lin-reg-h2-p4(1)

February 27, 2022

1 CS 412 Homework 2, Problem 4: Linear regression

In this section, we will explore linear regression model. The dataset we will use for this section is Wine Qualuty, whose description can be found here. This dataset contains **4898** examples, each containing **11** features (the first 11 columns), and the **last** (12-th) column is the value we want to predict. The dataset can be downloaded here `winequality-white.csv` (our code will download it directly).

Different from classification models, a regression model is used to predict real values rather than the category an example belongs to. Linear regression is a linear approach to modeling the relationship between features and real value target. To perform supervised learning, we represent the hypothesis as a linear function of features (x) to predict the output (y).

$$f(x) = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n \tag{8}$$

Here θ_i's are the **parameters** parameterizing the space of linear functions mapping from $\mathcal{X}$ to $\mathcal{Y}$. Our goal is to **learn** these parameters so that we can find a linear function in this hypothesis space to estimate the output y.

To simplify the notation and ease the computation, we **pad** the input x by letting $x_0 = 1$. That is, for an example with three features $x = [x_1, x_2, x_3]$, the padded feature vector will be $x = [1, x_1, x_2, x_3]$. Then, the linear function can be written as:

$$f_\theta(x) = \sum_{j=0}^{n} \theta_j x_j = \theta^\top x \tag{9}$$

where on the right-hand side above we are viewing θ and x both as vectors, and here $n = 11$ is the number of features.

2.1 Data preprocessing

Once we have received the dataset, we first need to preprocess it. Very often, the features in a dataset are of very different scale, which can slow down the optimization for Eq (10). To accelerate it, we need to normalize each feature by substracting its mean value, and then dividing by its standard deviation (std). Assuming $X_i = [x_i^{(1)}, ..., x_i^{(m)}]$ is the i-th feature in the training set (across the m examples), the normalized feature i for the j-th training example can be computed by:

$$\hat{x}_i^j = \frac{x_i^j - m_i}{s_i}, \text{ where } m_i = mean(X_i), \text{ and } s_i = std(X_i). \tag{11}$$

Step 1: normalize the training set {-}

In the following code block, implement a function `featureNormalization`. The input is the training set. The output is the normalized training set, along with the mean and std of each features. You will need the mean and std to apply to the test set later.

[11]:
```
import numpy as np
import matplotlib.pyplot as plt
def featureNormalization(X):
    """
  Normalize each feature for the input set
  Input:
  - X: a 2-D numpy array of shape (num_train, num_features)
  Outputs:
  - X_normalized: a 2-D numpy array of shape (num_train, num_features)
  - X_mean: a 1-D numpy array of length (num_features)
  - X_std: a 1-D numpy array of length (num_features)
  """
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    num_train = X.shape[0]
    num_features = X.shape[1]

    X_mean = np.zeros(num_features)
    X_std = np.zeros(num_features)

    X_normalized = np.zeros((num_train, num_features))

    for i in range(num_features):
        f = []
        for j in range(num_train):
            f.append(X[j][i])

        X_mean[i] = np.mean(f)
        X_std[i] = np.std(f)

        for j in range(num_train):
            X_normalized[j][i] = (f[j] - X_mean[i])/X_std[i]

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return X_normalized, X_mean, X_std

"""
 Unit test case
 Should print
[[ 0.49109521 -0.40242644 -0.92156595 -0.1052653 ]
 [ 0.90298151  1.37532553  1.3897809   1.27398003]
 [-1.39407672 -0.97289909 -0.46821496 -1.16871473]]
```

```
[0.99 3.12 4.47 4.51]
[0.63124216 2.26128282 1.34553583 3.70492465]
"""
X = np.array([[1.30,2.21,3.23,4.12], [1.56, 6.23, 6.34, 9.23], [0.11, 0.92, 3.
↪84, 0.18]])
X_normalized, X_mean, X_std = featureNormalization(X)
print(X_normalized)
print(X_mean)
print(X_std)
```

```
[[ 0.49109521 -0.40242644 -0.92156595 -0.1052653 ]
 [ 0.90298151  1.37532553  1.3897809   1.27398003]
 [-1.39407672 -0.97289909 -0.46821496 -1.16871473]]
[0.99 3.12 4.47 4.51]
[0.63124216 2.26128282 1.34553583 3.70492465]
```

Step 2: normalize the test set {-}

The above normalization function will be used for the training set. At test time, we will need to normalize the test data in the same way. However, we shouldn't compute new mean and std from the test set itself, because it may be inconsistent with the training data. Instead, we will apply the mean m_i and std s_i computed from the training set. Given a text example $[x_1, \ldots, x_m]$, we just transform x_i into $(x_i - m_i)/s_i$, where m_i and s_i are computed from the training data as in the *where* clause of Eq (11).

In the following code block, implement a function `applyNormalization`, which normalizes the test set for each feature using the provided mean and std.

[13]:
```
def applyNormalization(X, X_mean, X_std):
    """
  Normalize each feature for the input set X
  Input:
  - X: a 2-D numpy array of shape (num_test, num_features)
  - X_mean: a 1-D numpy array of length (num_features)
  - X_std: a 1-D numpy array of length (num_features)

  Output:
  - X_normalized: a 2-D numpy array of shape (num_test, num_features)
  """

  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    num_train = X.shape[0]
    num_features = X.shape[1]

    X_normalized = np.zeros((num_train, num_features))

    for i in range(num_features):
        for j in range(num_train):
            X_normalized[j][i] = (X[j][i] - X_mean[i])/X_std[i]
```

```
  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return X_normalized

"""
Unit test case
Should print
[[ 3.00e-01  1.21e+00  6.15e-01  4.02e+01]
 [ 5.60e-01  5.23e+00  2.17e+00  9.13e+01]
 [-8.90e-01 -8.00e-02  9.20e-01  8.00e-01]]
"""
X =  np.array([[1.30,2.21,3.23,4.12], [1.56, 6.23, 6.34, 9.23], [0.11, 0.92, 3.
↪84, 0.18]])
X_mean = np.array([1.0, 1.0, 2.0, 0.1])
X_std = np.array([1.0, 1.0, 2.0, 0.1])
X_normalized = applyNormalization(X, X_mean, X_std)
print(X_normalized)
```

```
[[ 3.00e-01  1.21e+00  6.15e-01  4.02e+01]
 [ 5.60e-01  5.23e+00  2.17e+00  9.13e+01]
 [-8.90e-01 -8.00e-02  9.20e-01  8.00e-01]]
```

Implement the loss function {-} Given a training set, the way to learn these parameters is to make $f_\theta(x)$ close to y. To measure the closeness, we use Mean-Squared-Error (MSE) here. The loss function can therefore be defined as:

$$L(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(f_\theta(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m}\sum_{i=1}^{m}(\theta^\top x^{(i)} - y^{(i)})^2, \tag{10}$$

where the superscript (i) denotes the i-th example, and m is the total number of training samples. To learn the parameter θ, our goal is to **minimize** the above loss function. Here, we will explore exact method to learn the parameter: Closed-form solution (root of the gradient).

We will use MSE to measure the quality of our parameters, also known as the loss. In the following code block, implement a function `computeMSE`. Follow Equation (10), and the function should compute the MSE for the input set with the given θ.

[89]:
```
def computeMSE(X, y, theta):
    """
  Compute MSE for the input set (X,y) with theta
  Inputs:
  - X: a 2-D numpy array of shape (num_samples, num_features+1)
  - y: a 1-D numpy array of length (num_samples)
  - theta: a 1-D numpy array of length (num_features+1)
  Output:
  - error: MSE, a real number
  """
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
    num_samples = X.shape[0]
    num_features = X.shape[1]
    error = 0

    for i in range(num_samples):
        s = 0
        for j in range(num_features):
            s+=X[i][j]*theta[j][0]

#         print(s, y[i])
        error += (s - y[i])**2

    error/=(2*num_samples)

  # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    return error

# Unit test case:
# Should print 73.0
X = np.array([[2.0, 1.0, 3.0], [3.0, 6.0, 2.0]])
y =  np.array([1.0, 1.0])
theta = np.array([[1.0], [2.0],[1.0]])
error = computeMSE(X, y, theta)
print(error)
```

```
73.0
```

[68]:
```
# First load the data (this code block is not for grading)

import urllib.request
import shutil

url = 'https://www.dropbox.com/s/0ocbatan8pgm59i/winequality-white.csv?dl=1'
file_name = 'winequality-white.csv'
with urllib.request.urlopen(url) as response, open(file_name, 'wb') as out_file:
    shutil.copyfileobj(response, out_file)

data = np.genfromtxt(file_name, delimiter=";", skip_header=1)
print(data.shape)
```

```
(4898, 12)
```

2.5 Closed-form solution

We can find the θ explicitly by finding the root of the gradient equation $\nabla_\theta L(\theta) = 0$ (i.e., the θ such that $\nabla_\theta L(\theta) = 0$), we can obtain a closed-form solution of θ that minimizes the loss L by algebra.

Step 1: find the root of the gradient equation to obtain the closed-form solution of θ

Type your result in the following lines:

$$\nabla_\theta L(\theta) = 0 \quad \Rightarrow \quad \theta = (X^\top X)^{-1} X^\top y$$

Then, implement a function `closeForm` to compute the closed-form solution of θ using the expression you have derived above.

[73]:
```
def closeForm(X, y):
    """
  Compute close form solution for theta
  Inputs:
  - X: a numpy array of shape (num_train, num_features+1)
  - y: a 1-D numpy array of length (num_train)
  Output:
  - theta: a 1-D numpy array of length (num_features+1)
  """
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    Xt = np.matrix.transpose(X)
    t = np.matmul(np.matmul(np.linalg.inv(np.matmul(Xt,X)),Xt),y)
    theta = []
    for e in t:
        theta.append([e])
# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    return np.array(theta)

"""
# Unit test case:
# Should return
[[ 0.76470588]
 [-0.17647059]
 [-0.11764706]]
"""
X = np.array([[2.0, 1.0, 3.0], [3.0, 6.0, 2.0], [-1, 0, 2.0]])
y = np.array([1.0, 1.0, -1.0])
theta = closeForm(X, y)
print(theta)
```

```
[[ 0.76470588]
 [-0.17647059]
 [-0.11764706]]
```

Step 2: evaluate the test error using closed-form solution {-}

Evaluate the new θ on test set by **printing** the test error (MSE) in the format: "test MSE using close form solution is : ___".

[115]:
```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

"""



if need to normalize the whole dataset with y answer below, if not, then please␣
↪check the one below it



"""
X = data

n = 3918

X_test = X[n:]
X_train = X[0:n] # approx 80% to train

X_train_normalized, X_mean, X_std = featureNormalization(X_train)
X_test_normalized = applyNormalization(X_test, X_mean, X_std)

y_train = X_train_normalized[0:,11]
y_test = X_test_normalized[0:,11]

theta = closeForm(X_train[0:n,0:11], y_train)

test_error = computeMSE(X_test[0:,0:11], y_test, theta)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

print('test MSE using close form solution is : ', test_error)


#_____________________If not to normalize the␣
↪y_____________________________________________-

X = data

n = 3918 # approx 80% to train

X_test = X[n:,0:11]
X_train = X[0:n,0:11]

y_test = X[n:,11]
y_train = X[0:n,11]
```

```
X_train_normalized, X_mean, X_std = featureNormalization(X_train)
X_test_normalized = applyNormalization(X_test, X_mean, X_std)

theta = closeForm(X_train_normalized, y_train)

test_error = computeMSE(X_test_normalized, y_test, theta)

print('test MSE using close form solution is : ', test_error)
```

```
test MSE using close form solution is :  0.30798526782057256
test MSE using close form solution is :  16.624083988802393
```

[ ]: