ASSIGNMENT 2 - Linear Regression with Gradient Descent

Matthew Dunne

Use linear regression with gradient descent to predict water temperature T_degC using the dataset from Assignment 1.

1) Only use 'Salnty', 'STheta' for predictors (same as HW #1)
2) Remove NaN / NA values from dataset (prior to building train/test sets) (same as HW #1)

In [1]:

```
import numpy as np
import os
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score, explained_variance_score
from math import sqrt
```

In [2]:

```
data=pd.read_csv('bottle.csv', usecols=['T_degC','Salnty', 'STheta'])
#remove NaN/NA values
data=data.dropna()
X=data[['Salnty', 'STheta']]
Y=data['T_degC']
```

3) Scale all features to improve convergence. It is highly encouraged that you review the appropriate method of handling normalization with train & test.

In [3]:

```
from sklearn import preprocessing
scaler=preprocessing.StandardScaler()
#split into training and test first
X_train, X_test, y_train, y_test = train_test_split(X, Y, random_state=0)
#fit and transform the training data for the predictors
X_train_scaled=scaler.fit_transform(X_train)
#then scale the test data for the predictors using the parameters derived from fitting/scaling on
the training
X_test_scaled=scaler.transform(X_test)

X_train_scaled_b = np.c_[np.ones((len(X_train_scaled), 1)), X_train_scaled] # add x0 = 1 to each in
stance of the training data
X_test_scaled_b = np.c_[np.ones((len(X_test_scaled), 1)), X_test_scaled] # add x0 = 1 to each
instance of the test data
print(X_train_scaled_b)
```

```
[[ 1.         -1.08252055 -1.2279179 ]
 [ 1.         -0.69895951 -1.20983697]
 [ 1.          0.96963937  0.92535686]
 ...,
 [ 1.          0.03782442 -0.55399221]
 [ 1.         -1.02401124 -0.6189192 ]
 [ 1.         -1.50942182 -0.66083409]]
```

Here I also reshaped y_train into an array of one column so that the code for the mini-batch operation will work.

In [4]:

```
y_train=y_train.reshape(-1, 1)
y_train
```

```
C:\Users\mjdun\Anaconda\lib\site-packages\ipykernel_launcher.py:1: FutureWarning: reshape is
deprecated and will raise in a subsequent release. Please use .values.reshape(...) instead
  """Entry point for launching an IPython kernel.
```

Out[4]:

```
array([[ 16.65],
       [ 17.14],
       [  6.4 ],
       ...,
       [ 14.76],
       [ 13.33],
       [ 12.71]])
```

We can see above the training data both before and after scaling. The scaled data is stored as a numpy array.

4) For the cost function (the thing we are trying to minimize) we will use mean squared error. Please use the same function utilized in Chapter 4 of The Hands On Machine Learning (Equation 4.5) for the derivative of the cost function.

```
derived_cost_function = 2/m * X_b.T.dot(X_b.dot(theta) - y)
```

5) Try mini batch sizes of 50, 2000 and 10,000. Comment on the prediction accuracy based on rmse, variance explained, and r-squared.
Note: Feel free to use any value for eta and epochs, but 0.1 eta and 100 epochs (number of times you go through the whole batch by taking mini-batches) are fine for this HW.

First with batch sizes of 50

In [6]:

```
theta_path_mgd = []
m = len(X_train_scaled_b)
eta=0.1
n_iterations = 100
minibatch_size = 50

np.random.seed(42)
theta = np.random.randn(3,1)  #random start for theta with three rows and one column

#run without a learning schedule
#t0, t1 = 200, 1000
#def learning_schedule(t):
    #return t0 / (t + t1)

#t = 0
for epoch in range(n_iterations):
    #shuffle our indices to make each epoch distinct subsets
    shuffled_indices = np.random.permutation(m)
    #reorder according to our new indices. Make sure x and y line up
    X_train_scaled_b_shuffled = X_train_scaled_b[shuffled_indices]
    y_train_shuffled = y_train[shuffled_indices]
    #go from observation 0 to the end in chunks = to mini batch size
    for i in range(0, m, minibatch_size):
        #t += 1
        xi = X_train_scaled_b_shuffled[i:i+minibatch_size]
        yi = y_train_shuffled[i:i+minibatch_size]
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        #eta = learning_schedule(t)
        theta = theta - eta * gradients
        #theta_path_mgd.append(theta)
rmse_fifty=sqrt(mean_squared_error(y_test,X_test_scaled_b.dot(theta)))
var_exp_fifty=explained_variance_score(y_test,X_test_scaled_b.dot(theta))
r2_fifty=r2_score(y_test,X_test_scaled_b.dot(theta))
theta
```

Out[6]:

```
array([[ 10.82943271],
       [  1.43439561],
       [ -6.14612831]])
```

The RMSE, Variance Explained, and R Squared for the minibatch size of 50 are respectively:

In [7]:

```
print(rmse_fifty, var_exp_fifty, r2_fifty)
```

```
0.48112145692253616 0.986984360439 0.986980768192
```

Mini-Batch sizes of 2000

In [8]:

```
m = len(X_train_scaled_b)
eta=0.1
n_iterations = 100
minibatch_size = 2000

np.random.seed(42)
theta = np.random.randn(3,1)  #random start for theta with three rows and one column

#run without a learning schedule
#t0, t1 = 200, 1000
#def learning_schedule(t):
    #return t0 / (t + t1)

#t = 0
for epoch in range(n_iterations):
    #shuffle our indices to make each epoch distinct subsets
    shuffled_indices = np.random.permutation(m)
    #reorder according to our new indices. Make sure x and y line up
    X_train_scaled_b_shuffled = X_train_scaled_b[shuffled_indices]
    y_train_shuffled = y_train[shuffled_indices]
    #go from observation 0 to the end in chunks = to mini batch size
    for i in range(0, m, minibatch_size):
        #t += 1
        xi = X_train_scaled_b_shuffled[i:i+minibatch_size]
        yi = y_train_shuffled[i:i+minibatch_size]
        #note yi has to be an array (which is why you converted it above)
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        #eta = learning_schedule(t)
        theta = theta - eta * gradients
        #theta_path_mgd.append(theta)
rmse_two_k=sqrt(mean_squared_error(y_test,X_test_scaled_b.dot(theta)))
var_exp_two_k=explained_variance_score(y_test,X_test_scaled_b.dot(theta))
r2_two_k=r2_score(y_test,X_test_scaled_b.dot(theta))
theta
```

Out[8]:

```
array([[ 10.83313419],
       [ -4.33355579],
       [  1.57658437]])
```

The RMSE, Variance Explained, and R Squared for the minibatch size of 2000 are respectively:

In [9]:

```
print(rmse_two_k, var_exp_two_k, r2_two_k)
```

```
4.733212559075289 -0.260037057641 -0.260050422588
```

Mini-Batch sizes of 10,000

In [10]:

```
m = len(X_train_scaled_b)
eta=0.1
n_iterations = 100
minibatch_size = 10000

np.random.seed(42)
theta = np.random.randn(3,1)  #random start for theta with three rows and one column
```

```
#run without a learning schedule
#t0, t1 = 200, 1000
#def learning_schedule(t):
    #return t0 / (t + t1)

#t = 0
for epoch in range(n_iterations):
    #shuffle our indices to make each epoch distinct subsets
    shuffled_indices = np.random.permutation(m)
    #reorder according to our new indices. Make sure x and y line up
    X_train_scaled_b_shuffled = X_train_scaled_b[shuffled_indices]
    y_train_shuffled = y_train[shuffled_indices]
    #go from observation 0 to the end in chunks = to mini batch size
    for i in range(0, m, minibatch_size):
        #t += 1
        xi = X_train_scaled_b_shuffled[i:i+minibatch_size]
        yi = y_train_shuffled[i:i+minibatch_size]
        gradients = 2/minibatch_size * xi.T.dot(xi.dot(theta) - yi)
        #eta = learning_schedule(t)
        theta = theta - eta * gradients
        #theta_path_mgd.append(theta)
rmse_ten_k=sqrt(mean_squared_error(y_test,X_test_scaled_b.dot(theta)))
var_exp_ten_k=explained_variance_score(y_test,X_test_scaled_b.dot(theta))
r2_ten_k=r2_score(y_test,X_test_scaled_b.dot(theta))
theta
```

Out[10]:

```
array([[ 10.84424185],
       [ -1.52533256],
       [ -0.9195565 ]])
```

The RMSE, Variance Explained, and R Squared for the minibatch size of 10,000 are respectively:

In [11]:

```
print(rmse_ten_k, var_exp_ten_k, r2_ten_k)
```

```
3.102511375375045 0.458619583381 0.458619420071
```

Conclusion

The mini-batch of size 50 has the best performance with the lowest Root Mean Squared Error and the highest R Squared/Variance Explained.