EE6310

Problem Set 7

Following the last problem set, this problem set will guide you to write the multiple linear regression algorithm in python like the last time we did with the linear regression. You can recycle the codes from the last homework! Below are the data points.

Name	ΔH_f [eV]	# CH_3	# CH_2	# CH	# C
ethane	-0.87	2	0	0	0
propane	-1.08	2	1	0	0
butane	-1.30	2	2	0	0
pentane	-1.52	2	3	0	0
hexane	-1.73	2	4	0	0
heptane	-1.95	2	5	0	0
octane	-2.16	2	6	0	0
nonane	-2.37	2	7	0	0
decane	-2.59	2	8	0	0
undecane	-2.80	2	9	0	0
dodecane	-3.01	2	10	0	0
tridecane	-3.23	2	11	0	0
tetradecane	-3.44	2	12	0	0
pentadecane	-3.68	2	13	0	0
hexadecane	-3.89	2	14	0	0
heptadecane	-4.08	2	15	0	0
octadecane	-4.30	2	16	0	0
nonadecane	-4.51	2	17	0	0
icosane	-4.72	2	18	0	0
dotriacontane	-7.22	2	30	0	0
2-methylpropane	-1.41	3	0	1	0
2-methylbutane	-1.59	3	1	1	0
2-methylpentane	-1.81	3	2	1	0
3-methylpentane	-1.78	3	2	1	0
2-methylhexane	-2.02	3	3	1	0
3-methylhexane	-1.99	3	3	1	0
3-ethylpentane	-1.97	3	3	1	0
2-methylheptane	-2.23	3	4	1	0
3-methylheptane	-2.20	3	4	1	0
4-methylheptane	-2.20	3	4	1	0
3-ethylhexane	-2.19	3	4	1	0
2-methylnonane	-2.70	3	6	1	0
5-methylnonane	-2.68	3	6	1	0
11-butyldocosane	-6.15	3	22	1	0
5-butyldocosane	-6.09	3	22	1	0
11-decylhenicosane	-7.25	3	27	1	0
2,2-dimethylpropane	-1.74	4	0	0	1
2,2-dimethylbutane	-1.92	4	1	0	1
2,3-dimethylbutane	-1.84	4	0	2	0

2,2-dimethylpentane	-2.14	4	2	0	1
2,3-dimethylpentane	-2.06	4	1	2	0
2,4-dimethylpentane	-2.09	4	1	2	0
3,3-dimethylpentane	-2.09	4	2	0	1
2,2-dimethylhexane	-2.33	4	3	0	1
2,3-dimethylhexane	-2.22	4	2	2	0
2,4-dimethylhexane	-2.27	4	2	2	0
2,5-dimethylhexane	-2.31	4	2	2	0
3,3-dimethylhexane	-2.28	4	3	0	1
3,4-dimethylhexane	-2.21	4	2	2	0
3-ethyl-2-methylpentane	-2.19	4	2	2	0
3-ethyl-3-methylpentane	-2.23	4	3	0	1
2,2-dimethylheptane	-2.55	4	4	0	1
2,2,3-trimethylbutane	-2.12	5	0	1	1
2,2,3-trimethylpentane	-2.28	5	1	1	1
2,2,4-trimethylpentane	-2.32	5	1	1	1
2,3,3-trimethylpentane	-2.24	5	1	1	1
2,3,4-trimethylpentane	-2.25	5	0	3	0
2,2,5-trimethylhexane	-2.62	5	2	1	1
2,3,5-trimethylhexane	-2.51	5	1	3	0
3-ethyl-2,4-dimethylpentane	-2.36	5	1	3	0
2,2,3,3-tetramethylbutane	-2.34	6	0	0	2
2,2,4,4-tetramethylpentane	-2.50	6	1	0	2
2,2,5,5-tetramethylhexane	-2.95	6	2	0	2
3,3,4,4-tetraethylhexane	-2.75	6	6	0	2
3-tert-butyl-2,2,4,4-tetramethylpentane	-2.45	9	0	1	3
2,2,3,3,4,4,5,5-octamethylhexane	-2.57	10	0	0	4
3,4-ditert-butyl-2,2,5,5-tetramethylhexane	-2.60	12	0	2	4

In python code, the following code can be copied and pasted to make a numpy array.

```
import numpy as np
data = np.array([[-0.87,2,0,0,0],[-1.08,2,1,0,0],[-1.30,2,2,0,0],[-
1.52,2,3,0,0],[-1.73,2,4,0,0],[-1.95,2,5,0,0],[-2.16,2,6,0,0],[-
2.37,2,7,0,0],[-2.59,2,8,0,0],[-2.80,2,9,0,0],[-3.01,2,10,0,0],[-
3.23,2,11,0,0],[-3.44,2,12,0,0],[-3.68,2,13,0,0],[-3.89,2,14,0,0],[-
4.08,2,15,0,0],[-4.30,2,16,0,0],[-4.51,2,17,0,0],[-4.72,2,18,0,0],[-
7.22,2,30,0,0],[-1.41,3,0,1,0],[-1.59,3,1,1,0],[-1.81,3,2,1,0],[-
1.78,3,2,1,0],[-2.02,3,3,1,0],[-1.99,3,3,1,0],[-1.97,3,3,1,0],[-
2.23,3,4,1,0],[-2.20,3,4,1,0],[-2.20,3,4,1,0],[-2.19,3,4,1,0],[-
2.70,3,6,1,0],[-2.68,3,6,1,0],[-6.15,3,22,1,0],[-6.09,3,22,1,0],[-
7.25,3,27,1,0],[-1.74,4,0,0,1],[-1.92,4,1,0,1],[-1.84,4,0,2,0],[-
2.14,4,2,0,1],[-2.06,4,1,2,0],[-2.09,4,1,2,0],[-2.09,4,2,0,1],[-
2.33,4,3,0,1],[-2.22,4,2,2,0],[-2.27,4,2,2,0],[-2.31,4,2,2,0],[-
2.28,4,3,0,1],[-2.21,4,2,2,0],[-2.19,4,2,2,0],[-2.23,4,3,0,1],[-
2.55,4,4,0,1],[-2.12,5,0,1,1],[-2.28,5,1,1,1],[-2.32,5,1,1,1],[-
2.24,5,1,1,1],[-2.25,5,0,3,0],[-2.62,5,2,1,1],[-2.51,5,1,3,0],[-
2.36,5,1,3,0],[-2.34,6,0,0,2],[-2.50,6,1,0,2],[-2.95,6,2,0,2],[-
2.75,6,6,0,2],[-2.45,9,0,1,3],[-2.57,10,0,0,4],[-2.60,12,0,2,4]])
```

Follow the steps below to make the linear regression algorithm. Let's get started!

(a) Use the slicing to separate the data into *x* (feature) and *y* (target). In the data array, the first column contains the *y*, and the second to the last column contains the *y*. The answer should have a form of:

```
x = …
y = …
```

(b) Define the function that predicts the $\hat{y}$ given $\boldsymbol{w}$, b, and $\boldsymbol{x}$. Remember the equation:

$$f_{w,b}(x) = \hat{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b$$

The input to this function are $\boldsymbol{w}$, b, and $\boldsymbol{x}$. First use the for loop to iterate over each training example. The function should have a form below.

```
def linear(w,b,x):
    # your code
    return y_hat
```

If the code is correct, you should get the result in the comment (written in ''')

```
yhat = linear([0.5,0.1,0.2,0.3],-8,x)
print(yhat)
'''
output is
[-7.  -6.9 -6.8 -6.7 -6.6 -6.5 -6.4 -6.3 -6.2 -6.1 -6.  -5.9 -5.8 -
5.7
 -5.6 -5.5 -5.4 -5.3 -5.2 -4.  -6.3 -6.2 -6.1 -6.1 -6.  -6.  -6.  -
5.9
 -5.9 -5.9 -5.9 -5.7 -5.7 -4.1 -4.1 -3.6 -5.7 -5.6 -5.6 -5.5 -5.5 -
5.5
 -5.5 -5.4 -5.4 -5.4 -5.4 -5.4 -5.4 -5.4 -5.4 -5.3 -5.  -4.9 -4.9 -
4.9
 -4.9 -4.8 -4.8 -4.8 -4.4 -4.3 -4.2 -3.8 -2.4 -1.8 -0.4]
'''
```

(c) In the class we learned about np.dot to perform $\boldsymbol{w} \cdot \boldsymbol{x}$. However, we still have to use for loop over each training example. Turns out, we can use matrix multiplication to further vectorize and speed up the operation. The feature vector $\boldsymbol{x}$, has a dimension of $\boldsymbol{x} \in \mathbb{R}^n$, where n is the number of feature. Let's define a new variable $\boldsymbol{X} \in \mathbb{R}^{m \times n}$. $\boldsymbol{X}$ is a 2D matrix where each row indicates a training example, with the n columns indicating features. Now, we have a weight vector, $\boldsymbol{w} \in \mathbb{R}^n$. We can multiply the each row of $\boldsymbol{X}$ with $\boldsymbol{w}$ and sum up by each row, by $(\boldsymbol{X}\boldsymbol{w}) \in \mathbb{R}^m$, as the inner dimension n disappears. Once the matrix multiplication is completed, the bias can be added by broadcasting. Essentially, we are

calculating the following:

$$f_{\boldsymbol{w},b}(\boldsymbol{X}) = \boldsymbol{X}\boldsymbol{w} + b$$

Use this information and further vectorize the for loop.

(d) With this vectorized linear function, you can still use the cost function from problem set 6 to calculate the cost function. Now, it's time to vectorize the gradient descent. Let's bring out the equation

$$w_j = w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\boldsymbol{w},b}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) x_j^{(i)}$$

$$b = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\boldsymbol{w},b}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)$$

We can also vectorize these equations through several steps. First, let's define residual $\boldsymbol{R} \in \mathbb{R}^m$ which corresponds to the vectorized version of $f_{\boldsymbol{w},b}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}$:

$$\boldsymbol{R} = \boldsymbol{X}\boldsymbol{w} + b - \boldsymbol{Y} = f_{\boldsymbol{w},b}(\boldsymbol{X}) - \boldsymbol{Y}$$

$\boldsymbol{R}$ is an abbreviation of residual. The $\boldsymbol{w}$ can be updated by:

$$\boldsymbol{w} = \boldsymbol{w} - \frac{\alpha}{m}\boldsymbol{R}\boldsymbol{X}$$

The matrix multiplication between $\boldsymbol{R} \in \mathbb{R}^m$ and $\boldsymbol{X} \in \mathbb{R}^{m \times n}$ result in $(\boldsymbol{R}\boldsymbol{X}) \in \mathbb{R}^n$ as the inner dimension m disappears by the summation. Note that we are supposed to take the mean, so we need to divide $\boldsymbol{R}\boldsymbol{X}$ by m. We can update the b by:

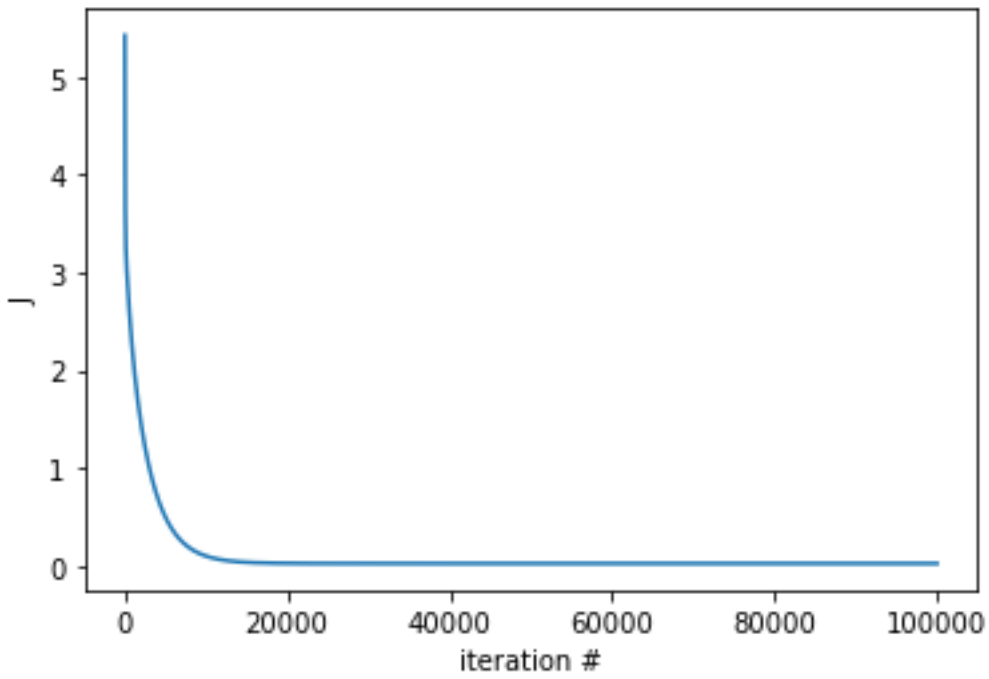
$$b = b - \alpha\bar{\boldsymbol{R}}$$

where the bar over $\boldsymbol{R}$ indicates taking the mean. Note that we don't need to have temporary $\boldsymbol{w}$ and b as we calculate the $\boldsymbol{R}$ before updating $\boldsymbol{w}$ and b (Think about why this is the case). Use the vectorized version of the gradient descent above to write the gradient descent code.

```
def gradient_descent(w,b,x,y,alpha):
    # your code
    return w,b
```

If your code is correct, you should get a figure shown below up running the code below:

```
w = [0.5,0.1,0.2,0.3]
b = -8
J = []
for i in range(100000):
    w,b = gradient_descent(w,b,x,y,0.001)
    J.append(cost_function(w,b,x,y))
plt.plot(np.arange(len(J)),J)
plt.xlabel('iteration #')
plt.ylabel('J')
plt.show()
```



(e) If you have stuck through, great work! Let's implement the convergence criterion $\varepsilon = 10^{-11}$ where if the J decreases by less than ε, the gradient descent stops. Simply modify the code block above. At what iteration, does the change in J becomes less than 10^{-11}?

(f) Now let's implement the feature scaling (z-score normalization) and see how many iterations are needed for J to change less than 10^{-11}. Does it converge faster than when the feature scaling was not performed? Remember to scale each feature individually!