CSC420 Assignment1

Zilun Zhang 1001175685

September 25 2017

1

(1) **Load the Boston housing data from the sklearn datasets module**

Please see the code file "q1.py". In order to let test result repeatable, I will use the result of setting random seed equals to 0. I will comment the line which sets random seed equals to 0 for the hand in version. If you want to repeat the result, uncomment it will be fine.

(2) **Describe and summarize the data in terms of number of data points, dimensions, target, etc**

Please see the result of code file "q1.py". I use pandas package to summarize data for each set (X, y).

```
Sample Of X Summary
                0           1           2           3           4           5  \
count  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000   
mean     3.593761   11.363636   11.136779    0.069170    0.554695    6.284634   
std      8.596783   23.322453    6.860353    0.253994    0.115878    0.702617   
min      0.006320    0.000000    0.460000    0.000000    0.385000    3.561000   
25%      0.082045    0.000000    5.190000    0.000000    0.449000    5.885500   
50%      0.256510    0.000000    9.690000    0.000000    0.538000    6.208500   
75%      3.647423   12.500000   18.100000    0.000000    0.624000    6.623500   
max     88.976200  100.000000   27.740000    1.000000    0.871000    8.780000   

                6           7           8           9          10          11  \
count  506.000000  506.000000  506.000000  506.000000  506.000000  506.000000   
mean    68.574901    3.795043    9.549407  408.237154   18.455534  356.674032   
std     28.148861    2.105710    8.707259  168.537116    2.164946   91.294864   
min      2.900000    1.129600    1.000000  187.000000   12.600000    0.320000   
25%     45.025000    2.100175    4.000000  279.000000   17.400000  375.377500   
50%     77.500000    3.207450    5.000000  330.000000   19.050000  391.440000   
75%     94.075000    5.188425   24.000000  666.000000   20.200000  396.225000   
max    100.000000   12.126500   24.000000  711.000000   22.000000  396.900000   

                12  
count  506.000000  
mean    12.653063  
std      7.141062  
min      1.730000  
25%      6.950000  
50%     11.360000  
75%     16.955000  
max     37.970000  
----------------------------
Sample Of y Summary
                 0
count  506.000000
mean    22.532806
std      9.197104
min      5.000000
25%     17.025000
50%     21.200000
75%     25.000000
max     50.000000
----------------------------
```

To look at the weight fairly, I normalized data.

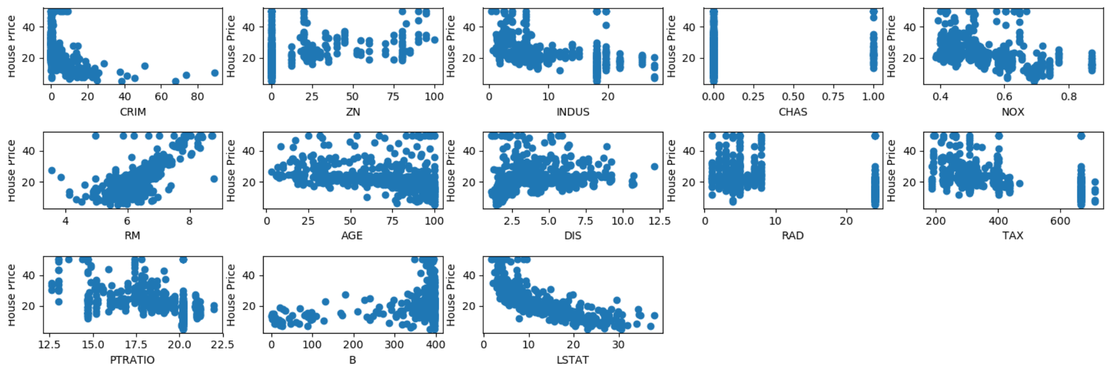
Here is the data summary **after normalization**:

```
Sample Of X Summary
                  0             1             2             3             4  \
count  5.060000e+02  5.060000e+02  5.060000e+02  5.060000e+02  5.060000e+02
mean   6.340997e-17 -6.343191e-16 -2.682911e-15  4.701992e-16  2.490322e-15
std    1.000990e+00  1.000990e+00  1.000990e+00  1.000990e+00  1.000990e+00
min   -4.177134e-01 -4.877224e-01 -1.557842e+00 -2.725986e-01 -1.465882e+00
25%   -4.088961e-01 -4.877224e-01 -8.676906e-01 -2.725986e-01 -9.130288e-01
50%   -3.885818e-01 -4.877224e-01 -2.110985e-01 -2.725986e-01 -1.442174e-01
75%    6.248255e-03  4.877224e-02  1.015999e+00 -2.725986e-01  5.986790e-01
max    9.941735e+00  3.804234e+00  2.422565e+00  3.668398e+00  2.732346e+00

                  5             6             7             8             9  \
count  5.060000e+02  5.060000e+02  5.060000e+02  5.060000e+02  5.060000e+02
mean  -1.145230e-14 -1.407855e-15  9.210902e-16  5.441409e-16 -8.868619e-16
std    1.000990e+00  1.000990e+00  1.000990e+00  1.000990e+00  1.000990e+00
min   -3.880249e+00 -2.335437e+00 -1.267069e+00 -9.828429e-01 -1.313990e+00
25%   -5.686303e-01 -8.374480e-01 -8.056878e-01 -6.379618e-01 -7.675760e-01
50%   -1.084655e-01  3.173816e-01 -2.793234e-01 -5.230014e-01 -4.646726e-01
75%    4.827678e-01  9.067981e-01  6.623709e-01  1.661245e+00  1.530926e+00
max    3.555044e+00  1.117494e+00  3.960518e+00  1.661245e+00  1.798194e+00

                 10            11            12
count  5.060000e+02  5.060000e+02  5.060000e+02
mean  -9.205636e-15  8.163101e-15 -3.370163e-16
std    1.000990e+00  1.000990e+00  1.000990e+00
min   -2.707379e+00 -3.907193e+00 -1.531127e+00
25%   -4.880391e-01  2.050715e-01 -7.994200e-01
50%    2.748590e-01  3.811865e-01 -1.812536e-01
75%    8.065758e-01  4.336510e-01  6.030188e-01
max    1.638828e+00  4.410519e-01  3.548771e+00
----------------------------
Sample Of y Summary
                0
count  506.000000
mean    22.532806
std      9.197104
min      5.000000
25%     17.025000
50%     21.200000
75%     25.000000
max     50.000000
----------------------------
```

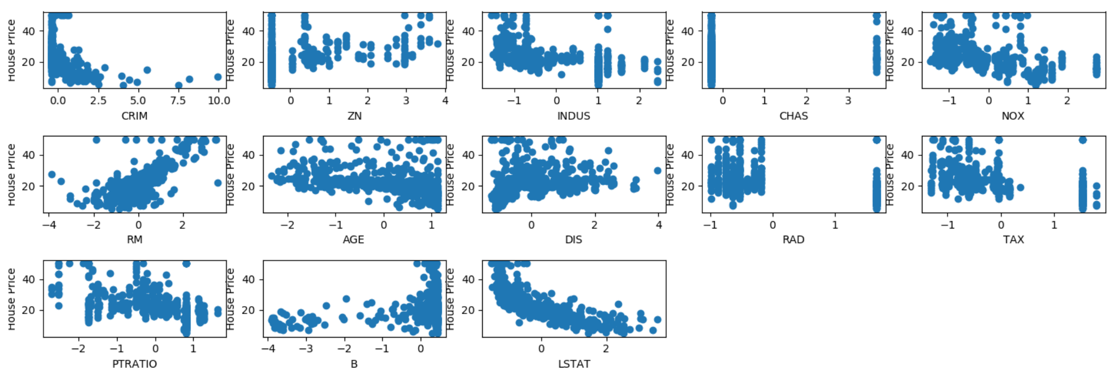
Here is the code of normalization:

```python
def normalize_data(X):
    matrix_X_substract_mean = X - np.mean(X, axis=0, keepdims=True)
    matrix_X_std = np.power(help_var(X), 0.5)
    normal_X = matrix_X_substract_mean/matrix_X_std
    return normal_X


def cal_var_vector(g):
    mean = g.mean(0)
    var_vector = np.zeros((1, g.shape[1]), dtype=float)
    for i in range(g.shape[0]):
        var_vector += np.power(g[i, :] - mean, 2)
    result = np.divide(var_vector, (g.shape[0]))
    return result


# calculate var along axis 0 for matrix.
def help_var(X):
    # loop features
    var_list = []
    for feature in range(X.shape[1]):
        column = X[:, feature]
        column = column.reshape(506, 1)
        var = cal_var_vector(column)
        var_list.append(var)
    return np.array(var_list).reshape(1, X.shape[1])
```

(3) **Visualization: present a single grid containing plots for each feature against the target. Choose the appropriate axis for dependent vs. independent variables.**



Here is the scatter graph of data **after normalization**.



Code:

```python
def visualize(X, y, features):
    plt.figure(figsize=(20, 5))
    feature_count = X.shape[1]
    # i: index
    for i in range(feature_count):
        plt.subplot(3, 5, i + 1)
        # Plot feature i against y
        plt.scatter(X[:, i], y)
        plt.xlabel(features[i])
        plt.ylabel("House Price")
    plt.tight_layout()
    plt.show()
```

(4) **Divide your data into training and test sets, where the training set consists of 80 % of the data points(chosen at random).**

Code:

```python
def split(X, y, train_split_rate):
    feature_count, data_X_count, train_X_count = X.shape[1], X.shape[0], ceil(X.shape[0] * train_split_rate)
    train_X = []
    test_X = []
    train_y = []
    test_y = []
    training_index = np.random.choice(data_X_count, int(train_X_count), replace=False)
    for index in range(data_X_count):
        if index in training_index:
            train_X.append(X[index])
            train_y.append(y[index])
        else:
            test_X.append(X[index])
            test_y.append(y[index])

    train_X = np.array(train_X)
    train_y = np.array(train_y)
    test_X = np.array(test_X)
    test_y = np.array(test_y)
    print("train set of X is: " + str(train_X.shape))
    print("train set of y is: " + str(train_y.shape))
    print("test set of X is: " + str(test_X.shape))
    print("test set of y is: " + str(test_y.shape))
    print()
    return train_X, train_y, test_X, test_y
```

(5) **Write code to perform linear regression to predict the targets using the training data. Remember to add a bias term to your model.**

Code:

```python
def fit_regression(X,Y):
    # implement linear regression
    # Remember to use np.linalg.solve instead of inverting!
    # add bias term
    feature_count = X.shape[1]
    X = np.insert(X, feature_count, 1, axis=1)
    # (X^T*X)W = X^T*y
    left = np.dot(X.T, X)
    right = np.dot(X.T, Y)
    # solve w
    return np.linalg.solve(left, right)
```

```python
def get_predict_value(w, X):
    feature_count = X.shape[1]
    # add bias term
    X = np.insert(X, feature_count, 1, axis=1)
    return np.dot(X, w)
```

(6) **Tabulate each feature along with its associated weight and present them in a table. Explain what the sign of the weight means in the third column ('INDUS') of this table. Does the sign match what you expected? Why?**

Before normalization:

CRIM	-0.109703
ZN	0.041546
INDUS	0.010951
CHAS	1.935669
NOX	-17.866761
RM	3.280849
AGE	0.004475
DIS	-1.385640
RAD	0.366823
TAX	-0.015388
PTRATIO	-0.894088
B	0.008872
LSTAT	-0.553587
BIAS	39.570475

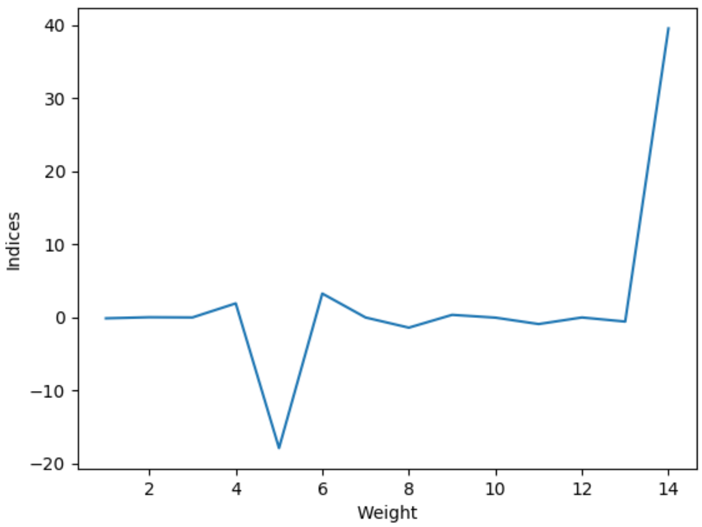
The sign of 'INDUS' is positive, which is: 0.010951, which makes sense because more proportion of non-retail business acres per town means less bussiness acres per town, which let business acres become more valued, therefore the retail housing price will increase."

After normalization:

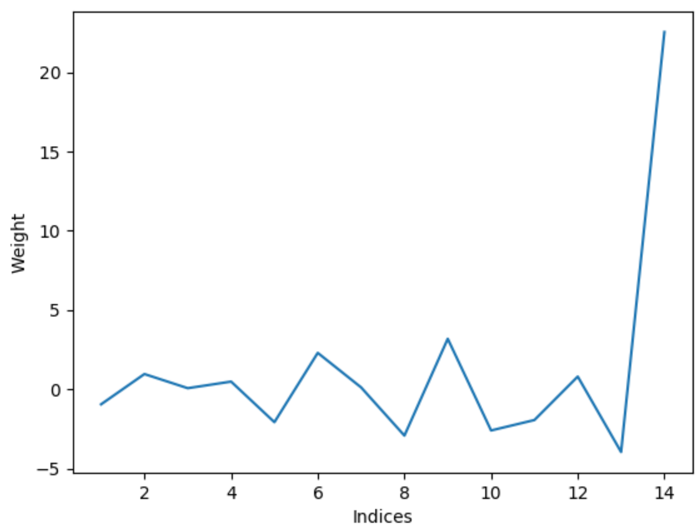
CRIM	-0.942165
ZN	0.968004
INDUS	0.075057
CHAS	0.491162
NOX	-2.068312
RM	2.302902
AGE	0.125840
DIS	-2.914872
RAD	3.190866
TAX	-2.590951
PTRATIO	-1.933739
B	0.809196
LSTAT	-3.949287
BIAS	22.540751

below is a graph of weights against indices:

Before normalization:



After normalization:



(7) **Test the fitted model on your test set and calculate the Mean Square Error of the result.**

MSE of model by using test data before normalization is: 16.5753061192.

MSE of model by using test data **after normalization** is: 16.5753061192.

Code:

```python
def mse(predict_value, test_value):
    # print(predict_value, test_value)
    return np.mean(np.power(test_value-predict_value, 2))
```

(8) **Suggest and calculate more error measurement metrics; justify your choice.**

Suggested norm 1 error(abs value), r square value and RMSE error, which are implemented on q1.py file.

Norm 1 loss of model before normalization is: 0.02999964327.

Norm 1 loss of model **after normalization** is: 0.02999964327.

R square coefficient of model before normalization is: 0.83951752255

R square coefficient of model **after normalization** is: 0.83951752255

RMSE loss before normalization is: 4.071278192308554.

RMSE loss **after normalization** is: 4.0712781923084265.

Code:

```python
def norm1_loss(test_value, predict_value):
    result_vector = predict_value - test_value
    return np.mean(np.abs(result_vector))/test_value.shape[0]
```

```python
def r_square_coeff(y, y_predict):
    ss_total = cal_var_vector(y.reshape(y,shape[0], 1))*int(y.shape
    [0])
    ss_res = mse(y, y_predict)*int(y.shape[0])
    return float(1 - np.divide(ss_res, ss_total))
```

```python
def rmse_loss(mse):
    return sqrt(mse)
```

(9) **Feature Selection: Based on your results, what are the most significant features that best predict the price? Justify your answer.**

Most significant feature to predict the price is LSTAT after normalization, which makes sense because more % lower status of the population in town means less quality of population in town, less opportunity that businessmen investment in that area, less infrastructure and less security in that area therefore the housing price will decrease in that area. It has negative effect for housing price, that's why it has a negative weight.

Also, RAD is important, it has the largest positive weight above all features. Index of accessibility to radial highways represent how developed and convenient a region. Obviously, more RAD means this region is more developed, therefore the housing price will go high. It has positive effect for housing price, that is why it has a positive weight.

2

(1)

$$
\begin{aligned}
L(\underline{w}) &= \frac{1}{2} \| A^{\frac{1}{2}} (y - X\underline{w}) \|^2 + \frac{\lambda}{2} \|\underline{w}\|^2 \\
&= \frac{1}{2} (A^{\frac{1}{2}} (y - X\underline{w}))^T (A^{\frac{1}{2}} (y - X\underline{w})) + \frac{\lambda}{2} \underline{w}^T \underline{w}. \\
&= \frac{1}{2} (A^{\frac{1}{2}} y - A^{\frac{1}{2}} X\underline{w})^T (A^{\frac{1}{2}} y - A^{\frac{1}{2}} X\underline{w}) + \frac{\lambda}{2} \underline{w}^T \underline{w} \\
&= \frac{1}{2} (y^T A^{\frac{1}{2}} - \underline{w}^T X^T A^{\frac{1}{2}}) (A^{\frac{1}{2}} y - A^{\frac{1}{2}} X\underline{w}) + \frac{\lambda}{2} \underline{w}^T \underline{w}. \\
&= \frac{1}{2} \Big(y^T A^{\frac{1}{2}} \cdot A^{\frac{1}{2}} y - y^T A^{\frac{1}{2}} A^{\frac{1}{2}} X\underline{w} \\
&\qquad - \underline{w}^T X^T A^{\frac{1}{2}} A^{\frac{1}{2}} y + \underline{w}^T X^T A^{\frac{1}{2}} A^{\frac{1}{2}} X\underline{w} \Big) \\
&\qquad + \frac{\lambda}{2} \underline{w}^T \underline{w} \\
&= \frac{1}{2} (y^T A y - y^T A X \underline{w} - \underline{w}^T X^T A y + \underline{w}^T X^T A X \underline{w}) \\
&\qquad + \frac{\lambda}{2} \underline{w}^T \underline{w} \\
&= \frac{1}{2} (y^T A y + \underline{w}^T X^T A X \underline{w} - 2 \underline{w}^T X^T A y) + \frac{\lambda}{2} \underline{w}^T \underline{w}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial \underline{w}} &= \frac{1}{2} (2 X^T A X \underline{w} - 2 X^T A y) + \lambda \underline{w} \\
&= X^T A X \underline{w} - X^T A y + \lambda \underline{w}
\end{aligned}
$$

$$\frac{\partial L}{\partial \underline{w}} = 0 \implies \quad X^T A X \underline{w} + \lambda \underline{w} = X^T A y$$

$$(X^T A X + \lambda I)\, \underline{w} = X^T A y$$

$$\underline{w}^* = (X^T A X + \lambda I)^{-1} X^T A y.$$

(2) Please see the file q2.py

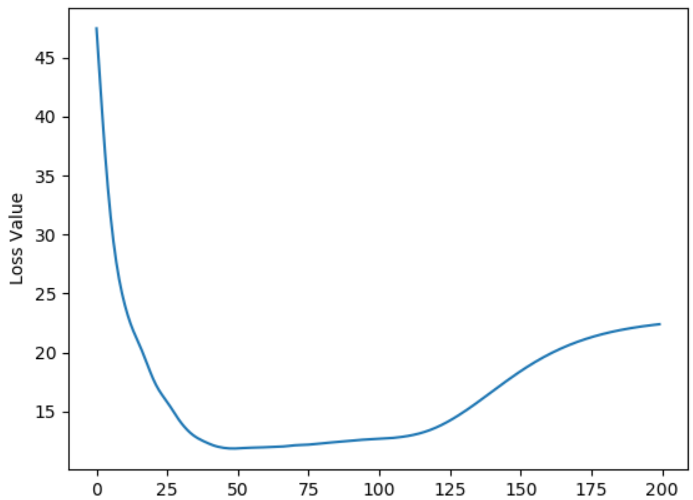
Code:

```
def LRLS(test_datum, x_train, y_train, tau, lam=1e-5):
    '''
    Input: test_datum is a dx1 test vector
           x_train is the N_train x d design matrix
           y_train is the N_train x 1 targets vector
           tau is the local reweighting parameter
           lam is the regularization parameter
    output is y_hat the prediction on test_datum
    '''

    # (X^T*A*X+lamda*I)W = X^T*A*Y
    # build a empty list contain diagonal of A.
    x_train_N = x_train.shape[0]
    x_train_d = x_train.shape[1]
    A = np.zeros((x_train_N, x_train_N), dtype=float)
    identity_matrix = np.identity(x_train_d)
    content_numerator = -l2(x_train, test_datum.T)
    content_denominator = np.multiply(2, np.power(tau, 2))
    numerator = np.divide(content_numerator, content_denominator)
    log_summation = misc.logsumexp(numerator)
    denominator = np.exp(log_summation)

    for i in range(x_train_N):
        A[i, i] = np.exp(numerator[i])/denominator

    temp = np.matmul(x_train.T, A)
    left_temp = np.matmul(temp, x_train)
    left = left_temp + np.multiply(lam, identity_matrix)
    right = np.dot(temp, y_train)
    w = np.linalg.solve(left, right)
    y_hat = np.matmul(test_datum.T, w)
    return float(y_hat)
```

```
def run_k_fold(x, y, taus, k):
    '''
```

```
    Input: x is the N x d design matrix
           y is the N x 1 targets vector
           taus is a vector of tau values to evaluate
           K in the number of folds
    output is losses a vector of k-fold cross validation losses one for each tau value
    '''
    loss = []
    # random process
    concatenate_matrix = np.concatenate((x, y[:, None]), axis=1)
    np.random.shuffle(concatenate_matrix)
    print(concatenate_matrix[0, 0])
    # split to k fold
    temp_container = np.array_split(concatenate_matrix, k)
    i = 0
    while i < k:
        print("fold number is:")
        print(i+1)
        fold = np.array(temp_container[i])
        x_test = fold[:, :d]
        print("x_test size is:")
        print(x_test.shape)
        y_test = fold[:, d]
        print("y_test size is:")
        print(y_test.shape)
        x_train = []
        y_train = []
        for j in range(k):
            if j == i:
                pass
            else:
                other_fold = np.array(temp_container[j])
                print("other_fold shape is:")
                print(other_fold.shape)
                x_train.append(other_fold[:, :d])
                y_train.append(other_fold[:, d])
        i += 1

        x_train = np.concatenate(np.array(x_train))
        y_train = np.concatenate(np.array(y_train))

        print("x train shape:")
        print(x_train.shape)
        print("y train shape:")
        print(y_train.shape)
        print("x test shape:")
        print(x_test.shape)
        print("y test shape:")
        print(y_test.shape)
        temp_loss = run_on_fold(x_test, y_test, x_train, y_train, taus)
        loss.append(temp_loss)
    output = np.array(loss).mean(0)
    print(output.shape)
    print(output)
    return output
```

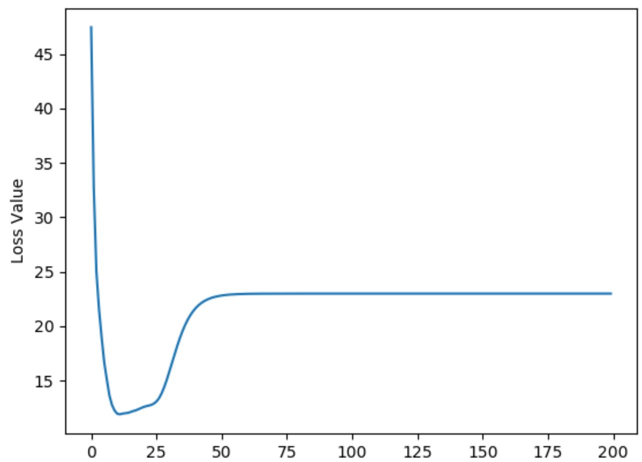
(3)



(4)

When $r- > \infty$, the loss converges around 22.4. When $r- > 0$, the loss goes to infinity.

For r = [10, 10^{10}] :



3

(1)

$S = \{a_1 \cdots a_n\}$.

$E(x) = \sum_{i=1}^{n} x_i P(x_i)$

$$\text{LHS} = E_I\left[\frac{1}{m}\sum_{i\in I} a_i\right]$$

$$= \frac{1}{m}\sum_{i=1}^{m} E(a_i)$$

since each index is drawn uniformly form the set $\{1, \cdots n\} \longrightarrow \frac{1}{n}$

$$= \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n}\frac{1}{n} a_j$$

$$= \frac{1}{n}\sum_{j=1}^{n} a_j$$

$$= \frac{1}{n}\sum_{i=1}^{n} a_i$$

$$= \text{RHS}.$$

(2)

know: $L_I(\underline{x}, y, \theta) = \frac{1}{m} \sum_{i \in I} l(\underline{x}^{(i)}, y^{(i)}, \theta)$

$\Rightarrow \nabla L_I(x, y, \theta) = \frac{1}{m} \sum_{i \in I} \nabla l(\underline{x}^{(i)}, y^{(i)}, \theta)$.

(differentiate both side preserved linearity.)

$E_I[\nabla L_I(\underline{x}, y, \theta)] = E_I[\frac{1}{m} \sum_{i=1}^{m} \nabla L_i(\underline{x}_i, y_i, \theta)]$

$= \frac{1}{m} \sum_{i=1}^{m} E(\nabla L_i(\underline{x}_i, y_i, \theta)]$.

$E(x) = \sum_{i=1}^{n} x_i P(x_i)$

since each index is drawn uniformly form the Set $\{1, \dots, n\}$ $\longrightarrow \frac{1}{n}$

$= \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{1}{n} \nabla L_j(\underline{x}_j, y_j, \theta)$

$= \frac{1}{n} \sum_{j=1}^{n} \nabla L_i(\underline{x}_j, y_j, \theta)$

$= \nabla L(\underline{x}, y, \theta)$

= RHS.

(3) It shows that in SGD, expected value of a mini-batch's loss, $E(\nabla L_I(\mathbf{x}, y, \theta))$ is equal to the true empirical gradient of loss over whole data set $\nabla L(\mathbf{x}, y, \theta)$ (a mini-batch's loss is an unbiased estimator to evaluate true gradient of loss of whole data set).

(4)

(a)

$$\frac{\partial L}{\partial \omega} = \frac{-2\mathbf{x}^{\mathbf{T}}(y - \mathbf{x}\omega)}{M} \tag{1}$$

(b)

```python
def lin_reg_gradient(X, y, w):
    """
    Compute gradient of linear regression model parameterized by w
    """
    s = X.shape[0]
    gradient_w = - 2 * np.matmul(X.T, (y - np.matmul(X, w)))/s
    return gradient_w
```

(5) In order to let the result repeatable, I set np.random.seed to be 0 again. Comment it if you need random value.

true gradient is:

```
true gradient is:
[  7.48161897e+02   2.66119727e+03   9.20005630e+03   8.36869099e+03
   5.19727844e+01   4.13891293e+02   4.69502078e+03   5.09428703e+04
   2.87367326e+03   7.18470104e+03   3.07658588e+05   1.38303337e+04
   2.76270255e+05   9.42091869e+03]
```

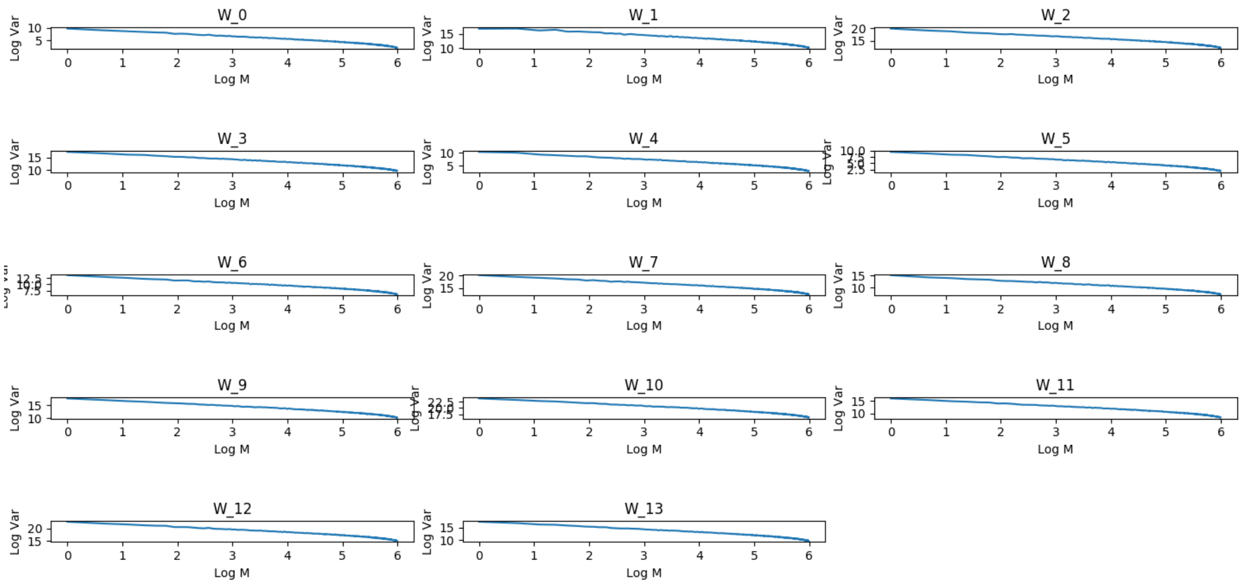
MSE is: 43870.8653703

Cosine similarity loss is: 0.999998706366

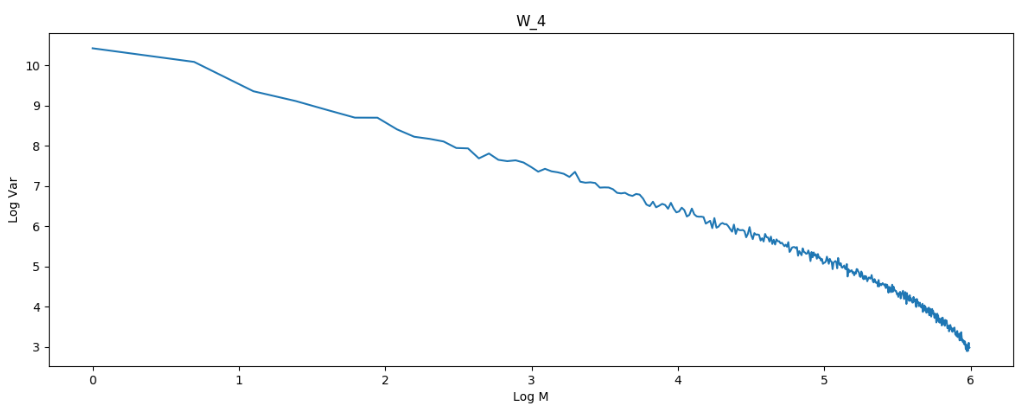
I think they are both meaningful, but in this case, cosine similarity is more meaningful. The reason is initial magnitude for each element in gradient vector are too large. So a small variation between prediction value and test value for each element in gradient vector will cause a huge difference, not mention square and summation all of them when we calculate MSE. Since vectors we compare has 14 dimensions, look at the angle between them is a good idea to measure the error in above situation (each entry's magnitude is too large). Also, the step we are doing is the first step of SGD, since the batch is randomly chosen, it can be far away from the optimal position in the beginning, therefore, for the ultimate purpose, a right direction is more important than MSE.

(6)

Result:



For a single w, i.e. w_4 (start from w_0):



Code:

```python
def calculate_gradient_var(batch_sampler, w, K):
    i = 0
    # init a 500 * 14 matrix
    gradient_matrix = np.zeros((K, batch_sampler.features), dtype = float)
    total_loss_gradient = 0
    while i < K:
        X_b, y_b = batch_sampler.get_batch()
        batch_grad = lin_reg_gradient(X_b, y_b, w)
        gradient_matrix[i, :] = batch_grad
        total_loss_gradient += batch_grad
        i += 1
    gradient = np.divide(total_loss_gradient, K)
    # take variance for each column result is a (1,14) matrix
    # vector_variance = np.log(np.sqrt(gradient_matrix.var(0)))
    vector_variance = np.log(cal_var(gradient_matrix))
    print("shape of vector variance is:")
    print(vector_variance.shape)
    return gradient, vector_variance


def cal_var(g):
    mean = g.mean(0)
    var_vector = np.zeros((1, g.shape[1]), dtype=float)
    for i in range(g.shape[0]):
        var_vector += np.power(g[i, :] - mean, 2)
    result = np.divide(var_vector, (g.shape[0]))
    return result


def plot_delta_j_vs_m(X, y, w, m_start, m_end, K):
    m_range = m_end-m_start
    x_list = np.arange(1, m_range+1)
    x_list = np.log(x_list)
    x_list.tolist()
    variance_matrix = np.zeros((m_range, X.shape[1]), dtype=float)
    plt.figure(figsize=(20, 5))
    for i in range(1, m_range+1):
        batch_sampler = BatchSampler(X, y, i)
        gradient, vector_variance = calculate_gradient_var(batch_sampler, w, K)
        print("i is:")
        print(i)
        variance_matrix[i-1, :] = vector_variance

    for j in range(X.shape[1]):
        print("j is: ")
        print(j)
        # y to plot is jth column for the vector variance.
        print(variance_matrix.shape)
        y_list = variance_matrix[:, j]
        plt.subplot(4, 4, j + 1)
        plt.plot(x_list, y_list)
```

```
        plt.title("W_{}".format(j))
        plt.xlabel("Log M")
        plt.ylabel("Log Var")
    plt.tight_layout()
    plt.show()
```