ML2016 HW1 Linear Regression

October 14, 2016

1 Linear regression function by Gradient Descent

1.1 Packages

```
import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt
```

1.2 Define the loss function

```
def computeLoss(X, y, coef):
    y_hat = np.dot(X, coef)
    return sum((y - y_hat)**2) / len(y)
```

1.3 Main loop of gradient descent

```
iteration = 3000 # number of iteration
eta = 0.00001 # learning rate
loss_history = list()
random.seed(1002) # Randomly initialize coefficients
coef = [random.random()/100.0 for i in range(train_norm.shape[1])]
for ite in range(iteration):
    loss_history.append(computeLoss(train_norm, y, coef))
    temp = list()
    ## compute all gradients and the updated coefficients
    for i in range(train_norm.shape[1]):
        temp.append(coef[i] - \
                      alpha * sum((np.dot(train_norm, coef) - y) \
                                  .multiply(train_norm.ix[:,i])))
    ## update all coefficients
    for i in range(train_norm.shape[1]):
        coef[i] = temp[i]
```

2 Method description

2.1 Preprocessing

I use **numpy** and **pandas** packages for data manipulation. First, I use the **pandas.melt** method to reshape the training data into a long one-value-per-row matrix. Second, I use a `for` loop to extract all possible 10-hour windows and build the complete one-record-per-row matrix with dimension (5652, 163) as shown below. The last column `PM2.5_h10` contains all PM2.5 values in the 10-th hour of all 10-hour windows.

AMB_TEMP_h1	CH4_h1	CO_h1	...	WS_HR_h9	PM2.5_h10
14.0	1.8	0.51	...	0.5	30.0
14.0	1.8	0.15	...	0.3	41.0
14.0	1.8	0.13	...	0.8	44.0
13.0	1.8	0.12	...	1.2	33.0
12.0	1.8	0.11	...	2.0	37.0
...	...	...	...	...	...

2.2 Data cleansing

Remove wrong (or strange) records that with any negative feature or outcome. For example, there are some records with `AMB_TEMP < 0`, which is not reasonable and should be discarded. This can be done by the following code snippet, in which `train_all` is the aforementioned 5652-by-163 matrix. The resulted `train_clean` is a 5300-by-163 matrix.

```
train_clean = train_all[(train_all >= 0).all(1)]
```

2.3 Spliting feature and outcome, normalization, and adding constant column

```
# Split features and outcome
y = pd.Series(train_clean.ix[:, train_clean.shape[1]-1])
del train_clean['y']
# Normalize (make all features to be zero-mean and unit-variance)
train_norm = (train_clean - train_clean.mean()) / train_clean.std()
# Add a column with all 1
train_norm.insert(loc = 0, column = 'intercept', value = 1)
```

3 Discussion on regularization

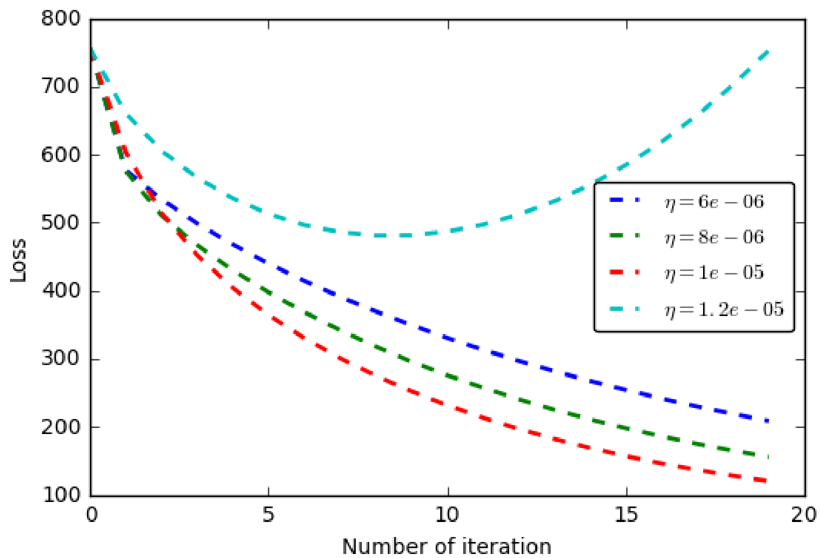
I also implement a polynomial regression, where the **squared** version of all `162` features are included to make a 5300-by-325 big feature matrix (including the first all-1 column). This matrix is also normalized.

Then, I try different regularizer (`0`, `0.1`, `0.3`, `1`, `3`, and `10`) and run 5-fold cross validation with iteration number `400`. The average training- and validation-error (mean-square-error) of $\lambda = 0$ (no regularization) and $\lambda = 1$ are shown below. It seems that $\lambda = 1$ gives a slightly smaller validation-error. If we run more and more iteration, there should be more improvement.

The implementation can be found in `polynomial_regression_reg.py` in the github repository.

λ	training	validation
0	33.6054	37.2652
1	33.6079	37.2573

4 Discussion on learning rate



learning rate

Since there are totally `162` features in the linear regression formula, the gradient summation term `sum((np.dot(train_norm, coef) - y).multiply(train_norm.ix[:,i]))` will be relatively large in each iteration, and hance we need a relatively small learning rate.

As showed in the above figure, choosing learning rate to be `0.00001` seems to be adequate.

The implementation can be found in `learning_rate_adjustment.py` in the github repository.