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:

f'(x) = 4x-3+1/x

f'(2) = 8-3+1/2 = 11/2

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 [ ]:
```
import math
l = [0.1,0.01,0.001,0.0001]

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

In [ ]:
```
# Compute gradient using the first method (Newton's difference quotient)
def method1 (f,x,h):
    return (f(x+h)-f(x))/h


g = []
for h in l:
    g.append(method1(f,2,h))
print(g)
```

```
[5.687901641694317, 5.518754151103744, 5.50187504165045, 5.5001875004201395]
```

In [ ]:
```
# Compute gradient using the second method
def method2 (f,x,h):
    s_y = f(x+h)-f(x-h)
    s_x = (x+h) - (x-h)
    slope = s_y/s_x
    return slope
g = []
for h in l:
    g.append(method2(f,2,h))
print(g)
```

```
[5.500417292784904, 5.500004166729123, 5.500000041666448, 5.500000000412448]
```

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

```
5.551115123125783
```

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

```
0.0
```

In [ ]:
```

```

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 $\vec{x} =< x_1, \ldots, x_m >$ 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

$$P_y = \frac{1}{1 + e^{-(b0+b1 \cdot x1+\cdots+bm \cdot xm)}} = \frac{1}{1 + e^{-(b0+\vec{b}1:m \cdot \vec{x})}}$$

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

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

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 points and $\vec{b}$. [5 pts]

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

In [ ]:
```
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.
    """
    y_1 = 0
    y_2 = 0
    for i in range(len(X)):
        u = b[0]+b[1:]@X[i][:]
        p_y = 1/(1+ np.exp(-u))
        if y[i] == 1:
            y_1+= math.log(p_y)
        else:
            y_2 += math.log(1-p_y)
    l = y_1+y_2
    return l
```

Test your log_likelihood function with a small example below.

In [ ]:
```
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[ ]:
```
-2.031735331771901
```

Now that we have a function to maximize, the next step is to compute the gradient of the log-likelihood with respect to 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. [7 pts]

In [ ]:
```
def compute_gradient(X,y,b):
# The inputs are the same as the ones of log_likelihood
    h = 0.0001
    grad = []
    for i in range(len(b)):
        b_h = np.copy(b)
        b_h[i]+=h
        grad.append((log_likelihood(X,y,b_h)-log_likelihood(X,y,b))/h)

    return grad
```

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

Out[ ]:
```
[-0.8785311466219525, -0.09479905564102609]
```

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 [ ]:
```
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 = np.copy(initial_b)
    for k in range(max_iteration):
        b_g = compute_gradient(X,y,b) #gradient
        b_n = b_g/np.linalg.norm(b_g)
        b_previous = np.array(b)
        b+= step_size*b_n
        if log_likelihood(X,y,b) < log_likelihood(X,y,b_previous):
            b = b_previous
            break
```

```
    return k,b
```

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 [ ]:
```
k,optimized_b = gradient_descent(X, y, b, 0.1, 1000)

# compute and print the probability for each row in X below using optimized_b
p = []

for i in range(len(X)):
        u = optimized_b[0]+optimized_b[1:]@X[i][:]
        p.append(1/(1+ np.exp(-u)))
for i, prob in enumerate(p):
 print(f"Probability of having label=1 for sample {i}: {prob:.30f}")
```

```
Probability of having label=1 for sample 0: 0.000000000000000000000021831003
Probability of having label=1 for sample 1: 0.000000002072568169002954875897
Probability of having label=1 for sample 2: 0.999999983718392826403942308 39
```

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.

Fit three models with the training set using different step size ranging in [0.01,0.05,0.1] and set the max number of iterations as 10000. How do the final log-likelihood value and the number of iterations change with different step sizes? [7 pts]

In [ ]:
```
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 [ ]:
```
# Fit three models with different step size, report the final log-likelihood,
# number of iterations and the final coefficent vector b.
step_list = [0.01,0.05,0.1]
max_iter = 10000
l = []
b_list = []
k_list = []
for step in step_list:
    k,opt_b = gradient_descent(X_train,y_train,np.array(random_b),step,max_iter)
```

```
    k_list.append(k)
    b_list.append(opt_b)
    l.append(log_likelihood(X_train,y_train,opt_b))
print("Log likelihood:{}".format(l))
print("iteration:{}".format(k_list))
print("coefficient b:{}".format(b_list))
```

```
Log likelihood:[-7.485570166015979, -20.108245768517058, -45.79924229738929]
iteration:[6578, 260, 34]
coefficient b:[array([-1.49474447e+01,  9.48452874e-01, -1.41658660e+00,  9.296895
53e-01,
        9.95545390e-01,  4.67617047e-01, -4.36983986e-04,  8.66828153e-01,
        1.05299023e+00,  1.44164397e+00]), array([-2.92886087,  0.1966536 ,  0.240
68184,  0.14110111,  0.14737802,
        0.0249783 ,  0.17537148, -0.42920884,  0.4092927 ,  0.09522708]), array([
0.02521586, -0.17804215,  0.58560065,  0.00327295, -0.04870264,
       -0.19807859,  0.16778882, -0.48738013,  0.28408043,  0.21789842])]
```

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 [ ]:
```
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 [ ]:
```
 # Predict the probabilities of y=1 for the test dataset
X_test = np.array(X_test)
n_test = len(X_test)
X_n = np.hstack((np.ones((n_test, 1)), X_test))

p = []
b = b_list
for i in range(len(b_list)):
        z = np.dot(b[i],X_n)
        probabilities_test = 1 / (1 + np.exp(-z))
        # Convert probabilities to binary labels using a threshold of 0.5
        predicted_labels = (probabilities_test > 0.5).astype(int)
        # Calculate the accuracy by comparing the predicted labels with the ground
        y_test = np.array(y_test)
        accuracy = np.mean(predicted_labels == y_test)*100

        print(f"Model {i+1} (Step size: {step_list[i]})")
        print(f"Accuracy: {accuracy:.2f}%\n")
```

```
Model 1 (Step size: 0.01)
Accuracy: 30.00%

Model 2 (Step size: 0.05)
Accuracy: 60.00%

Model 3 (Step size: 0.1)
Accuracy: 80.00%
```