Machine Learning Assignment 2 -- Spam Classifier

物理三 B03202017 李漪莛

Logistic Regression

- Feature Set

  58 features in the feature set. (1 bias + 57 features, same as in "spam_train.csv")

- Training Data and Cross Validation Data

  I divided 4001 samples into 2 groups. If the id of the sample satisfies (id % 4 == 1), then it will be assigned into cross validation data(1000 samples), and others are training data(3001 samples). Thus, the error assessment can be more accurate.

- Normalization

  I randomly chose 1000 samples in the training data to accumulate and get the coefficient of the normalization. Then, I normalized all the data before training and cross validation. And it actually gets a better performance by testing.

- Regularization (Def lambda, λ = coefficient of regularization)

  I selected λ by considering the following steps:

  1. $\lambda \in \{ 1 \times 10^a \text{ or } 3 \times 10^a \text{, with } a \in Z \}$
  2. Choose several "a", and test which one can get the better performance.
  3. If [CV-error minus train-error] can be smaller after choosing a bigger λ. (Avoid overfitting.)
  4. If 2 λ get the same performance, then I will choose the one whose "lost" decreases more rapidly after the same iterations.

- Adam and Learning Rate (eta)

  I use Adam to replace normal gradient descent, and it can find the weight, which is much closer to the minimum lost. And since Adam will decrease "eta" gradually itself, I choose the eta, which won't cause "lost" divergence (not too big), and also the largest eta at the same time.

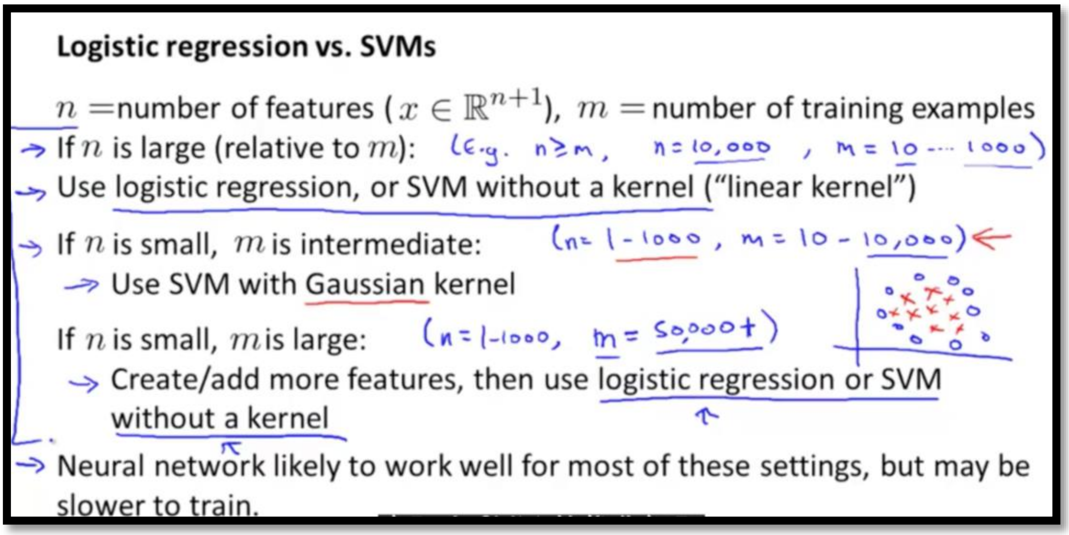
- Code in Training Part (Neglect all "self." in the code for simplicity)

```
def sigmoid(self, num):
    return 1/(1+np.exp(-num/exponential_norm))

def lost(self, y_dot, t):
    lost = 0
    for i in range(y.shape[0]): # y.shape[0] = size of training data
```

```
        if y[i][0] != y_dot[i][0]: # make sure no zero in "log" function
            lost += -(y[i][0] * np.log(y_dot[i][0]) + (1-y[i][0]) * 
np.log(1-y_dot[i][0]))
    print "lost = ", round(lost, 1), t

def train_model(self):
    i_matrix = np.eye(w.shape[0])
    i_matrix[0][0] = 0  # avoid to accumulate "lost" of the weight of the bias
    m = np.empty([w.shape[0], w.shape[1]], dtype = float)
    v = np.empty([w.shape[0], w.shape[1]], dtype = float)
    beta1 = 0.9
    beta2 = 0.999  # m, v, and beta are containers and coefficients of Adam
    for t in range(iterator):
        y_dot = self.sigmoid(np.dot(self.x, self.w))  # "y" I guess
        self.lost(y_dot, t)  # compute lost
        gra = -sum((y - y_dot)*x) + lam * sum(np.dot(i_matrix, w))
        m = beta1*m + array((1-beta1)* gra).reshape(w.shape[0], 1)
        v = beta2*v + array((1-beta2)*(gra**2)).reshape(w.shape[0], 1)
        m_dot = m/(1-beta1**(t+1))
        v_dot = v/(1-beta2**(t+1))
        w = w - eta*m_dot/(v_dot**(1/2) + 1E-8)  # update weight
```

Another Method – Support Vector Machine (SVM)

- I chose my second method among three algorithms:
  The First one is **neural network**, but I found that it is much slower and cannot make sure to find a global minimum.
  The Second is **SVM with Guassion kernel**. According to the figure below [1], this method can fit nonlinear feature and may be the most suitable one for training our data. (m = 4001 and n = 58 in this assignment)



However, I finally found that it is also very slow for training (5~10 minutes for one iteration), because its complexity = m x m x n = 4001² x 58, especially that I can only train by my laptop. Thus, I gave up this method, and chose the last one, **SVM without kernel**.

- <u>Feature Set</u>

  It is same as in logistic regression.

  In addition, I tried to add extra features, which are the square of 57 original ones, that is, total 115 features (1+57+57).

  The reason is that, after I made some plots for the data, I observed the relation between the features (x) and labels (y) is not only for linear equation.

- <u>Normalization</u> (Same as logistic regression)
- <u>Regularization</u> (Same method as logistic regression)

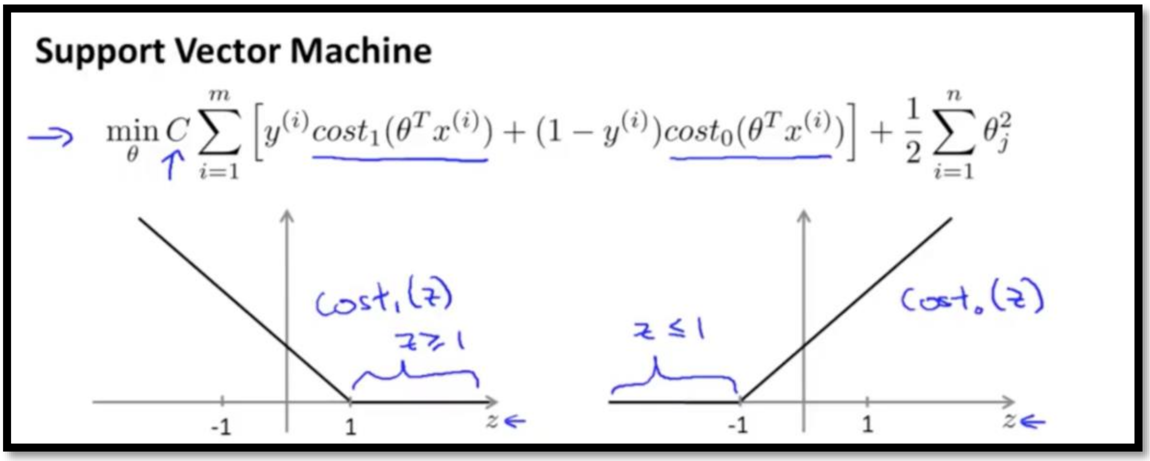
  Use different λ for the first 57 (original) and the last 57 (square) features. Since after I added additional features, the performance even got worse, no matter how I tried λ. With different λ, performance gets better.

- <u>Adam and Learning Rate</u> (eta)

  I adjusted "eta" as the same way as the logistic regression I mentioned above.

- <u>Code and Lost Function</u>

  I designed the lost function according to the lecture " Machine Learning" on Coursera [2], and tried to derive the equation of lost function and also its derivative, "grad" on my own.




```
def cost1(self, z): # the cost function I designed by observing the figure above
    if z >= 1: return 0  # slope = 0.28 by observing the figure above
    else:      return -slope*(z-1)
def cost0(self, z):
    if z <= -1: return 0
    else:       return slope*(z+1)
def grad_cost1(self, z):  # the diffirential of the cost1
        if z >= 1: return 0
        else:      return -slope
def grad_cost0(self, z):
    if z <= -1: return 0
    else:       return slope

def lost(self, z, t):  # compute lost function
    lost = 0
    for i in range(y.shape[0]):  # size of samples
        lost += y[i][0]*cost1(z[i][0])+(1-y[i][0])*cost0(z[i][0])
    lost = lost + sum((w[1:, :])**2)/2*lam
    print "lost = ", round(lost, 1), t

def train_model(self):
    i_matrix = np.eye(w.shape[0])
    i_matrix[0][0]  = 0  # avoid adding the lost of the weight of the bias
    m = np.empty([w.shape[0], w.shape[1]], dtype = float)
    v = np.empty([w.shape[0], w.shape[1]], dtype = float)
    beta1 = 0.9
```

```
    beta2 = 0.999
    grad = np.empty([w.shape[0], w.shape[1]], dtype = float)
    for t in range(iterator):
        z = np.dot(x, w)
        self.lost(z, t)  # compute lost function
        for i in range(w.shape[0]):  # compute gradient
            grad[i][0] = 0
            for j in range(y.shape[0]):
                grad[i][0] += y[j][0]*(grad_cost1(z[j][0])*x[j][i])
  + (1-y[j][0])*(grad_cost0(z[j][0])*x[j][i])
            if i!=0: grad[i][0] += (w[i][0]*lam)
        m = beta1*m + array((1-beta1)* grad).reshape(w.shape[0], 1)
        v = beta2*v + array((1-beta2)*(grad**2)). reshape(w.shape[0],
    1)
        m_dot = m/(1-beta1**(t+1))
        v_dot = v/(1-beta2**(t+1))
    w = w - eta*m_dot/(v_dot**(1/2) + 1E-8)  # update weight
```

Compare Methods and Conclusion

	λ with best performance	Error		Time for 1 iteration (4001 training data)
		Train	CV	
Logistic Regression with 58 features [Method 1]	0.0001	0.070	0.079	0.006 sec
SVM with 58 features [Method 2]	0.03	0.071	0.075	1.1~1.2 sec
SVM with 115 features and same lambda	1000	0.223	0.230	1.1~1.2 sec
SVM with 115 features and different lambdas	$\lambda_{orig} = 0.03$ $\lambda_{squa} = 10^4$	0.085	0.091	1.1~1.2 sec
SVM with kernel	Maybe it is better than others			5~10 min
neural network	I would like to try next time			

The performance of logistic regression and normal SVM is almost same, but the former one is much faster. Besides, after I add more features by my own, the performance gets worse. For instance, I have to set a bigger one lambda in “SVM with 115 features”, and it maybe means that the extra features is useless.

Reference

[1] Machine Learning lecture on Coursera – Using An SVM
https://www.coursera.org/learn/machine-learning/home/week/7

[2] Machine Learning lecture on Coursera – Large Margin Intuition
https://www.coursera.org/learn/machine-learning/home/week/7