REI 602M Homework 1

Júlíus Ingi Guðmundsson

TOTAL POINTS

100 / 100

QUESTION 1

1 1) 15 / 15

✓ - 0 pts Correct

- 2.5 pts rowsum is incorrect
- 2.5 pts rowsumpos is incorrect
- 2.5 pts matvecmul is incorrect
- 15 pts Problem is missing
- 1 pts Minor matmul error
- 5 pts rowsumpos completely wrong

QUESTION 2

2 2) 15 / 15

✓ - 0 pts Correct

- 2.5 pts rowsum is incorrect
- 2.5 pts rowsumpos is incorrect
- 2.5 pts matvecmul is incorrect
- 15 pts Problem is missing
- 5 pts rowsum completely wrong
- 5 pts rowsumpos completely wrong
- 4 pts rowsumpos just for matrix with two rows

QUESTION 3

40 pts

3.1 3a) 30 / 30

✓ - 0 pts Correct

- 7.5 pts Values for hyper-parameters (alpha/maxiter) not properly chosen
- 7.5 pts Code is inefficient, numpy is (almost) not used at all
- 7.5 pts Stochastic Gradient Descent is used instead of Batch Gradient Descent
- 15 pts On the right track but important parts are missing
- 30 pts Problem is missing

3.2 3b) 10 / 10

✓ - 0 pts Correct

- 2.5 pts The graph does not provide a clear illustration of the iteration progress
- 2.5 pts The output of the algorithm is not in agreement with the data set
- 5 pts Incorrect cost function
- 10 pts Problem is missing

QUESTION 4

30 pts

4.1 4a) 10 / 10

✓ - 0 pts Correct

- 2.5 pts Minor mistake in the derivation of the gradient
- 2.5 pts The case n>1 is missing
- 5 pts Several mistakes in the derivation of the gradient
- 10 pts An image showing the answer is missing from the report
- 10 pts Problem is missing
- 1 pts Missing i in n>1 case
- 1 pts Gradient is a vector not a sum of partial derivatives.
- 1 pts Mistake at the very end

4.2 4b) 20 / 20

✓ - 0 pts Correct

- 20 pts Problem is missing
- 5 pts Updating step is incorrect
- 5 pts Graph is not clear, e.g. no way to tell models apart
- 5 pts Conclusions are missing
- 5 pts More iterations are needed
- 2.5 pts Wrong data preparation

- **2.5 pts** Gradient just on one random point
- **2.5 pts** Wrong plot
- **2.5 pts** Wrong call of a linreg funcion
- **2.5 pts** Plot without x values

REI602M Machine Learning - Homework 1

Due: Sunday 24.1.2021

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

Name: Júlíus Ingi Guðmundsson, **email:** jig3@hi.is, **collaborators:** (if any)

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/

In [1]:
```
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 [2]:
```
def rowsum(A):
    n, m = A.shape
    rs = np.zeros(n)
    for i in range(0, n):
        for j in range(0, m):
            rs[i] = rs[i] + A[i,j]
    return rs

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

```
[6. 2.]
```

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

In [3]:
```
def rowsumpos(A):
    n, m = A.shape
    rsp = np.zeros(n)
    for i in range(0, n):
        for j in range(0, m):
            if A[i,j] >= 0:
                rsp[i] = rsp[i] + A[i,j]
    return rsp

# Test
A=np.array([ [1, 2, 3], [3, 4, -5] ])
print(rowsumpos(A)) # Outputs [6, 7]
```

```
[6. 7.]
```

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 [4]:
```
def matmul(A,B):
    n, m = A.shape
    m, p = B.shape
    C = np.zeros((n,p))
    for j in range(0, p):
        temp_B = np.transpose(B[:,j])
        C[j] = matvecmul(A,temp_B) # multiply matrix A with each transposed column o
    return np.transpose(C) #use shape to visualize vectors & matrices

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

```
[[ 50.  68.]
 [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 [5]:
```
def rowsum(A):
    rs = np.sum(A, axis = 1)
    return rs

def rowsumpos(A):
    rsp = np.where(A>0,A,0).sum(1)
    return rsp

def matmul(A,B):
    C = np.dot(A, B)
    return C

# Test rowsum, rowsumpos and matmul in the same way as before
# Rowsum test
A=np.array([[1, 2, 3], [3, 4, -5]])
print(rowsum(A)) # Outputs [6, 2]

# Rowsumpos test
A=np.array([ [1, 2, 3], [3, 4, -5] ])
print(rowsumpos(A)) # Outputs [6, 7]

# Matmul test
A=np.array([[1, 2, 3], [4, 5, 6] ])
```

1 1) **15 / 15**

✓ **- 0 pts** **Correct**

- 2.5 pts rowsum is incorrect

- 2.5 pts rowsumpos is incorrect

- 2.5 pts matvecmul is incorrect

- 15 pts Problem is missing

- 1 pts Minor matmul error

- 5 pts rowsumpos completely wrong

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

In [3]:
```
def rowsumpos(A):
    n, m = A.shape
    rsp = np.zeros(n)
    for i in range(0, n):
        for j in range(0, m):
            if A[i,j] >= 0:
                rsp[i] = rsp[i] + A[i,j]
    return rsp

# Test
A=np.array([ [1, 2, 3], [3, 4, -5] ])
print(rowsumpos(A)) # Outputs [6, 7]
```

```
[6. 7.]
```

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 [4]:
```
def matmul(A,B):
    n, m = A.shape
    m, p = B.shape
    C = np.zeros((n,p))
    for j in range(0, p):
        temp_B = np.transpose(B[:,j])
        C[j] = matvecmul(A,temp_B) # multiply matrix A with each transposed column o
    return np.transpose(C) #use shape to visualize vectors & matrices

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

```
[[ 50.  68.]
 [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 [5]:
```
def rowsum(A):
    rs = np.sum(A, axis = 1)
    return rs

def rowsumpos(A):
    rsp = np.where(A>0,A,0).sum(1)
    return rsp

def matmul(A,B):
    C = np.dot(A, B)
    return C

# Test rowsum, rowsumpos and matmul in the same way as before
# Rowsum test
A=np.array([[1, 2, 3], [3, 4, -5]])
print(rowsum(A)) # Outputs [6, 2]

# Rowsumpos test
A=np.array([ [1, 2, 3], [3, 4, -5] ])
print(rowsumpos(A)) # Outputs [6, 7]

# Matmul test
A=np.array([[1, 2, 3], [4, 5, 6] ])
```

```
B=np.array([[7, 10], [8, 11], [9, 12]])
print(matmul(A, B)) # Outputs [[50, 68], [122,167]]
```

```
[6 2]
[6 7]
[[ 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.

$(\theta_0, \theta_1, \theta_2, \theta_3) =$ (14.02358439, 3.91934852, 2.79267405, -0.02195703)

In [281…

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

alpha = 0.001 #arbitrary number first chosen and then adjusted for the learning rate
maxiter = 150 #number of iterations can be adjusted as well

def linreg_gd(X, y):
    theta = np.zeros(len(X[0])) #define a vector to hold the theta values
    for i in range(0, maxiter):
        prediction = np.dot(X, theta)
        gradient = X.T.dot(prediction - y)
        theta = theta - alpha * gradient # Updating theta values
    return theta

# Load the data
data = np.genfromtxt('Advertising_centered.csv', delimiter=',', skip_header=1)
```

2 2) **15 / 15**

✓ **- 0 pts** **Correct**

- 2.5 pts rowsum is incorrect

- 2.5 pts rowsumpos is incorrect

- 2.5 pts matvecmul is incorrect

- 15 pts Problem is missing

- 5 pts rowsum completely wrong

- 5 pts rowsumpos completely wrong

- 4 pts rowsumpos just for matrix with two rows

```
B=np.array([[7, 10], [8, 11], [9, 12]])
print(matmul(A, B)) # Outputs [[50, 68], [122,167]]
```

```
[6 2]
[6 7]
[[ 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.

$(\theta_0, \theta_1, \theta_2, \theta_3) =$ (14.02358439, 3.91934852, 2.79267405, -0.02195703)

In [281…

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

alpha = 0.001 #arbitrary number first chosen and then adjusted for the learning rate
maxiter = 150 #number of iterations can be adjusted as well

def linreg_gd(X, y):
    theta = np.zeros(len(X[0])) #define a vector to hold the theta values
    for i in range(0, maxiter):
        prediction = np.dot(X, theta)
        gradient = X.T.dot(prediction - y)
        theta = theta - alpha * gradient # Updating theta values
    return theta

# Load the data
data = np.genfromtxt('Advertising_centered.csv', delimiter=',', skip_header=1)
```

```
#print(data.shape) # Sanity check
n = data.shape[0]

# Insert code here
y = data[:,-1] # Output variable is in the last column in this file
X = np.c_[np.ones(n), data[:,0:-1]] # Insert a column of ones (intercept term)
theta_ex = np.linalg.solve(X.T.dot(X), X.T.dot(y))
print('Exact solution: theta =', theta_ex)

# Call your function and report theta values
print('Linear regression gradient descent solution: theta =', linreg_gd(X, y))
```

```
Exact solution: theta = [14.02358439  3.91934852  2.79267405 -0.02195703]
Linear regression gradient descent solution: theta = [14.02358439  3.91934852  2.792
67405 -0.02195703]
```

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 [279…

```
# Insert code to generate figure here
alpha = 0.000025 #arbitrary number first chosen and then adjusted
maxiter = 6500 #number of iterations can be adjusted as well

#Note: Different values for alpha and maxiter can be chosen to show a more gradual c

def linreg_gd(X, y):
    theta = np.zeros(len(X[0]))
    cost_history = np.zeros(maxiter)
    for i in range(maxiter):
        prediction = np.dot(X, theta)
        gradient = X.T.dot(prediction - y)
        theta = theta - alpha * gradient # Updating theta values
        cost = (1/2) * np.sum((X.dot(theta) - y) ** 2) # Calculating the cost functi
        cost_history[i] = cost
    return theta, cost_history

data = np.genfromtxt('Advertising_centered.csv', delimiter=',', skip_header=1)
#print(data.shape) # Sanity check
n = data.shape[0]
y = data[:,-1] # Output variable is in the last column in this file
X = np.c_[np.ones(n), data[:,0:-1]] # Insert a column of ones (intercept term)
theta_ex = np.linalg.solve(X.T.dot(X), X.T.dot(y))
print('Exact solution: theta=', theta_ex)

theta, cost_history = linreg_gd(X, y)
print('Linear regression gradient descent solution: theta =', theta)
plt.plot(cost_history)
plt.xlabel('Number of iterations')
plt.ylabel('Cost')
plt.show()
```

```
Exact solution: theta= [14.02358439  3.91934852  2.79267405 -0.02195703]
Linear regression gradient descent solution: theta = [14.02358439  3.91934852  2.792
67405 -0.02195703]
```

3.1 3a) 30 / 30

✓ **- 0 pts** **Correct**

- 7.5 pts Values for hyper-parameters (alpha/maxiter) not properly chosen

- 7.5 pts Code is inefficient, numpy is (almost) not used at all

- 7.5 pts Stochastic Gradient Descent is used instead of Batch Gradient Descent

- 15 pts On the right track but important parts are missing

- 30 pts Problem is missing

```
#print(data.shape) # Sanity check
n = data.shape[0]

# Insert code here
y = data[:,-1] # Output variable is in the last column in this file
X = np.c_[np.ones(n), data[:,0:-1]] # Insert a column of ones (intercept term)
theta_ex = np.linalg.solve(X.T.dot(X), X.T.dot(y))
print('Exact solution: theta =', theta_ex)

# Call your function and report theta values
print('Linear regression gradient descent solution: theta =', linreg_gd(X, y))
```

```
Exact solution: theta = [14.02358439  3.91934852  2.79267405 -0.02195703]
Linear regression gradient descent solution: theta = [14.02358439  3.91934852  2.792
67405 -0.02195703]
```

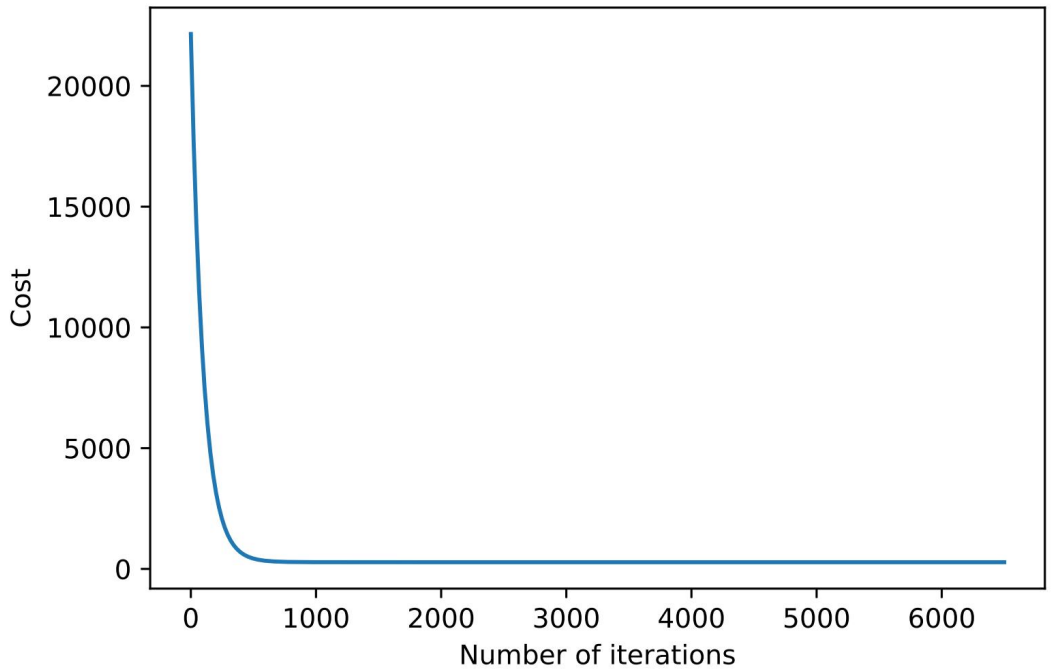
b) [10 points] Create a plot that shows the value of the cost function, $J(\theta)$ in each iteration when you apply your gradient descent function to the data in `Advertising_centered` .

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 [279…

```
# Insert code to generate figure here
alpha = 0.000025 #arbitrary number first chosen and then adjusted
maxiter = 6500 #number of iterations can be adjusted as well

#Note: Different values for alpha and maxiter can be chosen to show a more gradual c

def linreg_gd(X, y):
    theta = np.zeros(len(X[0]))
    cost_history = np.zeros(maxiter)
    for i in range(maxiter):
        prediction = np.dot(X, theta)
        gradient = X.T.dot(prediction - y)
        theta = theta - alpha * gradient # Updating theta values
        cost = (1/2) * np.sum((X.dot(theta) - y) ** 2) # Calculating the cost functi
        cost_history[i] = cost
    return theta, cost_history

data = np.genfromtxt('Advertising_centered.csv', delimiter=',', skip_header=1)
#print(data.shape) # Sanity check
n = data.shape[0]
y = data[:,-1] # Output variable is in the last column in this file
X = np.c_[np.ones(n), data[:,0:-1]] # Insert a column of ones (intercept term)
theta_ex = np.linalg.solve(X.T.dot(X), X.T.dot(y))
print('Exact solution: theta=', theta_ex)

theta, cost_history = linreg_gd(X, y)
print('Linear regression gradient descent solution: theta =', theta)
plt.plot(cost_history)
plt.xlabel('Number of iterations')
plt.ylabel('Cost')
plt.show()
```

```
Exact solution: theta= [14.02358439  3.91934852  2.79267405 -0.02195703]
Linear regression gradient descent solution: theta = [14.02358439  3.91934852  2.792
67405 -0.02195703]
```



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.

3.2 3b) 10 / 10

✓ **- 0 pts** Correct

- 2.5 pts The graph does not provide a clear illustration of the iteration progress

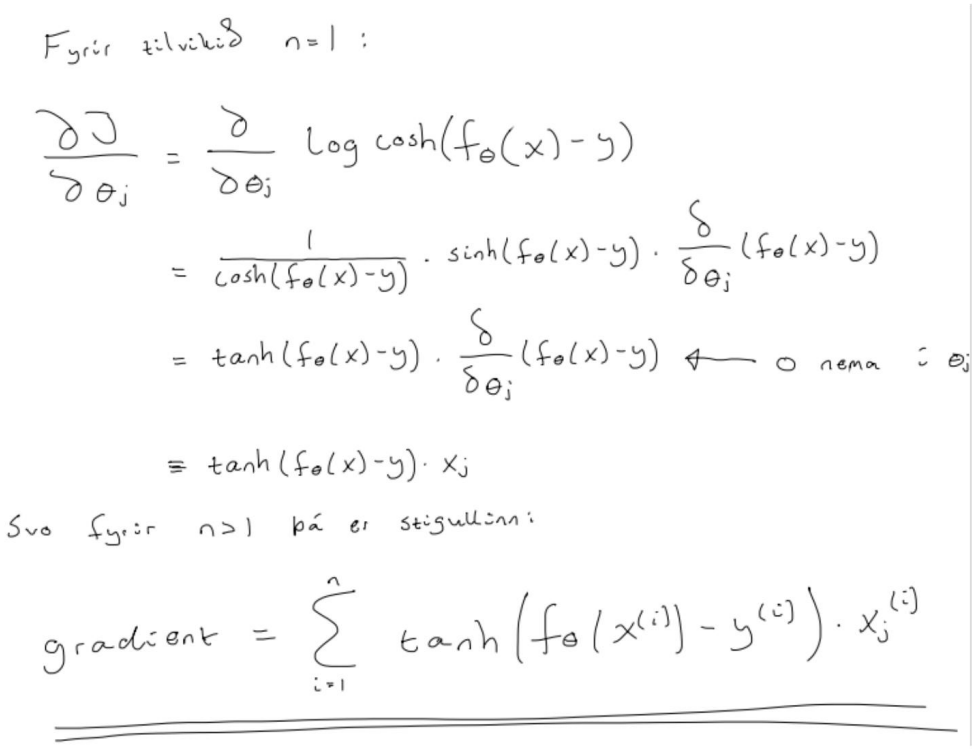
- 2.5 pts The output of the algorithm is not in agreement with the data set

- 5 pts Incorrect cost function

- 10 pts Problem is missing

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="gradient.png" alt="drawing" width="300"/>
```



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?

$(\theta_0, \theta_1) =$ (-0.24426574 2.99990751)

In [283…

```
def linreg_gd_logcosh(X, y):
    theta = np.zeros(len(X[0])) #define a vector to hold the theta values
    for i in range(0, maxiter):
        prediction = np.dot(X, theta)
        gradient = X.T.dot(np.tanh((prediction - y)))
        theta = theta - alpha * gradient
    return theta

# Load data
data = np.genfromtxt('outlier.csv', delimiter=',')
#print(data.shape) # Sanity check
n = data.shape[0]

y = data[:,-1] # Output variable is in the last column in this file
X = np.c_[np.ones(n), data[:,0:-1]] # Insert a column of ones (intercept term)


alpha = 0.1
maxiter = 1000
```

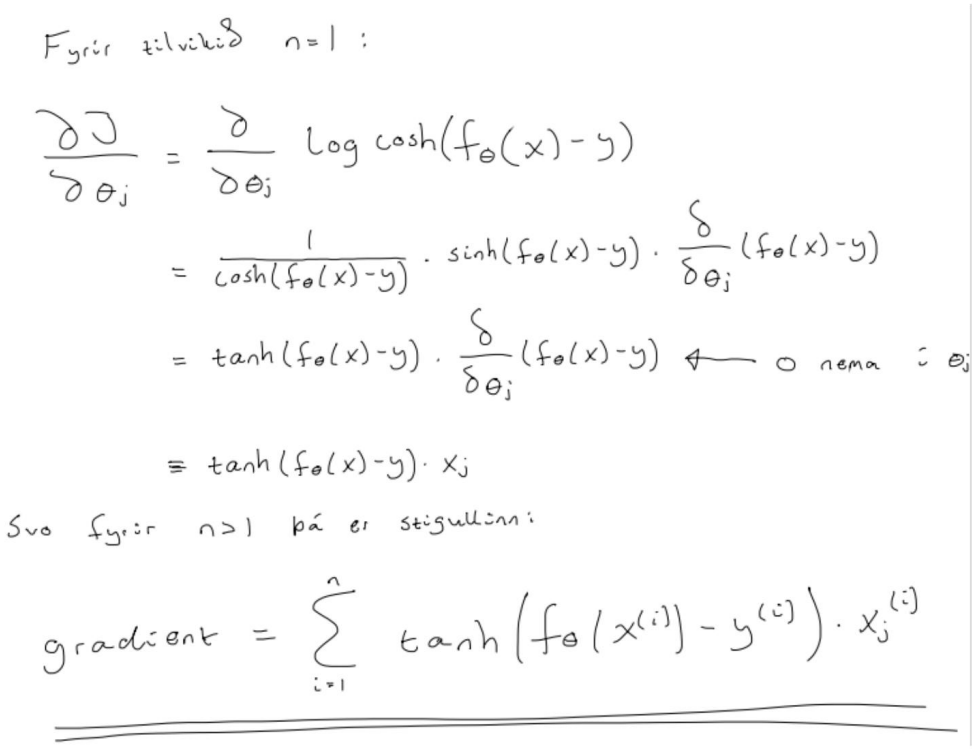
4.1 4a) 10 / 10

✓ - **0 pts** **Correct**

- **2.5 pts** Minor mistake in the derivation of the gradient
- **2.5 pts** The case n>1 is missing
- **5 pts** Several mistakes in the derivation of the gradient
- **10 pts** An image showing the answer is missing from the report
- **10 pts** Problem is missing
- **1 pts** Missing i in n>1 case
- **1 pts** Gradient is a vector not a sum of partial derivatives.
- **1 pts** Mistake at the very end

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="gradient.png" alt="drawing" width="300"/>
```



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?

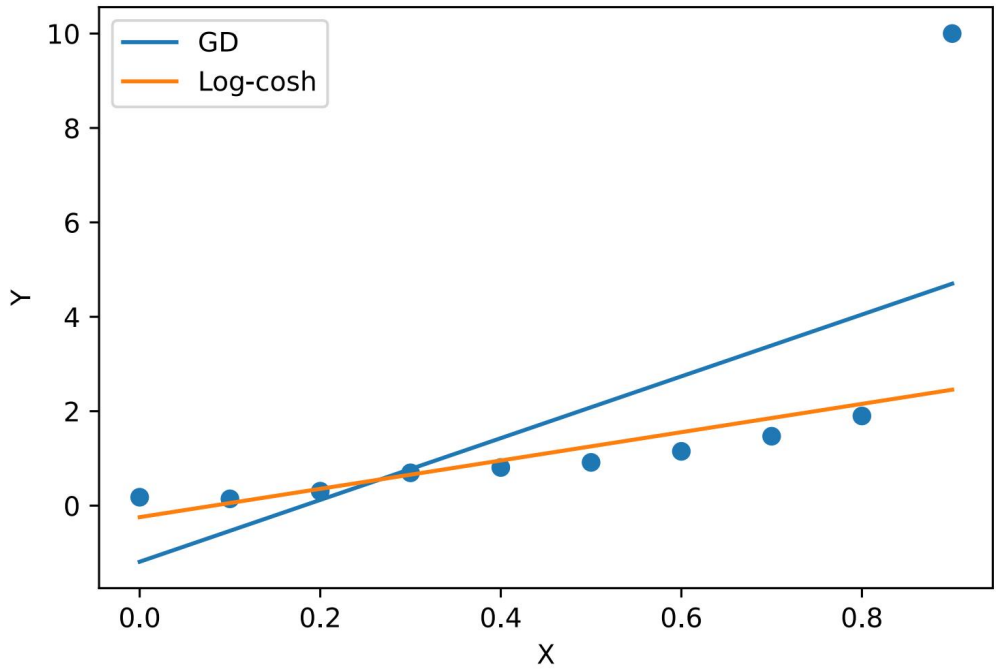
$(\theta_0, \theta_1) =$ (-0.24426574 2.99990751)

In [283…

```
def linreg_gd_logcosh(X, y):
    theta = np.zeros(len(X[0])) #define a vector to hold the theta values
    for i in range(0, maxiter):
        prediction = np.dot(X, theta)
        gradient = X.T.dot(np.tanh((prediction - y)))
        theta = theta - alpha * gradient
    return theta

# Load data
data = np.genfromtxt('outlier.csv', delimiter=',')
#print(data.shape) # Sanity check
n = data.shape[0]

y = data[:,-1] # Output variable is in the last column in this file
X = np.c_[np.ones(n), data[:,0:-1]] # Insert a column of ones (intercept term)


alpha = 0.1
maxiter = 1000
```

```
# Call your function and report theta values
theta_lrgd = linreg_gd(X, y)
intercept, slope = theta_lrgd
print("GD thetas =", linreg_gd(X, y))

theta_lrgd_logcosh = linreg_gd_logcosh(X, y)
intercept_logcosh, slope_logcosh = theta_lrgd_logcosh
print("Log-cosh thetas =", theta_lrgd_logcosh)

# Draw a figure to compare the results with the least squares solution
plt.scatter(X[:,1],y)
plt.xlabel('X')
plt.ylabel('Y')
plt.plot(X[:,1], slope * X[:,1] + intercept, label = 'GD')
plt.plot(X[:,1], slope_logcosh * X[:,1] + intercept_logcosh, label = 'Log-cosh')
plt.legend()
plt.show()
```

```
GD thetas = [-1.18829091  6.54442424]
Log-cosh thetas = [-0.24426574  2.99990751]
```



What can you conclude from this (single) experiment? It is clear that the log-cosh cost function provides a more accurate prediction compared to the least squares regression. The log-cosh cost function is not as strongly affected by outliers and therefore fits better on the graph. Least squares regression, on the other hand, is sensitive to outliers. If the problem has outliers and does not require difficult optimization, then the log-cosh cost function is better suited than the least squares regression.

4.2 4b) 20 / 20

✓ - **0 pts** **Correct**

- **20 pts** Problem is missing
- **5 pts** Updating step is incorrect
- **5 pts** Graph is not clear, e.g. no way to tell models apart
- **5 pts** Conclusions are missing
- **5 pts** More iterations are needed
- **2.5 pts** Wrong data preparation
- **2.5 pts** Gradient just on one random point
- **2.5 pts** Wrong plot
- **2.5 pts** Wrong call of a linreg funcion
- **2.5 pts** Plot without x values