REI602M Machine Learning - Homework 1

Due: Sunday 24.1.2021

Objectives: Python, NumPy and Matplotlib warmup, gradient descent, linear regression

Name: (Alexander Guðmundsson), **email:** (alg35@hi.is (mailto:alg35@hi.is)), **collaborators:** ()

Please provide your solutions by filling in the appropriate cells in this notebook, creating new cells as needed. Hand in your solution in PDF format on Gradescope. Make sure that you are familiar with the course rules on collaboration (encouraged) and copying (very, very, bad!)

This assignment is somewhat time consuming so start early.

1) [Python warmup, 15 points] The following code implements the matrix-vector product $y = Ax$ where A is an $n \times m$ matrix, x is a column vector with m elements and y a column vector with n elements, $y_i = \sum_{k=1}^{m} A_{ik} x_k$. (Note that in practice one would use NumPy's `dot` function to perform the matrix-vector multiplication).

Note: A useful Python/Nympy tutorial which covers most everything we need in REI602M can be found here: https://cs231n.github.io/python-numpy-tutorial/ (https://cs231n.github.io/python-numpy-tutorial/)

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

def matvecmul(A, x):
    # Computes the matrix-vector product Ax using elementwise operations
    n, m = A.shape
    assert(m == x.shape[0])
    y=np.zeros(n)
    for i in range(0, n):
        for j in range(0, m):
            y[i] = y[i] + A[i,j] * x[j]
    return y

# Test
A=np.array([[1, 2], [3, 4]])
x=np.array([5, 41])
print(matvecmul(A,x)) # Outputs [87, 179]
```

```
[ 87. 179.]
```

a) Write a Python function which computes the sum of each row in the matrix A, i.e. $y_i = \sum_{j=1}^{m} A_{ij},\ i = 1, \ldots, n$, by accessing individual matrix/vector elements directly as is done in the `matvecmul` function above.

In [7]:
```
import numpy as np

def rowsum(A):
    rs = []
    for i in A:
        s = sum(i)
        rs.append(s)

    return rs

# Test
A=np.array([[1, 2, 3], [3, 4, -5]])
print("rowsum(A) :", rowsum(A)) # Outputs [6, 2]
```

```
rowsum(A) : [6, 2]
```

b) Modify the `rowsum` function so that only positive elements are included in the sum, again by accessing individual matrix elements.

In [8]:
```
import numpy as np

def rowsumpos(A):
    rsp = []
    for i in A:
        s = sum(i)
        if s > 0:
            rsp.append(s)
    return rsp

# Test
A=np.array([ [1, 2, 3], [3, 4, -5] ])
print("rowsumpos(A) :", rowsumpos(A)) # Outputs [6, 2]
# Test2
B = np.array([[1, 2, 3], [3, -4, -5]])
print("rowsumpos(B) :", rowsumpos(B)) # Outputs [6]
```

```
rowsumpos(A) : [6, 2]
rowsumpos(B) : [6]
```

c) Compute the matrix product $C = AB$ where A is $n \times m$, B is $m \times p$ and $C_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$ is $n \times p$, by calling `matvecmul` repeatedly.

In [9]:
```
import numpy as np

def matmul(A,B):
    C = []
    #Using Xline x YColumn assuming X == A || B and Y == A || B
    x = [] #will be the array of all the columns in the array that has more lines
    Anew = [] # will be the array of the lines in the array that has less lines

    if len(A) > len(B):
        for i in range(len(A[0])):
            temp = [c[i] for c in A]
            Anew.append(temp) #creating an array of columns in A
        x = B

    else:
        for i in range(len(B[0])):
            temp = [c[i] for c in B]
            Anew.append(temp) #creating an array of columns in B
        x = A


    # now we can create the output array by looping through the columns by callir

    for k in x:
        tempArr = matvecmul(np.asarray(Anew), np.asarray(k))
        C.append(tempArr)

    return C

# Test
A1=np.array([[1, 2, 3], [4, 5, 6] ])
B1=np.array([[7, 10], [8, 11], [9, 12]])
print("matmul(A1, B1) :", matmul(A1, B1)) # Outputs [[50, 68], [122,167]]

# Test 2
A2=np.array([[7, 10], [8, 11], [9, 12]])
B2=np.array([[1, 2, 3], [4, 5, 6] ])
print("matmul(A2, B2) :", matmul(A2, B2)) # Outputs [[50, 68], [122,167]]
```

```
matmul(A1, B1) : [array([50., 68.]), array([122., 167.])]
matmul(A2, B2) : [array([50., 68.]), array([122., 167.])]
```

2) [NumPy warmup, 15 points] Repeat a), b) and c) in 1) using NumPy functionality. Aim for fast code by avoiding for-loops as much as possible.

In [10]:
```
import numpy as np

def rowsum(A):
    rs = np.asarray(np.sum(A,axis=1).tolist()) #using numpy sum to find the sum o
    return rs

def rowsumpos(A):
    rsp = np.asarray(np.sum(A, axis=1).tolist())
    rsp = rsp[np.where(rsp>0)] # removing all elements less than 0
    return rsp

def matmul(A,B):
    #Using XLine x YColumn assuming X == A || B and Y == A || B
    C = np.array([])
    if len(A) > len(B):
        C = np.dot(B,A)
    else:
        C = np.dot(A,B)

    return C


# Test rowsum(A)
A=np.array([[1, 2, 3], [3, 4, -5]])
print("rosum(A) :", rowsum(A)) # Outputs [6, 2]

# Test rowsumpos(A)
A=np.array([ [1, 2, 3], [3, 4, -5] ])
print("rowsumpos(A) :", rowsumpos(A)) # Outputs [6, 2]

# Test2 rowsumpos(A)
B = np.array([[1, 2, 3], [3, -4, -5]])
print("rowsumpos(B) :", rowsumpos(B)) # Outputs [6]

# Test matmul(A, B)
A1=np.array([[1, 2, 3], [4, 5, 6] ])
B1=np.array([[7, 10], [8, 11], [9, 12]])
print("matmul(A1, B1) :", matmul(A1, B1)) # Outputs [[50, 68], [122,167]]

# Test 2 matmul(A, B)
A2=np.array([[7, 10], [8, 11], [9, 12]])
B2=np.array([[1, 2, 3], [4, 5, 6] ])
print("matmul(A2, B2) :", matmul(A2, B2)) # Outputs [[50, 68], [122,167]]
```

```
rosum(A) : [6 2]
rowsumpos(A) : [6 2]
rowsumpos(B) : [6]
matmul(A1, B1) : [[ 50  68]
 [122 167]]
matmul(A2, B2) : [[ 50  68]
 [122 167]]
```

3) [Linear regression with gradient descent, 40 points] Here you implement a gradient descent algorithm for linear regression and apply it to a small data set. You then add code to track the

minimization process. For many learning algorithms this is a very important part of the training process.

a) [30 points] Create a function `linreg_gd` which implements gradient descent for linear regression with the least squares cost function, using maximum number of iterations as a stop criteria. See the lecture notes for details and the Jupyter notebook `vika01_demo` on Canvas. Try to avoid Python for-loops as much as possible by using NumPy functionality in your final implementation.

Use your function to fit a linear regression model on the form

$$f_\theta(x) = \theta_0 + \sum_{j=1}^{p} \theta_j x_j$$

to the `Avertising_centered` dataset provided with this notebook.

Note 1: You can use the `linear_reg` dataset used in the `vika01_demo` notebook to debug your code. Write the θ values to the screen every iteration (or every 100 or 1000 or ...) to monitor convergence. You can compare the output of your code with the values you get by solving the normal equations directly.

Note 2: You may want to begin by implementing a version that does not rely heavily on NumPy. Once you get it working, gradually introduce NumPy operations into the code.

Note 3: The advertising dataset is discussed in the ISLR textbook (see sections 2.1, 3.1 and 3.2). Here the data has been transformed by subtracting the mean of each input variable and dividing by its standard deviation so that all inputs now have mean zero and standard deviation one (more on this later in the course). As a result, gradient descent converges faster to the optimal θ values but note that the values now differ from those reported in the book.

$(\theta_0, \theta_1, \theta_2, \theta_3) =$ (14.02338317, 3.9190074, 2.79334112, -0.02077732)

In [12]:

```
import numpy as np
import matplotlib.pyplot as plt




def linreg_gd(X, y, theta):
    n = X.shape[0]
    alpha = 0.0001 #alpha value
    maxiter = 10 #max amount of iterations
    p = X.shape[1]
    SSQ = 0
    for iter in range(maxiter):
        for i in range(1,n):
            error = (np.dot(theta,X[i,:]) - y[i]) # the the Sum of Square error
            SSQ = SSQ + (error**2)
            for j in range(0,p):
                theta[j] = theta[j] - alpha*error*X[i,j]
    return theta,SSQ




# Load the data

#TV,Radio,Newspaper,Sales
data = np.genfromtxt('./Advertising_centered.csv', delimiter=',',
                     skip_header=1)
print(data.shape)

# Insert code here
n = data.shape[0]

y = data[:,-1]


X = np.c_[np.ones(n), data[:,0:-1]]



# Call your function and report theta values
p = X.shape[1]
theta = np.zeros(p)


SSQarr = np.zeros(100) #all SSQ values
totalTheta = np.zeros(100) #all theta values in all iterations
for i in range(1,101):
    theta,SSQ = linreg_gd(X,y,theta)
    totalTheta[i-1] = theta[0]
    SSQarr[i-1] = SSQ
    print("after "+str(i*10) + ' iterations')
    print('(θ0,θ1,θ2,θ3) =', theta)
```

```
after 480 iterations
(θ0,θ1,θ2,θ3) = [14.0144367   3.91186162  2.78778349 -0.03077045]
after 490 iterations
(θ0,θ1,θ2,θ3) = [14.01461499  3.9119156   2.78810646 -0.03110382]
after 500 iterations
(θ0,θ1,θ2,θ3) = [14.0147609   3.91195995  2.7883904  -0.03139672]
after 510 iterations
(θ0,θ1,θ2,θ3) = [14.01488031  3.9119964   2.78864002 -0.03165407]
after 520 iterations
(θ0,θ1,θ2,θ3) = [14.01497802  3.91202634  2.78885949 -0.03188019]
after 530 iterations
(θ0,θ1,θ2,θ3) = [14.01505795  3.91205094  2.78905245 -0.03207887]

after 540 iterations
(θ0,θ1,θ2,θ3) = [14.01512334  3.91207115  2.78922209 -0.03225345]
after 550 iterations
(θ0,θ1,θ2,θ3) = [14.01517682  3.91208775  2.78937125 -0.03240684]
after 560 iterations
(θ0,θ1,θ2,θ3) = [14.01522055  3.91210138  2.78950239 -0.03254163]
after 570 iterations
(θ0,θ1,θ2,θ3) = [14.01525631  3.91211258  2.78961769 -0.03266006]
```

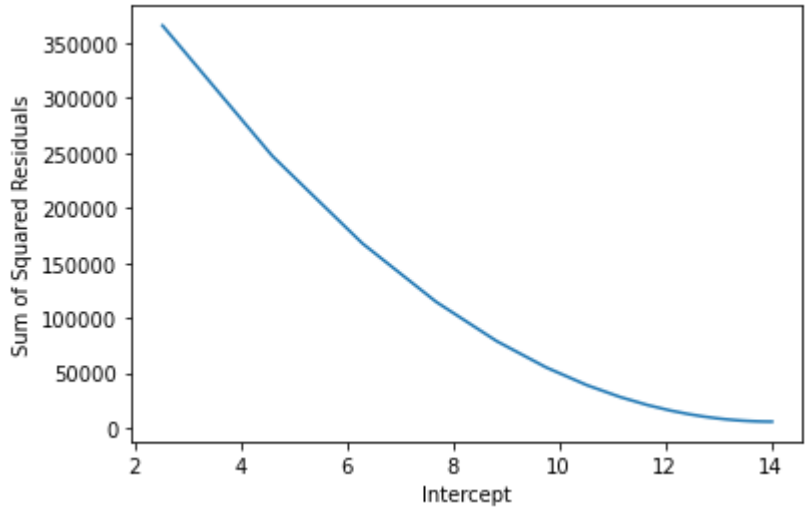
b) [10 points] Create a plot that shows the value of the cost function, $J(\theta)$ in each iteration when you apply your gradient descent function to the data in `Advertising_centered` .

Note: Create a vector J with k_max elements where k_max is the maximum number of iterations. Use `matplotlib.pyplot.plot` or `matplotlib.pyplot.semilogy` to create the figure.

In [13]:
```
import numpy as np
from matplotlib.pyplot import figure
from matplotlib.pyplot import plot

plt.plot(totalTheta,SSQarr)
plt.xlabel('Intercept')
plt.ylabel('Sum of Squared Residuals')
```

Out[13]:
```
Text(0, 0.5, 'Sum of Squared Residuals')
```



4) [An alternative cost function for linear regression, 30 points]

The least-squares cost function

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

is the workhorse of linear regression but it has a significant drawback, namely it is sensitive to 'outliers', data points that differ significantly from the rest. If the prediction, $f_\theta(x^{(i)})$ differes considerably from the true value $y^{(i)}$, the squared difference will have a large contribution to $J(\theta)$, magnifying the effect of outlier points. Using the absolute error $|f_\theta(x^{(i)}) - y^{(i)})|$ instead of the squared error, reduces the effects of outliers but the price to pay is the optimization becomes more difficult.

The *log-cosh* cost function

$$J(\theta) = \sum_{i=1}^{n}\log\cosh(f_\theta(x^{(i)}) - y^{(i)})$$

alleviates the outlier problem to some extent by behaving like the squred error when the difference between model predictions and data is small but like the absolute error when the difference is large. The log-cosh function is differentiable and can be used in gradient descent algorithms.

Note 1: Outliers in data can arise for many reasons, they can e.g. represent faulty measurements or simply be due to high variability in the data. Detecting outliers prior to fitting a machine learning model is in general not trivial. A machine learning algorithm should preferrably be robust to the presence of (few) outliers in the data.

a) [10 points] Derive the gradient for the *log-cosh* cost function

Note 1: Start with the case $n = 1$. The case $n \geq 1$ follows by noting that the derivative of a sum of functions is equal to the sum of their derivatives.

Note 2: You can use the LaTeX support in the Jupyter/Colab notebooks (see e.g. this notebook for examples) or simply write down your solution on paper, take a photo with your phone and include as an image below using

```
<img src="mynd.png" alt="drawing" width="300"/>
```

Insert derivation here

$$J(\theta) = \sum_{i=1}^{n} \log\cosh(f_\theta(x^{(i)}) - y^{(i)})$$

let's start by finding the derive when n = 1:

for simplicity I will give

$$f_\theta(x^{(i)}) - y^{(i)} = u$$

Let's use the chain rule to find

$$\log\cosh(u)$$

The chain rule is defined as:

$$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x) \; where \; f(x) = ln(x) \; and \; g(x) = cosh(x)$$

that means that:

$$\frac{d}{dx} ln(\cosh(u)) = \frac{1}{\cosh(u)} \sinh(u)$$

b) [20 points] Implement a gradient descent algorithm for linear regression that uses the *log-cosh* cost function by modifying your code from problem 3). Test your code on the `outlier.csv` data set using the model

$$f_\theta(x) = \theta_0 + \theta_1 x_1$$

and compare the results by applying least squares regression to the same data. Create a scatter plot of the data that includes the two regression lines in the plot (see `v01_demo`). What can you conclude from this (single) experiment?

$(\theta_0, \theta_1) =$ (0.67039587, 1.12416193)

In [14]:
```
import numpy as np

def linreg_gd_logcosh(X, y,theta):
    n = X.shape[0]
    alpha = 0.0001 #alpha value
    maxiter = 10 #max amount of iterations
    p = X.shape[1]
    SSQ = 0
    for iter in range(maxiter):
        ones = np.ones(p)
        for i in range(1,n):
            cosh = (np.cosh(np.dot(theta,X[i,:]) - y[i]))
            sinh = (np.sinh(np.dot(theta,X[i,:]) - y[i]))
            error = (1/cosh)*sinh
            SSQ = SSQ + error
            for j in range(0,p):
                theta[j] = theta[j] - alpha*error*X[i,j]
    return theta,SSQ

# Load data
data = np.genfromtxt('./outlier.csv', delimiter=',',
                     skip_header=1)
print(data.shape)

# Insert code here
n = data.shape[0]

p = data.shape[1]

y = data[:,-1]

X = np.c_[np.ones(n), data[:,0:-1]]

# Call your function

#Theta for linreg_gd_logcosh function
thetaLog = np.zeros(p)
Logarr = np.zeros(1000) #all logcosh values
totalLogTheta = np.zeros(1000) #all theta values in all iterations
for i in range(1,1001):
    thetaLog,Log = linreg_gd_logcosh(X,y,thetaLog)
    totalLogTheta[i-1] = thetaLog[0]
    Logarr[i-1] = Log
    if (i-1)%10 == 0:
        print("after "+str((i-1)*10) + ' iterations')
        print('(θ0,θ1) =', thetaLog)




#Theta for linreg_gd function
thetaSSQ = np.zeros(p)
SSQarr = np.zeros(1000) #all SSQ values
totalSSQTheta = np.zeros(1000) #all theta values in all iterations
for i in range(1,1001):
```

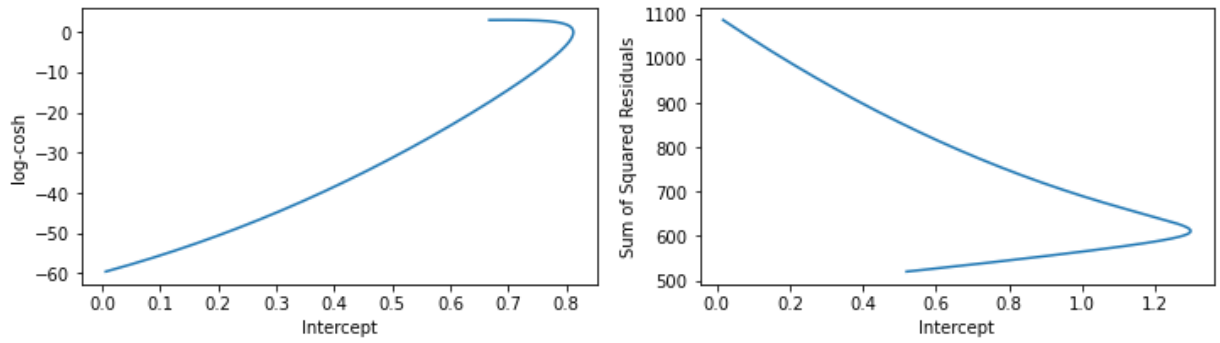
```
    thetaSSQ,SSQ = linreg_gd(X,y,thetaSSQ)
    totalSSQTheta[i-1] = thetaSSQ[0]
    SSQarr[i-1] = SSQ
```

```
(θ0,θ1) = [0.80800165 0.80258622]
after 4900 iterations
(θ0,θ1) = [0.80661295 0.80994013]
after 5000 iterations
(θ0,θ1) = [0.80508331 0.81720522]
after 5100 iterations
(θ0,θ1) = [0.80342369 0.82438727]

after 5200 iterations
(θ0,θ1) = [0.80164424 0.83149163]
after 5300 iterations
(θ0,θ1) = [0.79975436 0.83852326]
after 5400 iterations
(θ0,θ1) = [0.79776274 0.84548675]
after 5500 iterations
(θ0,θ1) = [0.79567744 0.85238634]
after 5600 iterations
(θ0,θ1) = [0.79350588 0.85922599]
after 5700 iterations
(θ0,θ1) = [0.79125498 0.86600933]
```

In [15]:
```
import numpy as np
from matplotlib.pyplot import figure
import matplotlib.pyplot as plt


# Draw a figure to compare the results with the least squares solution
plt.figure(figsize=(12,3))
plt.subplot(1,2,1)
plt.plot(totalLogTheta,Logarr)
plt.xlabel('Intercept')
plt.ylabel('log-cosh')
plt.subplot(1,2,2)
plt.plot(totalSSQTheta,SSQarr)
plt.xlabel('Intercept')
plt.ylabel('Sum of Squared Residuals')
plt.show()
```

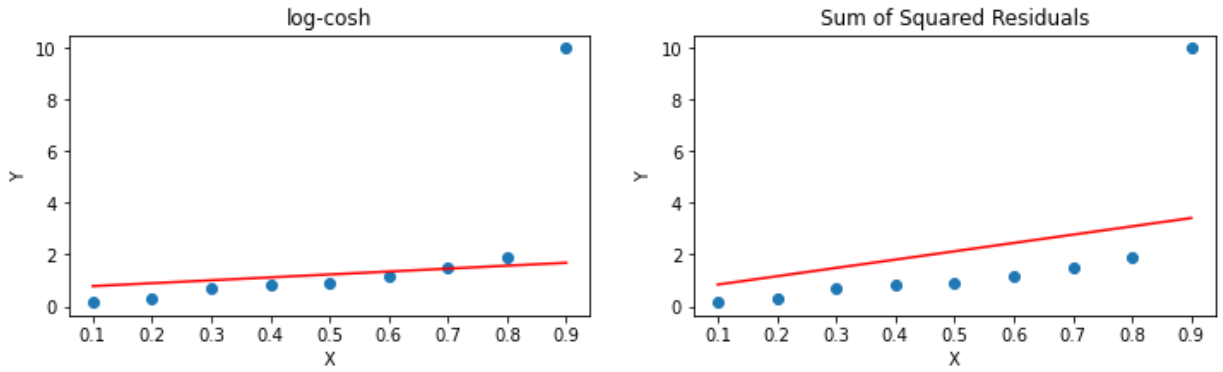


In [16]:
```
import numpy as np
from matplotlib.pyplot import figure
import matplotlib.pyplot as plt


#figure of best line in logcosh
plt.figure(figsize=(12,3))
plt.subplot(1,2,1)
line = thetaLog[0] + thetaLog[1]*X[:,1]
plt.scatter(X[:,1],y)
plt.plot([min(X[:,1]),max(X[:,1])], [min(line), max(line)], color='red')
plt.title('log-cosh')
plt.xlabel('X')
plt.ylabel('Y')

#figure of best line in Sum of squared Residuals
plt.subplot(1,2,2)
line = thetaSSQ[0] + thetaSSQ[1]*X[:,1]
plt.scatter(X[:,1],y)
plt.plot([min(X[:,1]),max(X[:,1])], [min(line), max(line)], color='red')
plt.title('Sum of Squared Residuals')
plt.xlabel('X')
plt.ylabel('Y')
```

Out[16]: Text(0, 0.5, 'Y')



From this we can conclude that Sum of Squared Residuals will not reach 0 because of the outlier in the data, log-cosh does reach 0 at intercept = 0.8.

In [ ]: