REI602M Machine Learning - Homework 1

Due: Sunday 24.1.2021

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

Name: Þorsteinn Sigurðsson, **email:** ths251@hi.is, **collaborators:** Ágúst logi, Jón ingimarsson, Arnar leó.

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

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

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

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
            q,p = B.shape
            assert(m==q)
            C = np.zeros((n,q))
            for i in range(0,n):
                C [:,i]= matvecmul(A,B[:, i])
            C = np.delete(C, len(C), 1)

            return C
        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.sum(A, axis=1, where = [A>0])
            return rsp

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

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

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

        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 2]
[[ 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.

$(\theta_0, \theta_1, \theta_2, \theta_3) =$ (... insert your results here ...)

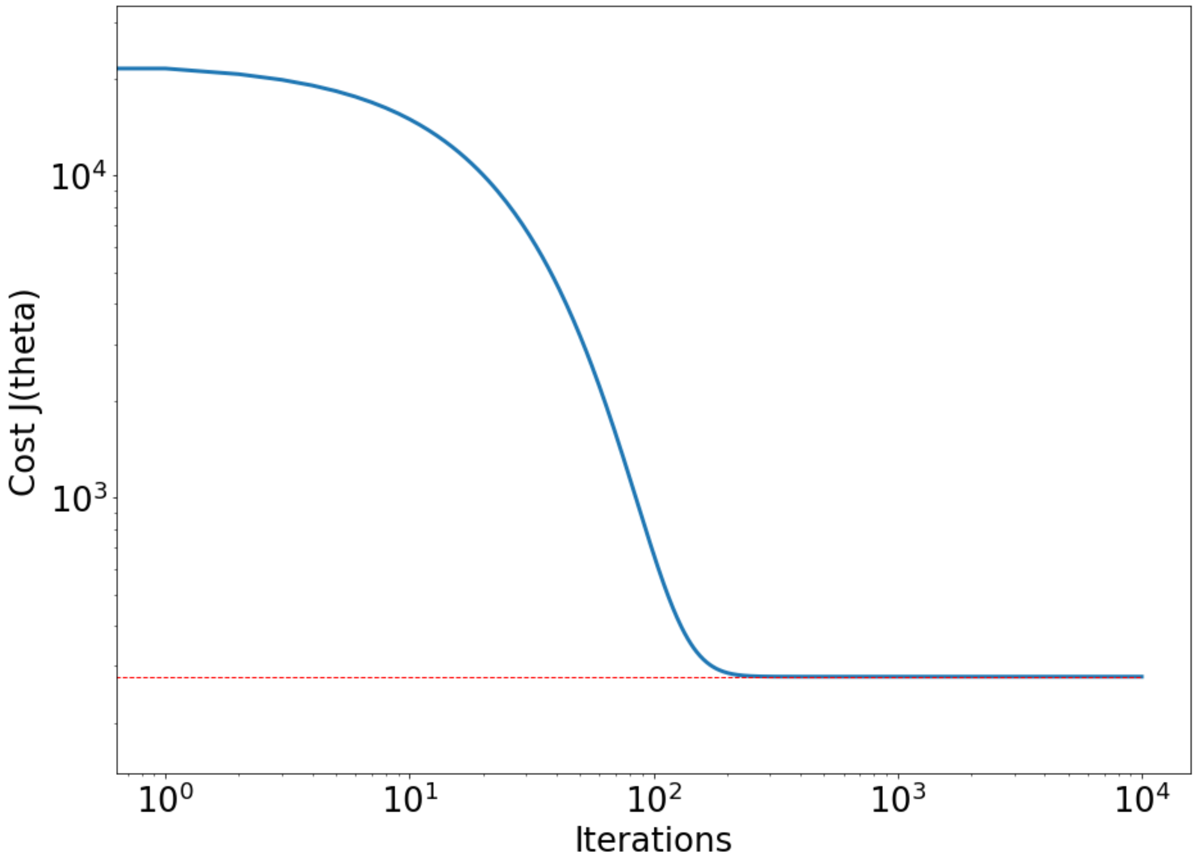
In [6]:
```python
import matplotlib.pyplot as plt

def linreg_gd(X, y):
    theta = np.linalg.solve(X.T.dot(X), X.T.dot(y))
    return theta

# Load the data
# Insert code here


data=np.genfromtxt('Advertising_centered.csv', delimiter=',', skip_header=1)
X = np.c_[np.ones(data.shape[0]), data[:,0:-1]]
y=data[:,-1]

# Call your function and report theta values
print(f'Theta gildin okkar : {linreg_gd(X,y)}')
```

```
Theta gildin okkar : [14.02358439  3.91934852  2.79267405 -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 [7]:
```
# Insert code to generate figure here
theta = linreg_gd(X,y)
maxIteration = 10000

def cost(theta, X, y):
    return 0.5*np.sum((np.matmul(X, theta)-y)**2)



def iterated_cost(J, theta, X, y):
    n = X.shape[0]
    p = X.shape[1]
    theta = np.zeros(p)
    alpha= 0.0001
    #Lúppa til að finna hvert cost á theta.
    for i in range(maxIteration):
        J[i] = cost(theta, X, y)
        yhat = np.matmul(X, theta)
        error = yhat-y
        theta = theta-alpha*np.dot(X.T, error)
    plot_cost(J)
    return None

def plot_cost(J):
    #Fall til að plotta
    plt.rcParams.update({'font.size': 26})
    plt.figure(figsize = (15.0, 11.0))
    plt.plot(J, lw=3)
    plt.xscale("log")
    plt.yscale("log")
    plt.ylabel("Cost J(theta)")
    plt.xlabel("Iterations")
    plt.ylim(0.5*np.min(J), 1.50*np.max(J))
    plt.plot([0, maxIteration], [np.min(J), np.min(J)], 'r--', lw=1)
    plt.show()
    return None

J = np.zeros(maxIteration)
iterated_cost(J, theta, X, y)
```



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.

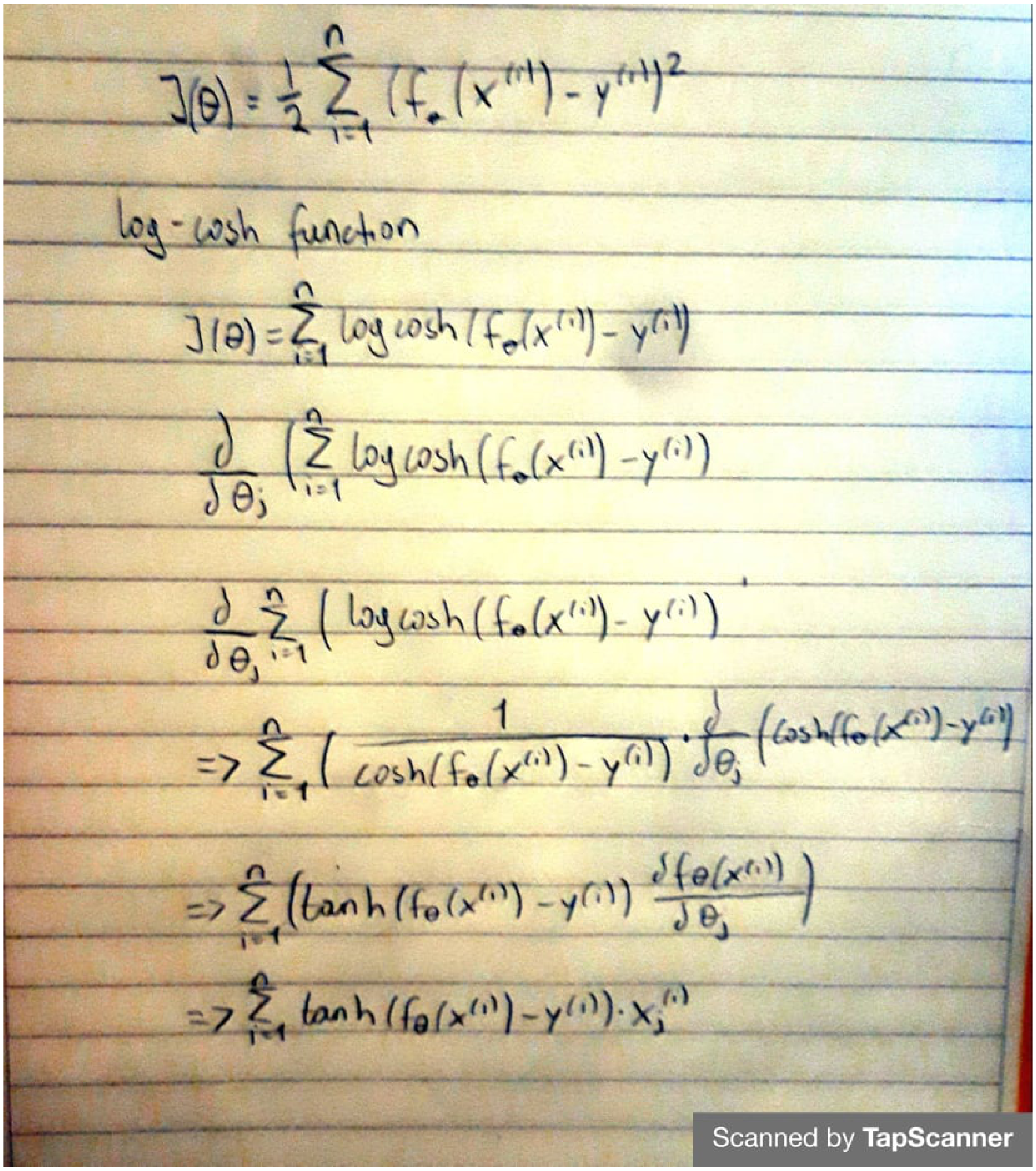
Note 2: You can use the LaTeX support in the Jupyter/Colab notebooks (see e.g. this notebook for examples) or simply write down your solution on paper, take a photo with your phone and include as an image below using

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

Insert derivation here ...

In [8]:
```
from IPython.display import Image
Image(filename="logcosh.png")
```

Out[8]:



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

log-cosh function

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

$$\frac{\partial}{\partial\theta_j}\left(\sum_{i=1}^{n} \log\cosh(f_\theta(x^{(i)}) - y^{(i)})\right)$$

$$\frac{\partial}{\partial\theta_j}\sum_{i=1}^{n}\left(\log\cosh(f_\theta(x^{(i)}) - y^{(i)})\right)$$

$$\Rightarrow \sum_{i=1}^{n}\left(\frac{1}{\cosh(f_\theta(x^{(i)}) - y^{(i)})} \cdot \frac{\partial}{\partial\theta_j}(\cosh(f_\theta(x^{(i)}) - y^{(i)})\right)$$

$$\Rightarrow \sum_{i=1}^{n}\left(\tanh(f_\theta(x^{(i)}) - y^{(i)}) \frac{\partial f_\theta(x^{(i)})}{\partial\theta_j}\right)$$

$$\Rightarrow \sum_{i=1}^{n} \tanh(f_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

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

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

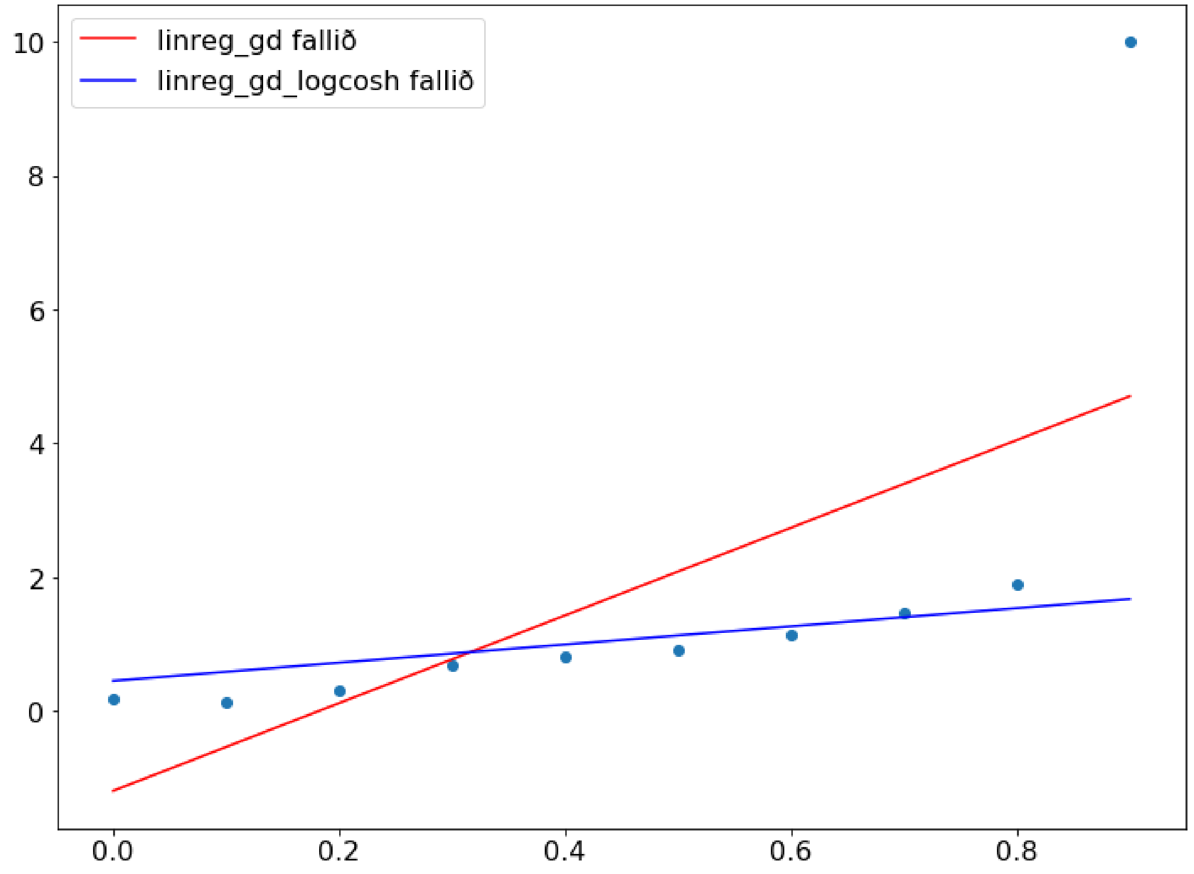
and compare the results by applying least squares regression to the same data. Create a scatter plot of the data that includes the two regression lines in the plot (see `v01_demo`). What can you conclude from this (single) experiment?

$(\theta_0, \theta_1) =$ (... insert your results here ...)

In [10]:
```
def linreg_gd_logcosh(X, y):
    n = X.shape[0]
    p = X.shape[1]
    theta = np.zeros(p)
    alpha = 0.0001

    for i in range(maxIteration):
        #Tekið frá afleiðunni í logcosh fallinu
        y_hat = np.matmul(X, theta)
        error = y_hat-y
        theta = theta - alpha*np.matmul(np.tanh(error), X)

    return theta

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

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


# Call your function
theta_a = linreg_gd(X, y)
theta_b = linreg_gd_logcosh(X,y)
print(theta_a)
print(theta_b)
```

```
[-1.18829091  6.54442424]
[0.45273858 1.3576434 ]
```

In [11]:
```
plt.rcParams.update({'font.size': 16})
plt.figure(figsize = (12.0, 9.0))
plt.scatter(X[:, 1], y)

plt.plot(X[:, 1], (theta_a[0]+theta_a[1]*X[:, 1]), c='r', label='linreg_gd fal
lið')
plt.plot(X[:, 1], (theta_b[0]+theta_b[1]*X[:, 1]), c='b', label='linreg_gd_log
cosh fallið')
plt.title("")
plt.legend()
plt.show()
```



In [64]:
```
##Sjáum að logcosh fallið fellur betur á línuna miðað við grafið okkar og (out
liers) hafa minni áhrif.
```