ERHVERVSAKADEMI AARHUS

Machine Learning, August 25th, 2023.

Exercises Day 2.

Sila. August 25th, 2023.

Exercise 1.1.

Jupyter, up and running. See exercise in “JupyterUpAndRunnng.pdf” (Canvas).

Exercise 1.2.

Getting started.

Lets start by plotting some points:

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

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

plt.plot(X,y, "b.")
plt.axis([0,2,0,15])

plt.plot()
plt.show()
```

See (check) that X and y are arrays as expected?

Notice:

```
import numpy
a = [1, 1, 1 ,1, 1]
ar = numpy.array(a)
print ar + 2
```

gives

[3, 3, 3, 3, 3]

There are many resources on the internet how numpy arrays actually work. E.g. see

https://s3.amazonaws.com/assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf

Exercise 1.3.

This exercise will try to build a little intuition about the cost function.

Now we are going to find a hypothesis

Hypothesis:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Parameters:

$$\theta_0, \theta_1$$

that minimizes the cost function:

Cost Function:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

Goal: $\underset{\theta_0, \theta_1}{\text{minimize}}\ J(\theta_0, \theta_1)$

Given X and y as in exercise 1.2, then in this case we assume that we should look for theta0 in say range [1,10] and theta1 in range [0.5,5]...

We can then calculate the cost function for these theta values.

There are many ways to calculate the cost function in Python, below is one version using numpy.

```
def cost(a,b,X,y):
    ### Evaluate half MSE (Mean square error)
    m = len(y)
    error = a + b*X - y
    J = np.sum(error ** 2)/(2*m)
    return J
```

You can use this version of the method, or you can implement your own. In either way, verify that it works as expected.

You can then make some intervals to investigate:

```
ainterval = np.arange(1,10, 0.05)
binterval = np.arange(0.5,5, 0.05)
```

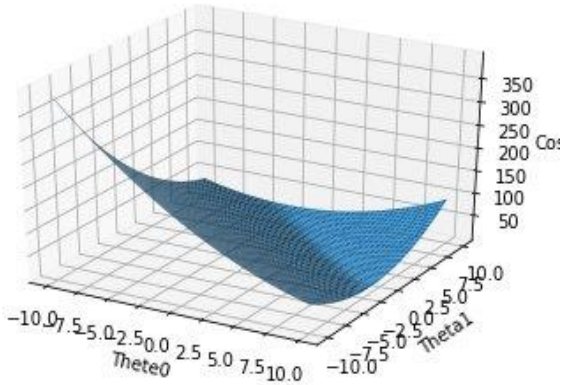
Stepping through them like

```
for atheta in ainterval:
    for btheta in binterval:
        print("xy: %f:%f:%f" % (atheta,btheta,cost(atheta,btheta, X, y)))
```

What do you find? What are the best values for theta ? Try to experiment with the parameters, change the intervals, and the step-value from 0.05 to higher and smaller values.

Exercise 1.4.

It is often helpful to see a plot of the cost function, in order to get an intuition of where local minimums might be located.



It it possible to plot the cost function in most computer languages (given that relavant libraries are installed). You are free to pick the tools you are most familiar with.

(And) you can also try it out in Python. See the code below for inspiration (remember that different versions of Python plot libraries might not work exactly the same). Make small alterations, improvements, and see if you can figure out what it is supposed to do?

I.e make a number of plots, so that allow you to “zoom” in on the solution.

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

def cost(a,b):
    ### Evaluate half MSE (Mean square error)
    m = len(Ydots)
    error = a + b*Xdots - Ydots
    J = np.sum(error ** 2)/(2*m)
    return J
```

```
Xdots = 2 * np.random.rand(100, 1)
Ydots = -5 + 7 * Xdots + np.random.randn(100, 1)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ainterval = np.arange(-10,10, 0.05)
binterval = np.arange(-10,10, 0.05)

X, Y = np.meshgrid(ainterval, binterval)
zs = np.array([cost(x,y) for x,y in zip(np.ravel(X), np.ravel(Y))])
Z = zs.reshape(X.shape)

ax.plot_surface(X, Y, Z)
ax.set_xlabel('Thete0')
ax.set_ylabel('Theta1')
ax.set_zlabel('Cost')
plt.show()
```

Exercise 1.5.

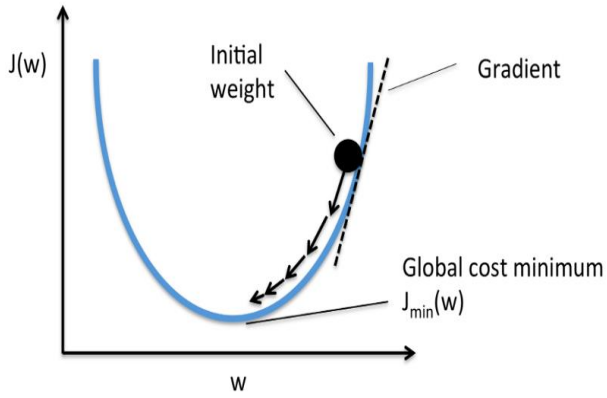
Still,it would be nice if we had some method to find our theta0 and theta1 precisely.

Enters gradient descent.

The math behind it isn’t terribly complicated. But don't worry, you will just be given the formulas to work with. What we’re doing here is applying partial derivatives with respect to both theta0 and theta1 to the cost function to point us to the lowest point.

If you remember your math, a derivative of zero means you are at either a local minima or maxima. Which means that the closer we get to zero, the better. When we reach close to, if not, zero with our derivatives, we also inevitably get the lowest value for our cost function.

In case you wonder if there couldn't be more loal minima in the cost function, then take a closer look at the plot of the cost function in the previous exercise, here this appears not to be the case. Again we won't prove this, but it looks ok.



So, we need an update function for our theta values. Something like this:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad \text{(for } j = 0 \text{ and } j = 1\text{)}$$

Here, alfa will be our learning rate.

All we need now is to know what the derivatives are.

Looking around on the internet, it turns out a lot of friendly people have worked this out for us:

$$\theta_0,\ j = 0 : \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

$$\theta_1,\ j = 1 : \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

In Python that would give us gradients (something) like this:

```
theta0_gradient = (2/N) * sum(y_hypothesis - y)

theta1_gradient = (2/N) * sum(X * (y_hypothesis - y))
```

Where we would update our existing values like this:

```
theta0_current = theta0_current - (learning_rate * theta0_gradient)

theta1_current = theta1_current - (learning_rate * theta1_gradient)
```

(Why do you think there is a minus sign here? What happens if you switches the sign in front of the learning rate?)

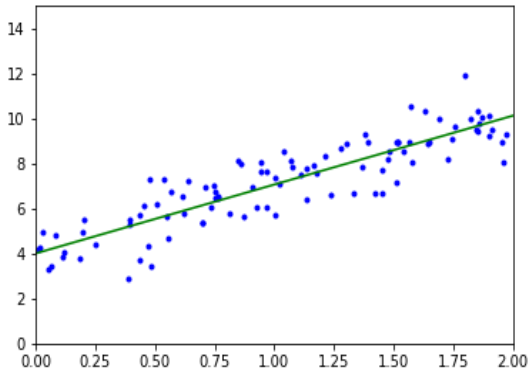
This should give us a python program looking something like this:

```
import matplotlib.pyplot as plt
import numpy as np
def linear_regression(X, y, theta0=0, theta1=0, epochs=10000, learning_rate=0.0001):
    N = float(len(y))
```

```
    for i in range(epochs):
        y_hypothesis = (theta1 * X) + theta0
        cost = sum([data**2 for data in (y-y_hypothesis)]) / N
        theta1_gradient = -(2/N) * sum(X * (y - y_hypothesis))
        theta0_gradient = -(2/N) * sum(y - y_hypothesis)
        theta0 = theta0 - (learning_rate * theta0_gradient)
        theta1 = theta1 - (learning_rate * theta1_gradient)

    return theta0, theta1, cost

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

theta0, theta1, cost = linear_regression(X, y, 0, 0, 1000, 0.01)

plt.plot(X,y, "b.")
plt.axis([0,2,0,15])

# lets plot that line:
#X_new = np.array([[0],[2]])
#X_new_b = np.c_[np.ones((2,1)), X_new]
#y_predict = X_new_b.dot([theta0, theta1])
#plt.plot(X_new, y_predict, "g-")
```

Try to understand what is going on in this program. Run it, and see if you get the expected values?

Also, try to experiment with the datasize, and learning rates. Again, does it work as expected?



Exercise 1.6.

There is also a mathematical equation that gives the result directly.

See p. 116 (v2.) in the “Hands-on Machine Learning ” book:

Equation 4-4. Normal Equation

$$\hat{\theta} = \left(\mathbf{X}^T \cdot \mathbf{X}\right)^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

Don't worry about the math, we will just take this as a given here. The book also tell us what this will (can) look like in Python.

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

X = 2 * np.random.rand(50, 1)
y = 4 + 3 * X + np.random.randn(50, 1)

X_b = np.c_[np.ones((50, 1)), X]
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

X_new = np.array([[0],[2]])
X_new_b = np.c_[np.ones((2,1)), X_new]
y_predict = X_new_b.dot(theta_best)

plt.plot(X,y, "g.")
plt.axis([0,2,0,15])
plt.plot(X_new, y_predict, "r-")

plt.plot()
plt.show()
```

Again, try to go through the steps, and see that we end up with something that looks right. Here, it is fine if you just have a small gist of what is going on here.

Using Scikit-learn makes it even easier.

```
from sklearn import linear_model
import matplotlib.pyplot as plt
import numpy as np

X = 2 * np.random.rand(500, 1)
y = 4 + 3 * X + np.random.randn(500, 1)

lm = linear_model.LinearRegression()
model = lm.fit(X,y)

plt.plot(X,y, "g.")
plt.axis([0,2,0,15])
```

```
#fit function
f = lambda x: lm.coef_*x + lm.intercept_
plt.plot(X,f(X), c="red")
plt.plot()
plt.show()
```

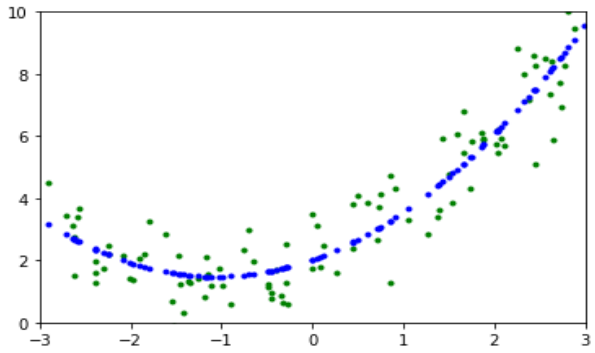
As usual, go through the steps, and verify that this works as expected.

Exercise 1.7.

Try to work through the example with polynomial regression p.130 (v2) in the “Hands-On ML” book, and see that it is also possible to make models that fit more complex data than the straight line we have worked with so far.

Something along these lines?

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

from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

X = 6 * np.random.rand(100, 1) - 3
y = 0.5 * X * X + X + 2 + np.random.randn(100, 1)

plt.plot(X,y, "g.")
plt.axis([-3,3,0,10])

poly_features = PolynomialFeatures(2, include_bias=False)
X_poly = poly_features.fit_transform(X)

lm = LinearRegression()
lm.fit(X_poly, y)

#fit function
f = lambda x: lm.coef_[0][1]*x*x + lm.coef_[0][0]*x + lm.intercept_
plt.plot(X,f(X), "b.")

plt.show()
```



And while you are at it, why not try with a 3 degree polynomial? Redoing this exercise with some points, where it is a 3 dimensional polynomial that give us the best fit.

Y = a * x^3 + b * x^2 + c*x + d