Gradient decent

Exact Gradient Computation

Given a function f, sometimes we can compute its exact gradient at any x if f's derivative is easy to compute. For example, let $f(x) = 2x^2 - 3x + \ln x$, where $x > 0$. Please compute the derivative of f and report its gradient at $x = 2$.

Your answer:
df/dx = 4x -3 +1/x
when x = 2
df/dx|x=2 = 4*2-3+1/2 = 5.5
Therefore, the gradient at x = 2 is 5.5

Numerical Gradient Computation [5 pts]

Instead of computing the derivative of a function, we can also estimate the gradient numerically with various methods. These methods are essential, especially when a callable function is not exposed due to privacy reasons, or it is hard to differentiate analytically.

To numerically compute the gradient, the simple way is to follow Newton's difference quotient: $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$. Another two-point formula is to compute the slope through the points $(x - h, f(x - h))$ and $(x + h, f(x + h))$. Let us reuse the example function $f(x) = 2x^2 - 3x + \ln x$ and test the precision of these two approaches. Define the function in the next cell, and try to compute its gradient via both methods at $x = 2$. Range h value in [0.1,0.01,0.001,0.0001] and report all gradients calculated. Which method is more accurate, and why does it work better?

In [1]:
```
import math

def f(x):
    return 2*x**2 -3*x + math.log(x)
    # Your code here
```

In [2]:
```
# Compute gradient using the first method (Newton's difference quotient)
h = [0.1,0.01,0.001,0.0001]
x = 2
for hvalue in h:
    slope = (f(x+hvalue)-f(x))/hvalue
    print('h value: ', hvalue, ' slope estimated: ', slope)
```

```
h value:  0.1  slope estimated:  5.687901641694317
h value:  0.01  slope estimated:  5.518754151103744
h value:  0.001  slope estimated:  5.50187504165045
h value:  0.0001  slope estimated:  5.5001875004201395
```

In [3]:
```
# Compute gradient using the second method
for hvalue in h:
    slope = (f(x+hvalue)-f(x-hvalue))/(hvalue *2)
    print('h value: ', hvalue, ' slope estimated: ', slope)
```

```
h value:  0.1  slope estimated:  5.500417292784909
h value:  0.01  slope estimated:  5.500004166729067
h value:  0.001  slope estimated:  5.500000041665842
h value:  0.0001  slope estimated:  5.500000000417948
```

Two-point formula is more accurate when computing the gradient. Because two-point formula provides an estimate of the slope of the function over a small interval, rather than relying solely on the function value at a single point.

Remark: You may find the gradient more accurate when using a smaller h value. However, this is not always the case. Due to the finite precision of the floating-point, rounding errors always exist and can dominate the computation when the h value is too small. Run the following two cells to observe such scenarios.

In [4]:
```
eps = 1e-15
print((f(2+eps)-f(2-eps))/2/eps)
```

```
5.551115123125783
```

In [5]:
```
eps = 1e-20
print((f(2+eps)-f(2-eps))/2/eps)
```

```
0.0
```

Logistic regression

Logistic regression is a classification tool that models the probability of an event taking place by having the log odds for the event be a linear combination of one or more independent variables. Specifically, let $\vec{x} =< x_1, \dots, x_m >$ be an m dimensional vector of independent variables (features), and y be the corresponding binary dependent variable (label). The probability of having $y = 1$ is modeled as

$$P_y = \frac{1}{1 + e^{-(b_0 + b_1 \cdot x_1 + \cdots + b_m \cdot x_m)}} = \frac{1}{1 + e^{-(b_0 + \vec{b}_{1:m} \cdot \vec{x})}}$$

Given a set of data points $< \vec{x}_k, y_k >$ with $k \in [1, n]$, how can we fit the model with these data, i.e., how to choose the best $\vec{b} = b_0, b_1 \cdots, b_m$?

One way is to write out the likelihood

$$\prod_{k:y_k=1} P_{y_k} \prod_{k:y_k=0} (1 - P_{y_k})$$

and find $b_0, b_1 \cdots, b_m$ that maximize its logarithm,

$$l = \sum_{k:y_k=1} \ln(P_{y_k}) + \sum_{k:y_k=0} \ln(1 - P_{y_k})$$

In contrast to computing the closed form gradient of a Least-squares loss in a linear model (chapter 5 of MML book), doing the same for logistic regression is not possible. Gradient descent can be used to optimize such function l, and we will implement it step-by-step. First, write a function log_likelihood in the next cell that computes the log-likelihood given data

In [6]:
```
import numpy as np
import sklearn
```

In [7]:
```
def log_likelihood(X,y,b):
    """
    X: n*m numpy data array.
    y: one dimension numpy data array of length n
    b: one dimension numpy data array of length m+1

    Return the log likelihood.
    """
    X = np.hstack((np.ones((X.shape[0], 1)), X))
    log_likelihood = 0

    for i in range(X.shape[0]):
        linear_combination = np.dot(X[i], b)
        probability_y = 1.0 / (1 + np.exp(-linear_combination))

        if y[i] == 1:
            log_likelihood += np.log(probability_y)
        else:
            log_likelihood += np.log(1 - probability_y)

    return log_likelihood
```

Test your log_likelihood function with a small example below.

In [8]:
```
X=np.array([[0.1],[0.5],[1.]])
y=np.array([0,0,1])
b=np.array([0.,1.])
# Your answer should be around -2.03
log_likelihood(X,y,b)
```

Out[8]:
```
-2.031735331771901
```

Now that we have a function to maximize, the next step is to compute the current gradient for parameter $\vec{b}$. Use the method with Newton's difference quotient, and set $h = 0.0001$. Implement the function compute_gradient in the next cell that returns the gradients of $\vec{b}$. [7 pts]

In [9]:
```
def compute_gradient(X,y,b):
# The inputs are the same as the ones of log_likelihood
    h = 0.0001
    gradient = np.zeros_like(b)
    b_copy = b.copy()
    original = log_likelihood(X, y, b_copy)
    for j in range(len(b)):
        b_copy[j] += h
        log_likelihood_plus = log_likelihood(X, y, b_copy)
        b_copy[j] -= h
        gradient[j] = (log_likelihood_plus - original) / h

    return gradient
```

In [10]:
```
# Test your function here, preserve the output
compute_gradient(X,y,b)
```

Out[10]:
```
array([-0.87853115, -0.09479906])
```

In [ ]:
```

```

Once we know how to compute the gradients, we can optimize the objective, which is log-likelihood in our case, using gradient descent. It iteratively changes the parameters in a small "step" towards the gradient direction, i.e., the direction where the objective increases at the fastest pace. Formally, denote the calculated gradients as $\Delta(\vec{b})$, we can update our parameters via $\vec{b} = \vec{b} + \gamma \cdot \Delta(\vec{b})$, where γ is the size of the "step". Repeat this process until the objective stop improving or a pre-set max number of iterations is reached. **Note in practice, the value of gradient changes over iterations and can be very large/small, so you should normalize the gradient vector every iteration, i.e., scale it to $\frac{\Delta(\vec{b})}{||\Delta(\vec{b})||_2}$, before using it to compute the new $\vec{b}$. Therefore, the update rule for parameters becomes $\vec{b} = \vec{b} + \gamma \cdot \frac{\Delta(\vec{b})}{||\Delta(\vec{b})||_2}$.**

Implement the gradient_descent function below. [7 pts]

In [11]:
```
def gradient_descent(X, y, initial_b, step_size, max_iteration):
    """
    X: n*m numpy data array.
    y: one dimension numpy data array of length n
    initial_b: one dimension numpy data array of length m+1
    step_size: scalar, the size of one step update
    max_iteration: scalar, the max number of iterations
    Return the updated coefficient vector b.
    """
    b = initial_b.copy()
    i = 0
    lll = float('-inf')
    while i < max_iteration:
        g = compute_gradient(X,y,b)
        norm = np.linalg.norm(g)
        newlll = log_likelihood(X,y,b)
        if newlll < lll:
            return b, i
        else:
            b = b + step_size * g/norm
            i +=1
        lll = newlll

    return b, i
```

Test the function with the previous example again. Print for each sample from X, based on your model, the probability of having label=1.

In [12]:
```
optimized_b, iterations = gradient_descent(X, y, b, 0.1, 1000)

# compute and print the probability for each row in X below using optim
print('optimized_b: ', optimized_b, ' number of iterations: ', iteratio
X_copy = np.hstack((np.ones((X.shape[0], 1)), X))
linear_combination = np.dot(X_copy, optimized_b)
probability_y = 1 / (1 + np.exp(-linear_combination))
print('Probabiliy of having label=1 : ', probability_y)
```

```
optimized_b:  [-60.22487755  80.4605066 ]  number of iterations:  100
0
Probabiliy of having label=1 :  [2.18284757e-23 2.07226370e-09 9.9999
9998e-01]
```

Next, we apply the implemented logistic regression model to a real dataset. The dataset is a trimmed breast-cancer-Wisconsin dataset from UCI machine learning Repository. Only 100 data points are offered in the training set to make sure the computation can be finished swiftly, no matter how you implement the optimizer. The training dataset is loaded in the next cell, and the vector $\vec{b}$ is also randomly initialized.

In [13]:
```
f = open("breast-cancer-wisconsin.data","r")
X_train = []
y_train = []
for line in f:
    tmp = []
    for part in line.strip().split(",")[1:-1]:
        tmp.append(float(part))
    y_train.append((0 if line.strip().split(",")[-1]=="2" else 1))
    X_train.append(tmp)
X_train = np.array(X_train)
y_train = np.array(y_train)
random_b = np.random.uniform(0,1,size=(10))
```

In [14]:
```
# Fit three models with different step size, report the final log-likel
# number of iterations and the final coefficent vector b.
training_step = [0.01,0.05,0.1]
max_iter = 10000
optimized_b = []
for step in training_step:
    opt_b, iterations = gradient_descent(X_train, y_train, random_b, ste
    final_likelihood = log_likelihood(X_train,y_train,opt_b)
    optimized_b.append(opt_b)
    print('final log-likelihood: ', final_likelihood)
    print('number of iterations: ', iterations)
    print('trainng size: ', step)
    print('final coefficient vector b: ', opt_b)
    print()
```

```
final log-likelihood:  -7.4854623637060165
number of iterations:  6598
trainng size:  0.01
final coefficient vector b:  [-1.49480881e+01  9.53163934e-01 -1.4129
9489e+00  9.33563095e-01
  9.97443527e-01  4.71190119e-01  3.42615276e-03  8.70188760e-01
  1.05566649e+00  1.44284202e+00]

final log-likelihood:  -20.111163866982643
number of iterations:  272
trainng size:  0.05
final coefficient vector b:  [-2.93258165  0.222884    0.24609098  0.
16036489  0.16039202  0.04539256
  0.19246876 -0.41041347  0.42188665  0.10393277]

final log-likelihood:  -46.455825306890965
number of iterations:  33
trainng size:  0.1
final coefficient vector b:  [ 0.21350136 -0.12515667  0.44552376  0.
21588382  0.03471867  0.01867804
  0.19773108 -0.66589446  0.33953741  0.03622703]
```

How do the final log-likelihood value and the number of iterations change with different step sizes? [7 pts]
Based on the test, the smaller the training step is, the larger the final log-likelihood value is. This means that the smaller training step can make the approximzation from gradient descent more likely to be accurate.
However, the smaller the training step is, the larger number of iterations it uses to get untill no improvement in approximation. The larger number of iterations lead to a longer running time, which is more time-consuming/expensive especially if we have a large training set.

Finally, load the test dataset, and predict for each sample in the test set what labels it should have using the model obtained. Compare your results with the ground truth labels, and report the accuracy rate. [4 pts]

In [15]:
```
f = open("test_data.txt","r")
X_test = []
y_test = []
for line in f:
    tmp = []
    for part in line.strip().split(",")[1:-1]:
        tmp.append(float(part))
    y_test.append((0 if line.strip().split(",")[-1]=="2" else 1))
    X_test.append(tmp)
```

In [16]:
```
# Predict based on your models and report the accuracy
estimations = []
for test in X_test:
        test.insert(0, 1)
for b in optimized_b:
    print('optimized_b for use: ', b)
    estimation = []
    for test in X_test:
        linear_combination = np.dot(test, b)
        probability_y = 1 / (1 + np.exp(-linear_combination))
        if probability_y < 0.5:
            estimation.append(0)
        else:
            estimation.append(1)
        print('Probabiliy of having label=1: ', round(probability_y,4))
    print('Estimation: ', estimation)
    print()
    estimations.append(estimation)
```

```
optimized_b for use:  [-1.49480881e+01  9.53163934e-01 -1.41299489e+0
0  9.33563095e-01
  9.97443527e-01  4.71190119e-01  3.42615276e-03  8.70188760e-01
  1.05566649e+00  1.44284202e+00]
Probabiliy of having label=1:  0.0259
Probabiliy of having label=1:  0.9938
Probabiliy of having label=1:  1.0
Probabiliy of having label=1:  1.0
Probabiliy of having label=1:  0.9912
Probabiliy of having label=1:  1.0
Probabiliy of having label=1:  0.0021
Probabiliy of having label=1:  1.0
Probabiliy of having label=1:  0.0188
Probabiliy of having label=1:  0.171
Estimation:  [0, 1, 1, 1, 1, 1, 0, 1, 0, 0]

optimized_b for use:  [-2.93258165  0.222884    0.24609098  0.1603648
9  0.16039202  0.04539256
  0.19246876 -0.41041347  0.42188665  0.10393277]
Probabiliy of having label=1:  0.1498
Probabiliy of having label=1:  0.1805
Probabiliy of having label=1:  0.9985
Probabiliy of having label=1:  0.8459
Probabiliy of having label=1:  0.9961
Probabiliy of having label=1:  0.9929
Probabiliy of having label=1:  0.2123
Probabiliy of having label=1:  0.9734
Probabiliy of having label=1:  0.1689
Probabiliy of having label=1:  0.9372
Estimation:  [0, 0, 1, 1, 1, 1, 0, 1, 0, 1]

optimized_b for use:  [ 0.21350136 -0.12515667  0.44552376  0.2158838
2  0.03471867  0.01867804
  0.19773108 -0.66589446  0.33953741  0.03622703]
Probabiliy of having label=1:  0.3182
Probabiliy of having label=1:  0.0576
Probabiliy of having label=1:  0.9818
Probabiliy of having label=1:  0.7791
Probabiliy of having label=1:  0.9962
Probabiliy of having label=1:  0.9976
Probabiliy of having label=1:  0.6776
Probabiliy of having label=1:  0.9365
Probabiliy of having label=1:  0.4762
Probabiliy of having label=1:  0.9571
Estimation:  [0, 0, 1, 1, 1, 1, 1, 1, 0, 1]
```

In [17]:
```
from sklearn.metrics import accuracy_score
print('true y: ', y_test)
for estimation in estimations:
    print('Accuracy Rate: ', accuracy_score(y_test, estimation))
```

```
true y:  [0, 1, 1, 1, 1, 1, 0, 1, 0, 1]
Accuracy Rate:  0.9
Accuracy Rate:  0.9
Accuracy Rate:  0.8
```

In [ ]:
```

```