Machine Learning: HW1 Report

B03901034 吳泓霖

Linear regression function by Gradient Descent

```
#!/usr/bin/python
import math
from operator import mul

# Initialize values
# The parameters for the linear regression
params = [[0 for x in range(9)] for x in range(18)] + [[0]]
# The gradient for each parameter
grad = [[0 for x in range(9)] for x in range(18)] + [[0]]
# The sum of all square of gradient of each parameter, used for Adagrad
grad_sq_sum = [[0 for x in range(9)] for x in range(18)] + [[0]]
# Container of training data
train = [None] * 18
# Container of test data
test = []
# Container of the name of test data (e.g: 'id_1')
test_name = []

# How many iterations
loop = 40000
# Initial learning rate
l_rate = 1
# lambda for regularization
lam = 0
# Sample from hour No.prev to hour No.9
prev = 0
# A control panel to decide whether include as a feature or not
dataType = {
    'AMB_TEMP': [True, 0],
    'CH4': [True, 1],
    'CO': [True, 2],
    'NMHC': [True, 3],
    'NO': [True, 4],
    'NO2': [True, 5],
    'NOx': [True, 6],
    'O3': [True, 7],
```

```python
    'PM10': [True, 8],
    'PM2.5': [True, 9],
    "RAINFALL": [False, 10],
    "RH": [True, 11],
    'SO2': [True, 12],
    "THC": [True, 13],
    "WD_HR": [True, 14],
    "WIND_DIREC": [False, 15],
    "WIND_SPEED": [True, 16],
    "WS_HR": [True, 17],
}
# Return whether include as a feature
def include(string):
    return dataType.get(string, [False])[0]
# Return the column number of a feature
def getIndex(string):
    return dataType.get(string)[1]

# Return the estimated value by calculating from the parameters and given data
def estim(data):
    num = 0
    for i in range(len(data)):
        if data[i] != None:
            num += sum(map(mul, params[i][prev:9], data[i][prev:9]))
    num += params[len(params) - 1][0]
    return num

# Update the gradient table for all parameters
def gradient(data):
    actual = data[getIndex('PM2.5')][9]
    estimate = estim(data)
    for i in range(len(data)):
        if data[i] != None:
            for j in range(prev, len(grad[i])):
                grad[i][j] -= 2 * (actual - estimate) * data[i][j]
                grad[i][j] += 2 * lam * params[i][j]
    grad[len(grad) - 1][0] -= 2 * (actual - estimate)

# Update the parameters with gradient descent
def updateParam():
    global grad
    size = len(train)
    for i in range(size):
        if train[i] != None:
            for j in range(prev, len(params[i])):
                grad_sq_sum[i][j] += math.pow(grad[i][j], 2)
                params[i][j] -= rate(i, j) * grad[i][j]
    grad_sq_sum[size][0] += math.pow(grad[size][0], 2)
    params[size][0] -= rate(size, 0) * grad[size][0]
    # Reset gradient table for the next iteration
    grad = [[0 for x in range(9)] for x in range(18)] + [[0]]

# Adagrad
def rate(i, j):
    return l_rate / math.sqrt(grad_sq_sum[i][j])

# Read training data
...some code...

# Read test data
...some code...

# Training
for x in range(loop):
    # Loop through every 10 hours
    for i in range(len(train[getIndex('PM2.5')]) - 10):
        gradient(map(lambda x: x[i : i + 10] if x != None else None, train))
    updateParam()

# Output result
...some code...

```

Method

The best public result on Kaggle is based on Linear regression, as presented in the previous section. I included all data as a feature (except rainfall and wind direction, because 1. Only a few days were raining 2. The wind direction is an angle, not a value). This means there are a total of $(18-2)\times 9+1 = 145$ parameters in the estimated model.
I used every 10 hours (can cross another day) as a training set, which will take the data from the first 9 hours as a feature.
I didn't use any external libraries to implement the gradient descent, as most operations are not that complicated and can be implemented with neat features like "map". The gradient part is also relatively easy and can be done with simple arithmetics.

Regularization

Regularization will only effect the loss function, and base on the results I experimented with lambda, I can't see a substantial improvement with small values; however if the value was assigned with a value that is too large, it will damage the effectiveness of Adagrad and make the result unable to converge. So I didn't enable regularization for the final submission.

Learning Rate

Initially, I implemented linear regression with a constant learning rate. The learning rate has to be very small to make the result converge. With the small and constant learning rate, it will take forever for the result to be a value that is acceptable. With adaptive learning rate a.k.a Adagrad, the result will quickly converge to an area near the optimal solution, and will slowly approach the best result without being too aggressive.

Others

The best result on Kaggle is based on the linear regression method stated in the first section, however I had tried to change the model to a quadratic function and train only with PM2.5 data. The public test set is not as good as the best on Kaggle, however it is not that bad either, so I'd like to give this new model a shot for private data set. It is one of the two preferred submission, another is the public best case.
It is better to run the python script with pypy, which runs the code MUCH faster than the official Python interpreter and requires no adjustments in the code.
Running the linear regression script might exceed 1 hour as there are a lot of variables. With the quadratic model, it can finish in 10 minutes. Both time are calculated running with pypy, which means it will take a lot more time to finish the tasks with the official Python interpreter.