COMPSCI 371D Homework 6

Part 1: Automatic Differentiation

Automatic Differentiation Basics

In [1]:
```
_names = {}

def reset_auto_diff(name=None):
    global _names
    if name is None:
        _names = {}
    else:
        _names.pop(name, None)
```

In [2]:
```
class Node:
    def __init__(self, value, gradient=None, name=None, variables=None):
        self.value = value
        self.gradient = gradient
        self.name = name
        self.variables = variables
```

In [3]:
```
class Internal(Node):
    def __init__(self, value, gradient, variables):
        super(Internal, self).__init__(value, gradient=gradient, name=None,
                                       variables=variables)

    def __str__(self):
        grad_string = 'Gradient wrt {}:\n{}'.format(self.variables, self.gradient)
        return '{}\n{}'.format(self.value, grad_string)
```

In [4]:
```
class Variable(Node):
    def __init__(self, value, name):
        assert name not in _names, 'Different independent variables must have diffe
rent name'
        assert isinstance(value, int) or isinstance(value, float), 'Variables must
be scalars'
        value = float(value)
        _names[name] = name
        gradient = [1.0]
        super(Variable, self).__init__(value, gradient=gradient, name=name, variabl
es=[name])

    def __str__(self):
        return '{}:\n{}'.format(self.name, self.value)
```

In [5]:
```
def print_variables(*args, **keywords):
    try:
        print(keywords['title'])
    except KeyError:
        pass
    print(*args, sep='\n\n', end='\n\n')
```

In [6]:
```python
def scale_list(sequence, factor):
    return [factor * item for item in sequence]
```

A Botched Implementation of `times`

In [7]:
```python
def bad_times(a, b):
    assert isinstance(a, Node) and isinstance(b, Node)
    value = a.value * b.value
    a_gradient = scale_list(a.gradient, b.value)
    b_gradient = scale_list(b.gradient, a.value)
    # The following two + are list concatenation operators
    gradient = a_gradient + b_gradient
    variables = a.variables + b.variables
    return Internal(value, gradient, variables)
```

In [8]:
```python
reset_auto_diff()

u = Variable(2.0, 'u')
v = Variable(4.0, 'v')
print_variables(u, v)

p = bad_times(u, v)
print_variables(p, title='product')

s = bad_times(u, u)
print_variables(s, title='square')
```

```
u:
2.0

v:
4.0

product
8.0
Gradient wrt ['u', 'v']:
[4.0, 2.0]

square
4.0
Gradient wrt ['u', 'u']:
[2.0, 2.0]
```

Problem 1.1

Solution

In [9]:
```
def merge(*partials):
    derivatives = {}
    for p in partials:
        values = p[0]
        keys = p[1]
        for i in range(len(keys)):
            v = values[i]
            k = keys[i]
            if k in derivatives.keys():
                v = v + derivatives[k]
            derivatives.update({k: v})
    return list(derivatives.values()), list(derivatives.keys())
```

In [10]:
```
def times(a, b):
    assert isinstance(a, Node) and isinstance(b, Node)
    value = a.value * b.value
    a_gradient = scale_list(a.gradient, b.value)
    b_gradient = scale_list(b.gradient, a.value)
    gradient, variables = merge((a_gradient, a.variables), (b_gradient, b.variable
s))
    return Internal(value, gradient, variables)
```

In [11]:
```
reset_auto_diff()

u = Variable(2.0, 'u')
v = Variable(4.0, 'v')
print_variables(u, v)

p = times(u, v)
print_variables(p, title='product')

s = times(u, u)
print_variables(s, title='square')
```

```
u:
2.0

v:
4.0

product
8.0
Gradient wrt ['u', 'v']:
[4.0, 2.0]

square
4.0
Gradient wrt ['u']:
[4.0]
```

Problem 1.2

Solution

In [12]:
```
def plus(a, b):
    assert isinstance(a, Node) and isinstance(b, Node)
    value = a.value + b.value
    a_gradient = scale_list(a.gradient, 1)
    b_gradient = scale_list(b.gradient, 1)
    gradient, variables = merge((a_gradient, a.variables), (b_gradient, b.variable
s))
    return Internal(value, gradient, variables)
```

In [13]:
```
reset_auto_diff()

u = Variable(2.0, 'u')
v = Variable(4.0, 'v')
print_variables(u, v)

z = plus(u, v)
print_variables(z, title='sum')

twice = plus(u, u)
print_variables(twice, title='twice')
```

```
u:
2.0

v:
4.0

sum
6.0
Gradient wrt ['u', 'v']:
[1.0, 1.0]

twice
4.0
Gradient wrt ['u']:
[2.0]
```

Problem 1.3

Solution

In [14]:
```
import numpy as np
import matplotlib.pyplot as plt
import math as math
%matplotlib inline
```

In [15]:
```
def cosine(x):
    assert isinstance(x, Node)
    value = math.cos(x.value)
    gradient = scale_list(x.gradient, -math.sin(x.value))
    return Internal(value, gradient, x.variables)
```

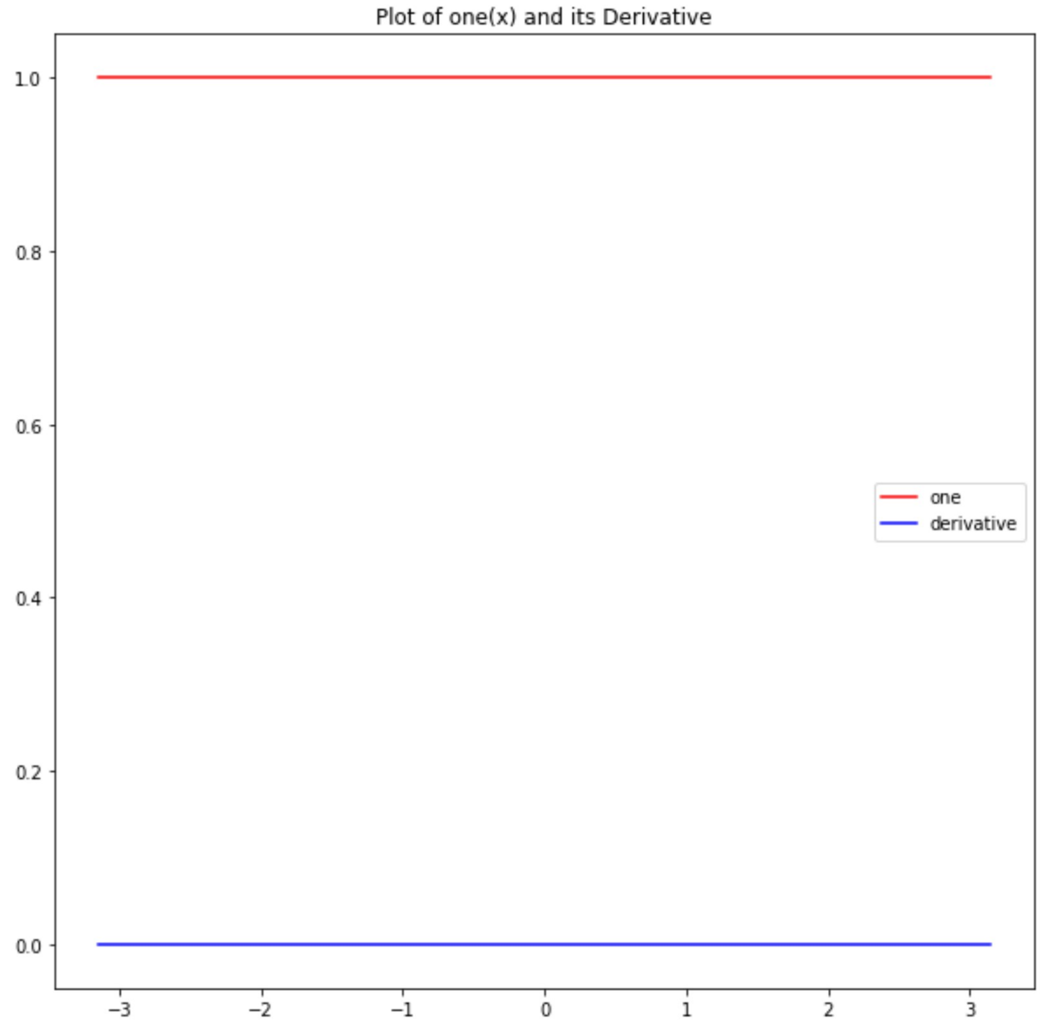
In [16]:
```
def sine(x):
    assert isinstance(x, Node)
    value = math.sin(x.value)
    gradient = scale_list(x.gradient, math.cos(x.value))
    return Internal(value, gradient, x.variables)
```

In [17]:
```
def one(x):
    assert isinstance(x, Node)
    return plus(times(cosine(x),cosine(x)),times(sine(x),sine(x)))
```

In [18]:
```
def derivative(a, name):
    assert isinstance(a, Node)
    if name not in a.variables:
        return 0
    index = a.variables.index(name)
    return a.gradient[index]
```

In [19]:
```
reset_auto_diff()
domain = np.linspace(-math.pi, math.pi, endpoint = True)
one_range = []
deriv_range = []
for d in domain:
    v = Variable(d, str(d))
    result = one(v)
    one_range.append(result.value)
    deriv_range.append(derivative(result, v.variables))
```

In [20]:
```
plt.figure(figsize = (10,10))
plt.title("Plot of one(x) and its Derivative")
plt.plot(domain, one_range, '-r', label = 'one')
plt.plot(domain, deriv_range, '-b', label = 'derivative')
x = plt.legend(loc = 'best')
```



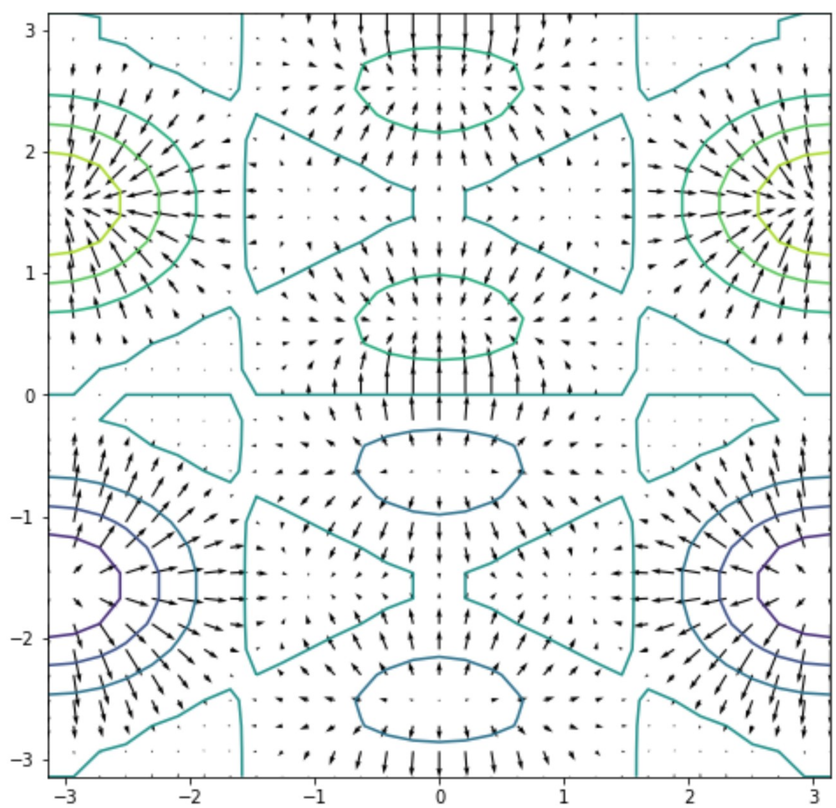
Problem 1.4

Solution

In [21]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

def plot_gradient(f, trange=(-np.pi, np.pi), samples=31):
    t = np.linspace(trange[0], trange[1], samples)
    x_values, y_values = np.meshgrid(t, t)
    z_values = np.zeros_like(x_values)
    g_x = np.zeros((samples, samples))
    g_y = np.zeros((samples, samples))
    reset_auto_diff(name='pg_x')
    reset_auto_diff(name='pg_y')
    x = Variable(0.0, 'pg_x')
    y = Variable(0.0, 'pg_y')
    for i in range(samples):
        for j in range(samples):
            x.value, y.value = x_values[i, j], y_values[i, j]
            z = f(x, y)
            z_values[i, j] = z.value
            g_x[i, j], g_y[i, j] = derivative(z, 'pg_x'), derivative(z, 'pg_y')

    plt.figure(figsize=(8, 8))
    plt.contour(t, t, z_values)
    plt.quiver(t, t, g_x, g_y)
    plt.show()
```

In [22]:
```
def trig(x, y):
    return times(times(cosine(x),sine(y)), plus(cosine(x),cosine(times(two, y))))
```

In [23]:
```
reset_auto_diff()
two = Variable(2.0, 'two')

plot_gradient(trig)
```



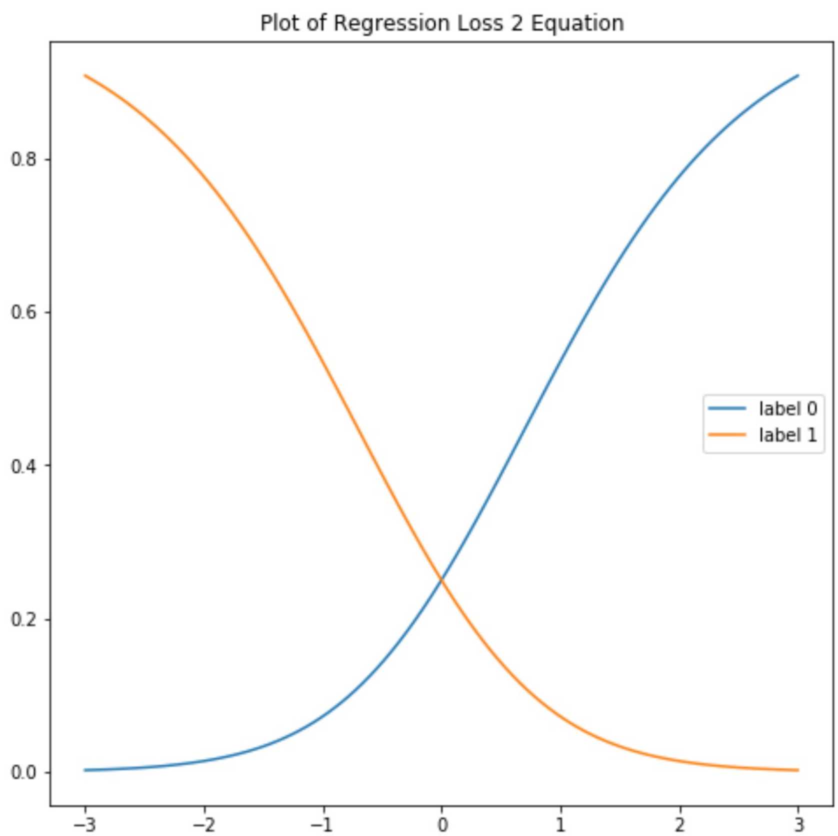
Part 2: Loss and Convexity

Problem 2.1

Solution

In [24]:
```
def regression_loss_2(y, alpha):
    score = 1 / (1 + np.exp(-alpha))
    return np.power(score - y, 2)
```

In [25]:
```
alphas = np.linspace(-3, 3, 101)
plt.figure(figsize=(8, 8))
for label in (0, 1):
    r = regression_loss_2(label, alphas)
    plt.plot(alphas, r, label='label {}'.format(label))
plt.legend(loc='best')
t = plt.title('Plot of Regression Loss 2 Equation')
```



Problem 2.2 (Exam-Style)

Solution

By substituting in 0 and α into our r_2 equation, we get $\rho(\alpha) = r_2(0, \alpha) = \ell_2(0, s(\alpha)) = p^2$.

Then $p^2 = ((1 + e^{-\alpha})^{-1})^2 = (1 + e^{-\alpha})^{-2}$.

$\rho(\alpha)$ is continuous and differentiable everywhere, so we can find the first derivative:

$$\rho'(\alpha) = \frac{2e^{-\alpha}}{(1 + e^{-\alpha})^3}$$

This is also continuous and differentiable everywhere, so we can find the second derivative:

$$\rho''(\alpha) = \frac{2e^{-2\alpha}(-e^{\alpha} + 2)}{(1 + e^{-\alpha})^4}$$

This second derivative is not postive for all of α because at $\alpha \geq ln(2)$, the e^{α} becomes ≥ 2 which makes the function negative, so $\rho(\alpha)$ is not convex.

Part 3: Logistic Regression and Regularization

In [26]:
```
import autograd.numpy as np
from autograd import grad, jacobian
```

In [27]:
```
def logistic(x, v):
    alpha = v[0] + v[1:] * x
    return 1.0 / (1.0 + np.exp(-alpha))
```

In [28]:
```
import pickle
with open('data.pickle', 'rb') as file:
    T = pickle.load(file)
```

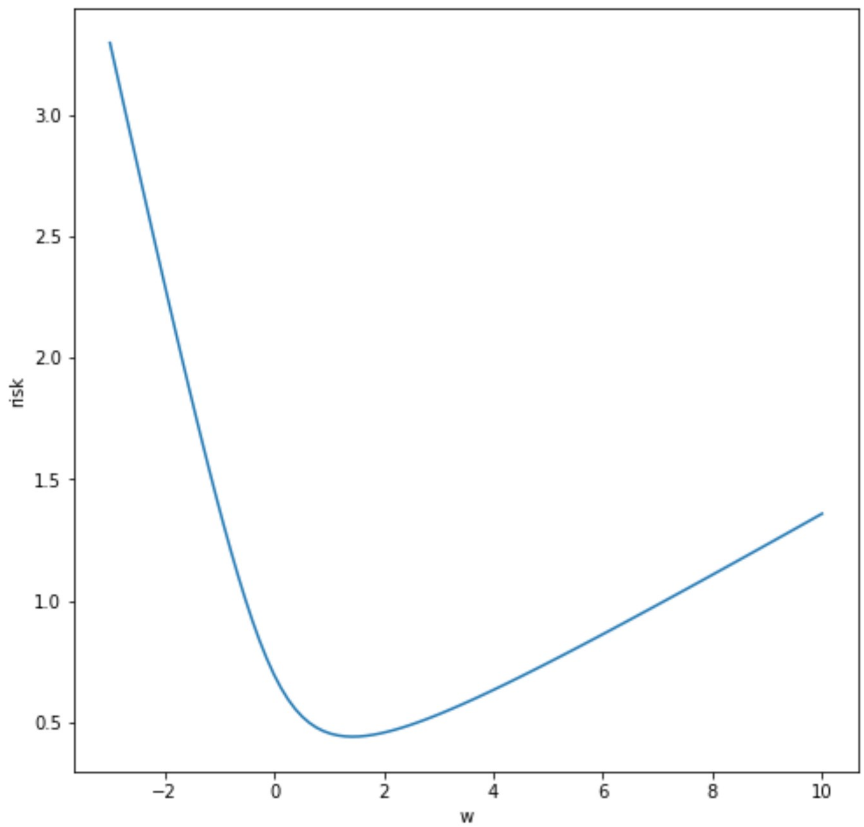
Problem 3.1

Solution

In [29]:
```
def risk(v, T):
    x = T['x']
    p = []
    small = 1.e-8
    for point in x:
        p.append(logistic(point, v))
    y = T['y']
    risk = []
    for w in range(len(v[1:])):
        total_loss = 0
        for i in range(len(y)):
            score = np.minimum(1.0 - small, np.maximum(small, p[i][w]))
            loss = - y[i] * np.log(score) - (1 - y[i]) * np.log(1 - score)
            total_loss += loss
        risk.append(total_loss/len(y))
    return np.array(risk)
```

In [30]:
```
def plot_risks(T, risk=risk):
    bw = []
    bw.append(0)
    for w in np.linspace(-3,10,101):
        bw.append(w)
    v = np.array(bw)
    j = risk(v, T)
    fig = plt.figure(figsize = (8,8))
    plt.plot(bw[1:], j)
    plt.xlabel('w')
    plt.ylabel('risk')
```

In [31]:
```
plot_risks(T)
```



Problem 3.2

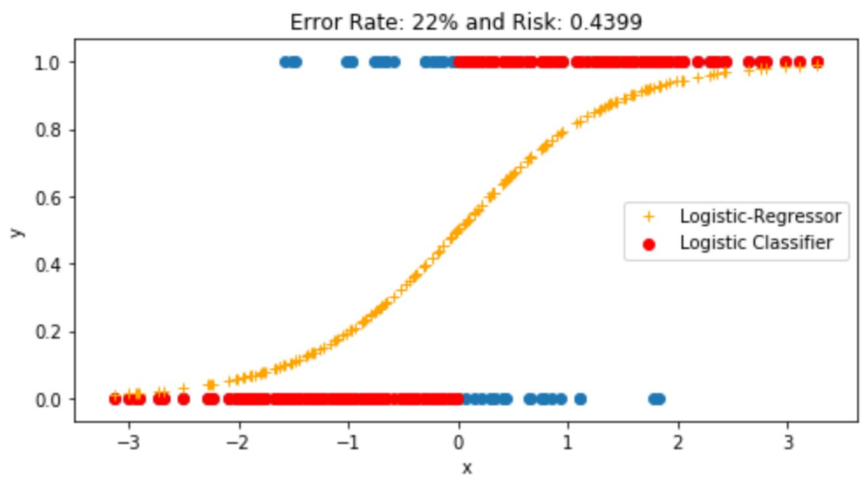
Solution

In [32]:
```
def classifier(x, v):
    y = []
    for data in x:
        if logistic(data, v) > (1/2):
            y.append(1)
        else:
            y.append(0)
    return np.array(y)
```

In [33]:
```
def performance(v, T):
    x = T['x']
    h = classifier(x, v)
    y = T['y']
    loss = 0
    for i in range(len(y)):
        if y[i] != h[i]:
            loss+=1
    error = loss/len(y) * 100
    return error, risk(v, T)
```

In [34]:
```
def plot_logistic_regressor(v, T):
    fig = plt.figure(figsize = (8,4))
    x = T['x']
    y = T['y']
    plt.scatter(x, y)
    regressor = logistic(x, v)
    plt.plot(x, regressor, '+', color='orange', label='Logistic-Regressor')
    log_classifier = classifier(x, v)
    plt.scatter(x, log_classifier, color='red',label='Logistic Classifier')
    error, risk = performance(v, T)
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Error Rate: %.0f%% and Risk: %.4f' %(error, risk[0]))
    plt.legend(loc='best')
```

In [35]:
```
v = np.array((0,1.4))
plot_logistic_regressor(v, T)
```



Problem 3.3

Solution

In [36]:
```
from scipy.optimize import minimize

def learn_logistic_regressor(T, lambda_reg=0.0):
    risk_T = lambda v: risk(v, T) + lambda_reg * np.inner(v, v)
    gradient = grad(risk_T)
    hessian = jacobian(gradient)
    v_0 = np.array((0,0))
    result = minimize(risk_T, v_0, method='Newton-CG', jac=gradient, hess=hessian)
    if result.success:
        v = result.x
        print("b = %.4f \nw = %.4f \nlambda_reg = %0.f \nIterations = %0.f"
              %(v[0], v[1], lambda_reg, result.nit))
        return result
    else:
        print(result.message)
        return None
```

In [37]:
```
v_star = learn_logistic_regressor(T)
```

```
b = -0.0095
w = 1.4299
lambda_reg = 0
Iterations = 6
```

Problem 3.4

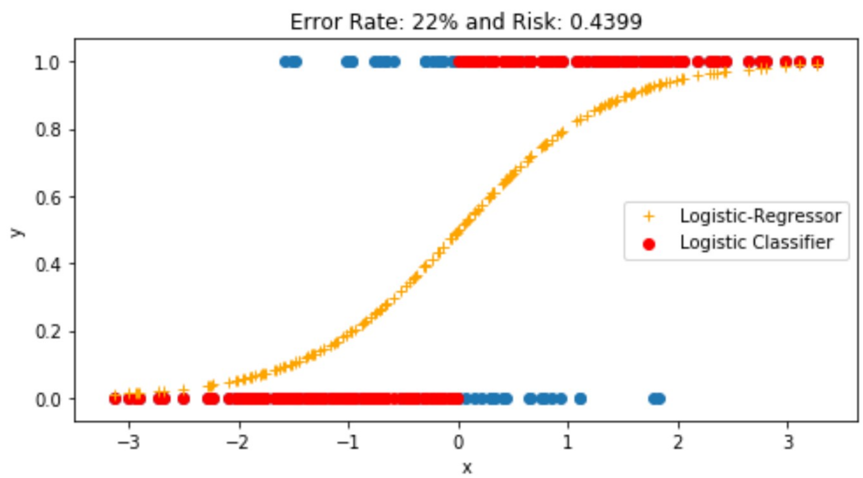
Solution

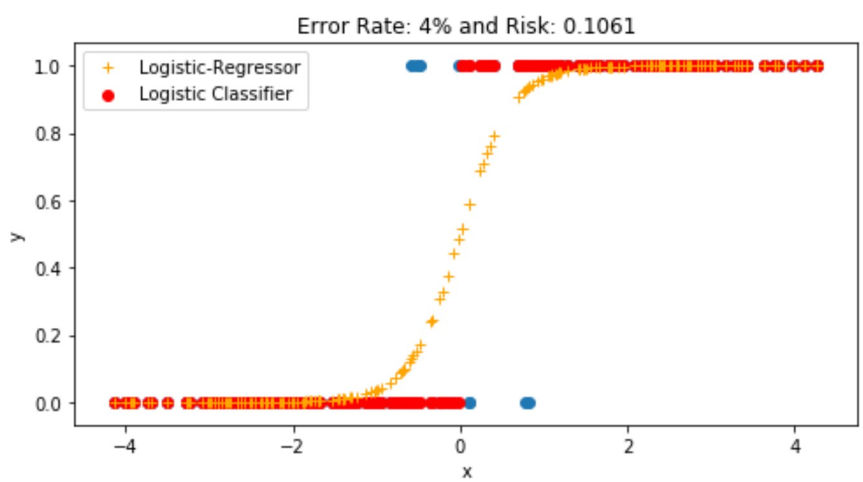
In [38]:
```
def spread_data(T, spread=0.0):
    x, y = T['x'].copy(), T['y'].copy()
    T_s = {}
    x_s = []
    for i in range(len(y)):
        adjust = -1
        if y[i] == 1:
            adjust = 1
        x_s.append(x[i] + adjust*spread)
    T_s['y'] = y
    T_s['x'] = x_s
    return T_s
```

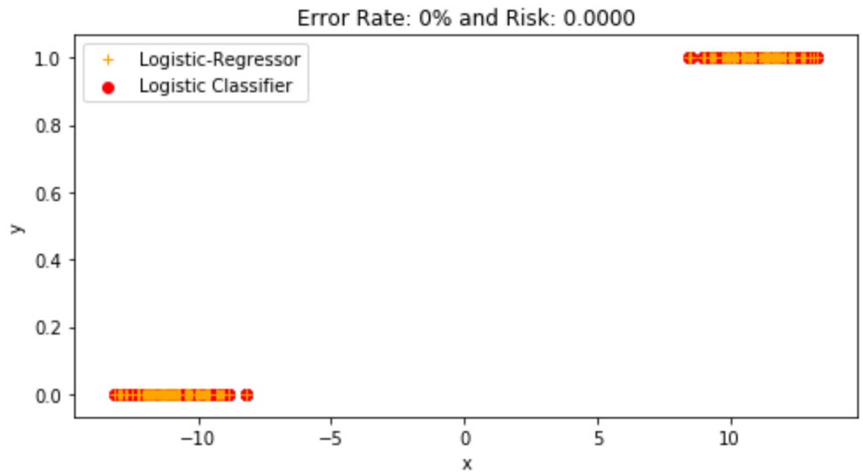
In [39]:
```
def spread_experiment(lambda_reg):
    for spread in [0.0, 1.0, 10.0]:
        T_s = spread_data(T, spread=spread)
        result = learn_logistic_regressor(T_s)
        print("Spread Value = %.f" %spread)
        print("---------------------------")
        plot_logistic_regressor(result.x, T_s)
```

In [40]:
```
spread_experiment(0.0)
```

```
b = -0.0095
w = 1.4299
lambda_reg = 0
Iterations = 6
Spread Value = 0
---------------------------
b = -0.0070
w = 3.2812
lambda_reg = 0
Iterations = 9
Spread Value = 1
---------------------------
b = -0.0017
w = 1.4882
lambda_reg = 0
Iterations = 15
Spread Value = 10
---------------------------
```



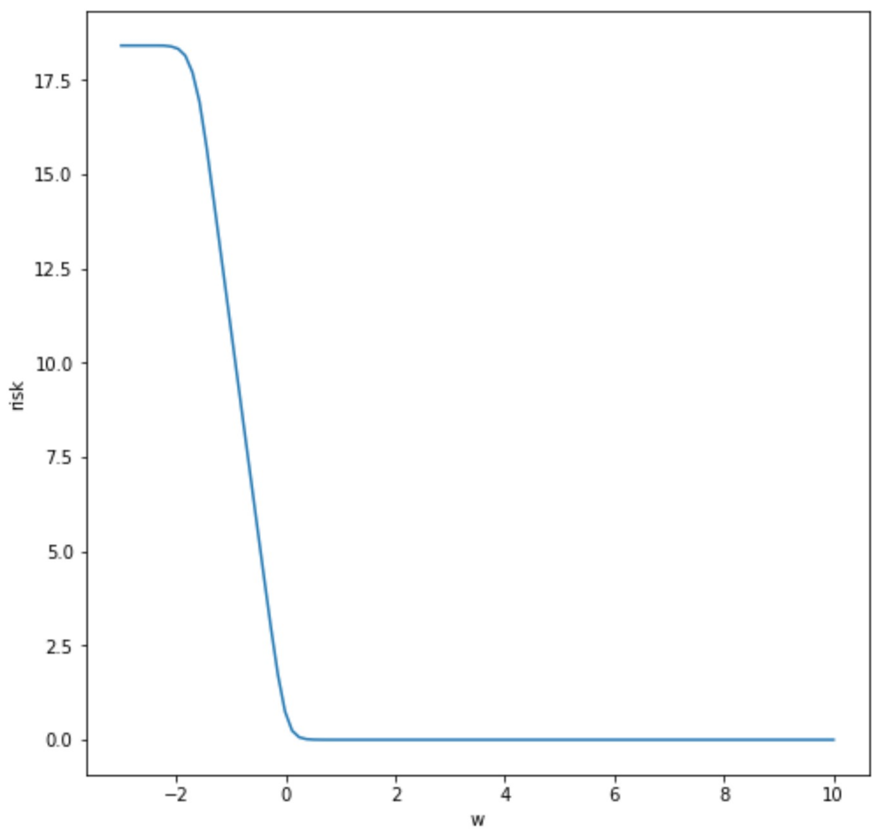




Problem 3.5

Solution

In [41]:
```
T_10 = spread_data(T, spread=10.0)
plot_risks(T_10)
```



Problem 3.6 (Exam-Style)

Solution

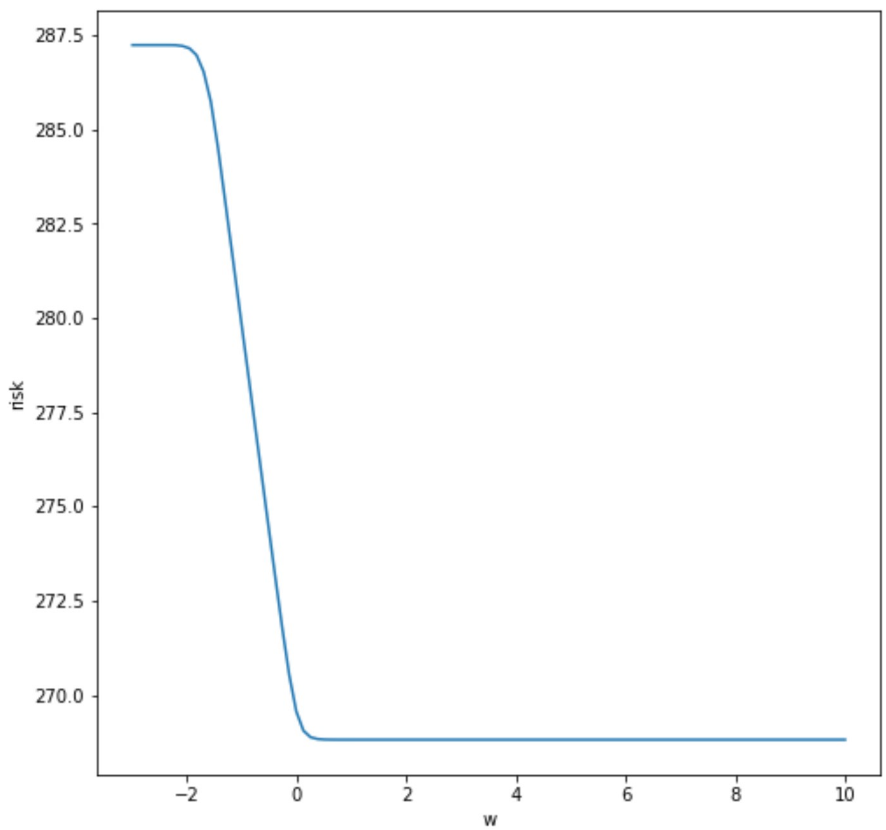
If b is zero, as w increases, this will cause the rise of the logistic-regressor function to be sharper. Since the set is separable, $\hat{w}$ is the value at which the logistic-regression function will perfectly separate the set. So any value of w beyond $\hat{w}$ will not reduce the loss, since at $\hat{w}$ the loss is at 0.

This can be seen in the graph of problem 3.5, at $w = 1.48$, the risk is 0, and stays zero after that.

Problem 3.7

Solution

In [42]:
```
risk_lambda = lambda v, T: risk(v, T) + 0.1 * np.inner(v, v)
plot_risks(T_10, risk=risk_lambda)
```



The regularization term causes the risk to significantly increase the risk of the function. Before, at $w = \hat{w}$, the risk was 0, but now its 270.