17336590586483436
best acc 0.9322033898305084
```

2.4 Plotting the decision boundary

To help you visualize the model learned by this classifier, we have provided the function *plotBoundary* which plots the (non-linear) decision boundary that separates the positive and negative examples.

In [394]:
```
def plot_boundary(theta, ax=None):
    """
    Function to plot the decision boundary for arbitrary theta,
    X, y, lambda value
    Inside of this function is feature mapping, and the minimiza
    tion routine.
    It works by making a grid of x1 ("xvals") and x2 ("yvals") p
    oints,
    And for each, computing whether the hypothesis classifies th
    at point as
    True or False. Then, a contour is drawn with a built-in pypl
    ot function.
    """
    ax = ax or plt.gca()
    x_range = np.linspace(-1,1.5,50)
    y_range = np.linspace(-1,1.5,50)
    xx, yy = np.meshgrid(x_range, y_range)
    X_fake = np.stack([xx, yy]).reshape(2, -1).T
    X_fake_fm = polynomial_feature_map(X_fake)
    y_pred_fake = hyposesis_function(X_fake_fm, theta)
    return ax.contour( x_range, y_range, y_pred_fake.reshape(50,50).T, [0.5
])
```

2.4.1 [10pts] Plot Decision Boundaries

(a) [2 pts] Use *plotBoundary* to obtain four subplots of the decision boundary for the following values of the regularization parameter: $\lambda \ = \ 0, 1, 5, 10$

(b) [2 pts] Comment on which plots are overfitting and which plots are underfitting.

Answer: lambda = 0 is over fillting. while lambda = 10 is underfitting

(c) [2 pts] Which is the model with the highest bias? The highest variance?

Answer: lambda = 0 is the highest variance, while lambda = 10 is the highest bias

(d) [2 pts] What is another way to detect overfitting?

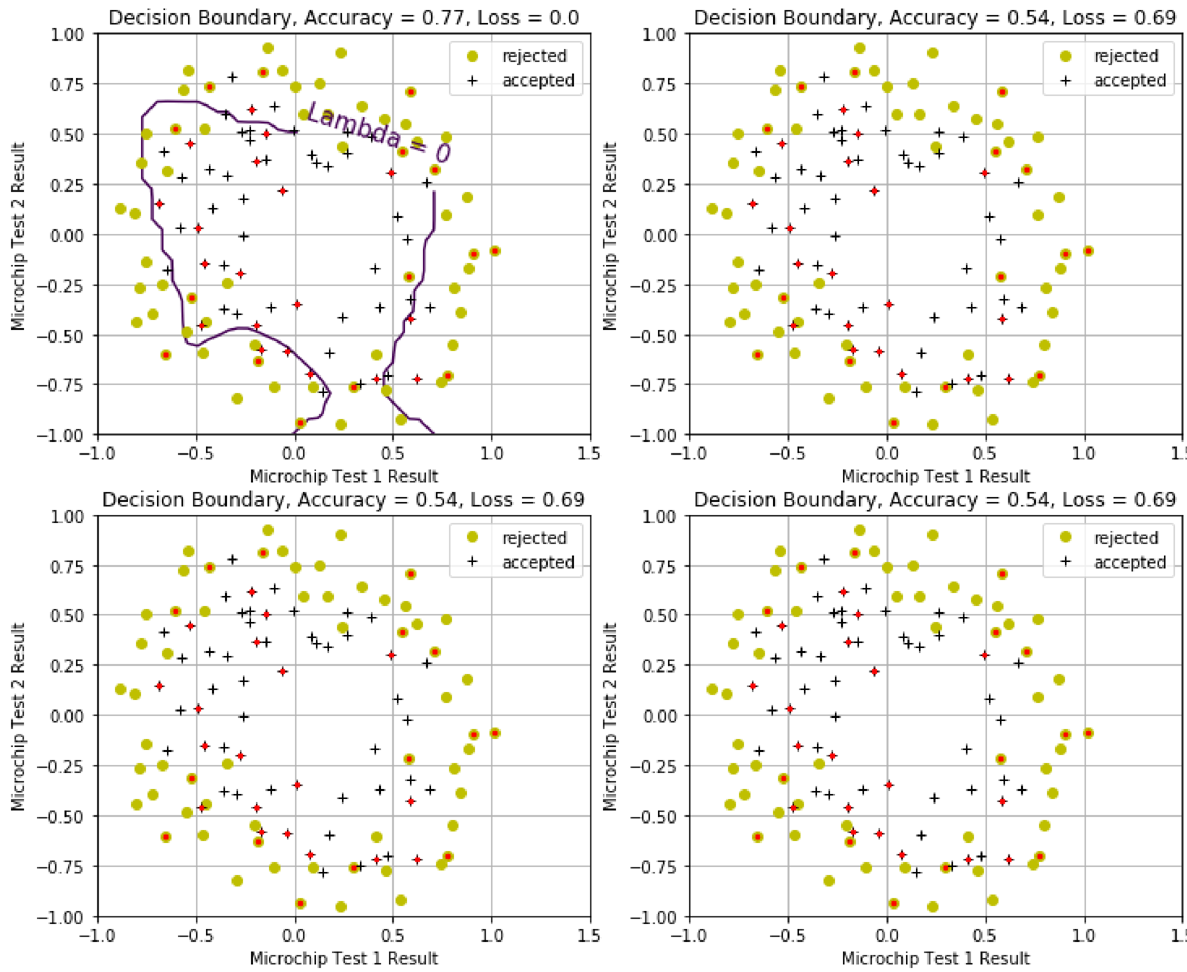
Answer: We can plot the error for the training set and the cross validation set as a function of training set size and check the vertical gap between two curves

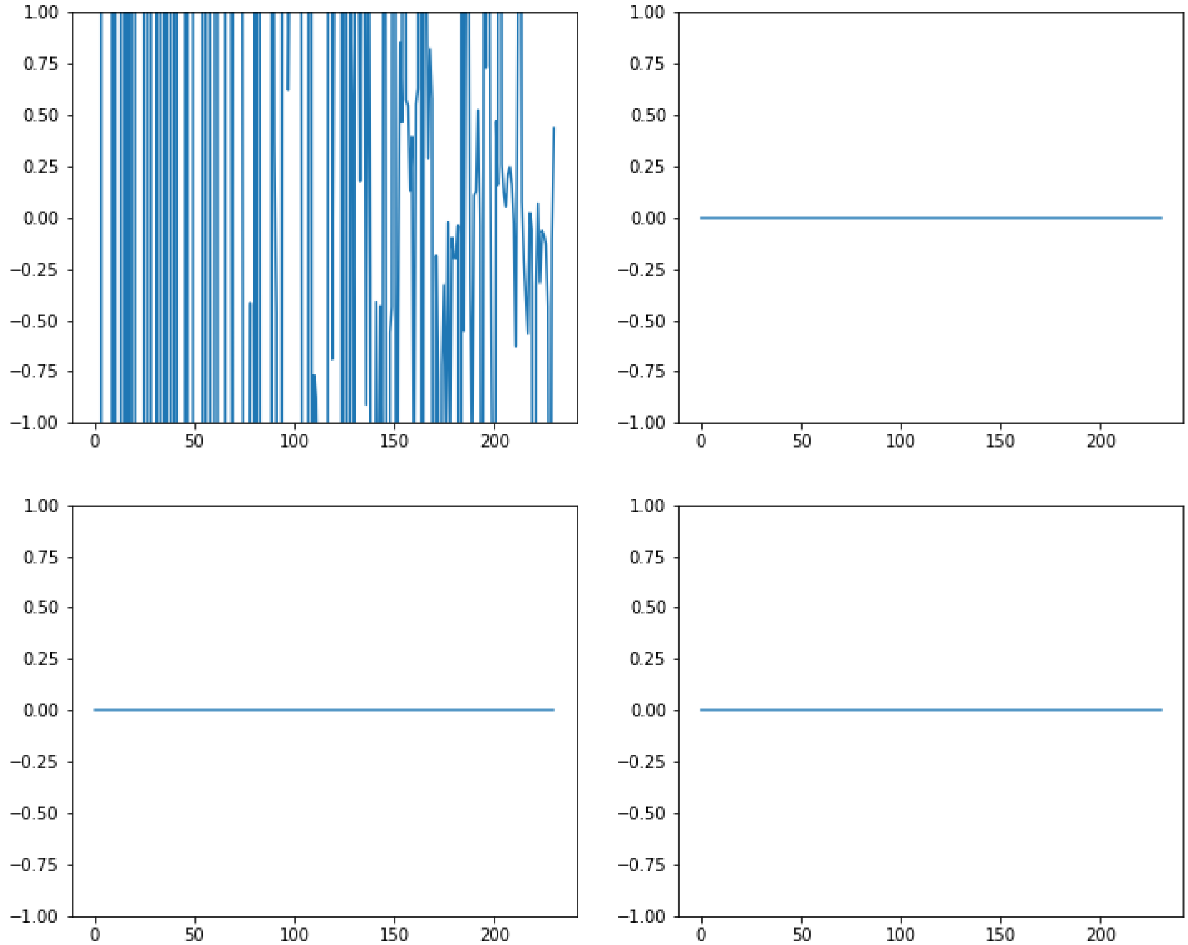
(e) [2 pts] Considering that later components of theta correspond to higher powers of monomials, plot values of theta and commend on effects of regularization

In [395]:
```
# (a) Build a figure showing contours for various values of regularization
 parameter, lambda

np.random.seed(2)
train_idx_mask = np.random.rand(X_pf.shape[0]) < 0.3
X_data_train = X_data[train_idx_mask]
X_pf_train, y_train = X_pf[train_idx_mask], y_data[train_idx_mask]
X_pf_test, y_test = X_pf[~train_idx_mask], y_data[~train_idx_mask]
print([x.shape for x in (X_pf_train, y_train, X_pf_test, y_test)])

def silent_optimize_w_lambda(lambda_):
    theta_init = np.zeros((X_pf.shape[1], 1))
    data = (X_pf_train, y_train, hyposesis_function)
    loss = partial(logistic_loss_theta_w_reg, *data, lambda_=lambda_)
    loss_grad = partial(logistic_loss_theta_w_reg_grad, *data, lambda_=lamb
da_)
    theta, final_loss = optimize(theta_init, loss, loss_grad, optimizer_fn=
None, max_iter=1000, print_every=0, show=False)
    return theta, final_loss

thetas = []
plt.figure(figsize=(12,10))

# wow, I mutates an object used in the scope of another function (plot_dat
a)
# don't do that! it is really hard to debug later

chip_plotting_spec['figsize'] = None

chip1_plotting_spec = {
    'X': X_data_train,
    'y': y_train,
    'xlabel': 'Microchip Test 1 Result',
    'ylabel': 'Microchip Test 2 Result',
    'labels': ['',''],
    'markers': ['ro', 'ro'],
    'mSize': 2.5,
    'figsize': None
}

# you might find following lines useful:
#
# cnt_fmt = {0.5: 'Lambda = %d' % lambda_}
# ax.clabel(cnt, inline=1, fontsize=15, fmt=cnt_fmt)
#
# red dots indicate training samples

for id_, lambda_ in enumerate([0, 1, 5, 10]):
    ax = plt.subplot(2, 2, id_+1)
    theta, loss = silent_optimize_w_lambda(lambda_)
    boundary = plot_boundary(theta, ax)
    cnt_fmt = {0.5: 'Lambda = %d' % lambda_}
    ax.clabel(boundary, inline=1, fontsize=15, fmt=cnt_fmt)
    ax.set_ylim([-1, 1])
    plot_data(**chip_plotting_spec)
    plot_data(**chip1_plotting_spec)
```

```
    num = accuracy(X_pf_test, y_test, theta)
    title = "Decision Boundary, Accuracy = " + str(round(num, 2)) + ", Loss
= " + str(round(loss, 2))
    plt.title(title)

plt.show()

# (e) [2 pts] Considering that later components of theta correspond to high
er powers
# of monomials, plot values of theta and commend on effects of regularizati
on

plt.figure(figsize=(12,10))

for id_, lambda_ in enumerate([0,1,5,10]):
    ax = plt.subplot(2, 2, id_+ 1, sharey=ax)
    theta, final_loss = silent_optimize_w_lambda(lambda_)
    ax.plot(np.arange(0,len(theta.ravel()),1),theta.ravel())
    ax.set_ylim([-1,1])

plt.show()
```

```
[(34, 231), (34, 1), (84, 231), (84, 1)]

C:\Users\timpc\Anaconda3\lib\site-packages\matplotlib\contour.py:1230: User
Warning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
C:\Users\timpc\Anaconda3\lib\site-packages\matplotlib\contour.py:1230: User
Warning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
C:\Users\timpc\Anaconda3\lib\site-packages\matplotlib\contour.py:1230: User
Warning: No contour levels were found within the data range.
  warnings.warn("No contour levels were found"
```





3. Written part

These problems are extremely important preparation for the exam. Submit solutions to each problem by filling the markdown cells below.

3.1 [10pts] Maximum likelihood for Logistic Regression

Showing all steps, derive the LR cost function using maximum likelihood. Assume that the probability of y given x is described by:

$$P(\ y = 1 \mid x\ ;\ \theta\ ) = h_{\theta}(x)$$

$$P(\ y = 0 \mid x\ ;\ \theta\ ) = 1 - h_{\theta}(x)$$

$\boxed{3.1}$

Two probability functions can be combined

$$P(y_i) = h_\theta(x_i)^{y_i}(1-h_\theta(x_i))^{1-y_i}$$

The likelihood of the entire dataset X is the product of the individual data point likelihoods

$$P(X|y) = \prod_{i=1}^{m} P(x_i|y_i) = \prod_{i=1}^{m} h_\theta(x_i)^{y_i}(1-h_\theta(x_i))^{1-y_i}$$

Now, we take the natural log, as maximizing the likelihood is the same as maximizing the log likelihood.

$$L(\theta) = \log(P(X|y)) = \sum_{i=1}^{m} y_i \log(h_\theta(x_i)) + (1-y_i)\log(1-h_\theta(x_i))$$

So, our cost function for LR comes out to be:

$$J(\theta) = -\frac{1}{m}L(\theta)$$

We can split the above into two cost funtions depending on the value of y_i

$$J(h_\theta(x_i), y_i) = -\log(1-(h_\theta(x_i)), \text{ if } y_i = 0$$

$$J(h_\theta(x_i), y_i) = -\log(h_\theta(x_i)), \text{ if } y_i = 1$$

3.2 [10pts] Logistic Regression Classification with Label Noise

Suppose you are building a logistic regression classifier for images of dogs, represented by a feature vector x, into one of two categories $y \in \{0, 1\}$, where 0 is "terrier" and 1 is "husky." You decide to use the logistic regression model $p(y = 1 \mid x) = h_\theta(x) = \sigma(\theta^T x)$. You collected an image dataset **D** $= \{x^{(i)}, t^{(i)}\}$, however, you were very tired and made some mistakes in assigning labels $t^{(i)}$. You estimate that you were correct in about τ fraction of all cases.

(a) Write down the equation for the posterior probability $p(t = 1 \mid x)$ of the label being 1 for some point x, in terms of the probability of the true class, $p(y = 1 \mid x)$.

(b) Derive the modified cost function in terms of $\theta, x^{(i)}, t^{(i)}$ and τ.

$\boxed{3.2}$

a) $p(t=1|x) = \tau p(t_i=1|x) + (1-\tau)\, p(t_i=0|X)$

b) $p(t_i=1|x) = h_\theta(x_i)$

$p(t_i=0|x) = 1 - h_\theta(\cancel{x}_i)$

$p(t=1|x) = \tau h_\theta(\cancel{x}_i) + (1-\tau)(1-h_\theta(x_i))$

$p(t=0|x) = \tau(1-h_\theta(x_i)) + (1-\tau)\, h_\theta(x_i)$

$p(t|x) = \tau\, h_\theta(x_i)^{t_i} \cdot (1-h_\theta(x_i))^{t_i} + (1-\tau)\, h_\theta(x_i)^{(1-t_i)} \cdot (1-h_\theta(x_i))^{(1-t_i)}$

If we take a negative log, we then get:

$$E(x_i) = -\frac{1}{N} \sum_{i=1}^{N} \Big[t_i \big[\ln(\tau h_\theta(x_i)) + \ln(\tau(1-h_\theta(x_i))) \big] + (1-t_i)\big[\ln((1-\tau) h_\theta(x_i)) + \ln((1-\tau)(1-h_\theta(x_i))) \big] \Big]$$

3.3 [10pts] Cross-entropy loss for multiclass classification

This problem asks you to derive the cross-entropy loss for a multiclass classification problem using maximum likelihood. Consider the multiclass classification problem in which each input is assigned to one of K mutually exclusive classes. The binary target variables y_k ∈ {0, 1} have a "one-hot" coding scheme, where the value is 1 for the indicated class and 0 for all others. Assume that we can interpret the network outputs as $h_k(x, \theta) = p(y_k = 1|x)$, or the probability of the kth class.

Show that the maximum likelihood estimate of the parameters θ can be obtained by minimizing the multiclass *cross-entropy* loss function

$L(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\log(h_k(x_i, \theta))$

where N is the number of examples $\{x_i, y_i\}$.

$\boxed{3.3}$ Consider $p(y_k=1) = h_k(x, \theta)$
for a given point y_{ik} we can express its likelihood as:

$$\prod_{k=1}^{K} \left(p(y_{ik}=1 \mid x_i) \right)^{y_{ik}}$$

, now we consider the likelihood over all i points given by

$$P(D, \theta) = \prod_{i=1}^{N} \prod_{k=1}^{K} h_k(x_i, \theta)^{y_{ik}}$$

we want to find θ, such that $\underset{\theta}{\operatorname{argmax}} (P(D,\theta))$

First we find the log likelihood

$$l(D,\theta) = \log \prod_{i=1}^{N} \prod_{k=1}^{K} (h_k(x_i,\theta))^{y_{ik}}$$

we can simplify this with the distributive log property

$$l(D,\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(h_k(x_i,\theta))$$

so, we get the cost function:

$$J(\theta) = -\frac{1}{m} l(D,\theta) = -\frac{1}{m} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log(h_k(x_i,\theta))$$

So, finding $\operatorname{argmax}(l(D,\theta))$ is the same as finding $\underset{\theta}{\operatorname{argmin}}(J(\theta))$

and $J(\theta)$ is the multiclass cross-entropy loss function