1. Logistic regression function

In Fig.1 , the main logistic regression code is shown, as the variable names suggest, the operations are mainly done by matrix operations.

```
for it in range(it_max):
    z_mat = M_mat * W + b
    sig_mat = 1 / (1 + np.power(np.e,-z_mat))
    err_mat = Y_mat - sig_mat;
    if it%1000 == 0:
        est_mat = sig_mat
        est_mat[est_mat>=0.5] = 1
        est_mat[est_mat<0.5] = 0
        print "[%5d"%it,"/",it_max,"] loss = %12.5f"%np.sum(
    # Gradient Descent
    db = -1.0 * np.sum(err_mat)
    dW = -M_mat.T * err_mat # linear weights
    # Regularization
    dW = dW + 2*lam*np.array(W)
    # Adagrad
    ada_b = np.sqrt(np.square(ada_b) + np.square(db))
    ada_w1 = np.sqrt(np.square(ada_w1) + np.square(dW))
    # Update Step
    b = b - rate * db / (ada_b + eps)
    W = W - rate * dW / (ada_w1 + eps)
```

Fig. 1

Before entering the loop in Fig. 1, I did some **standardization** of the data, by using the following formula:

$$ft = \frac{ft - ft.mean}{ft.std}$$

By doing so, it accelerates the convergence and also avoids having some overflow problems with the sigmoid function.

During this homework, I tried out 2 different implementations of the sigmoid function. The 1st one is the same as Fig.1 which directly computes the sigmoid, the 2nd one uses

```
def sigmoid(z):
    return np.exp(-np.logaddexp(0, -z))
```

which computes it in the log domain then reverse it, which could avoid the overflowing problem.

Notes on Regularization:

During the implementation of logistic regression, I tried several different λ during validation, the best one was λ=50, yet it turned out to be pretty poor for the private set. Hence, after the kaggle deadline, I did some analysis on how regularization would affect the results on both public and private set.

The results are shown in Table 1 and Fig. 2.

λ	0	0.01	0.1	1	2	3	5	20	50
Public ACC	0.927	0.927	0.927	0.920	0.920	0.917	0.913	0.907	0.903
Private ACC	0.903	0.900	0.893	0.890	0.893	0.893	0.897	0.887	0.870

Table 1

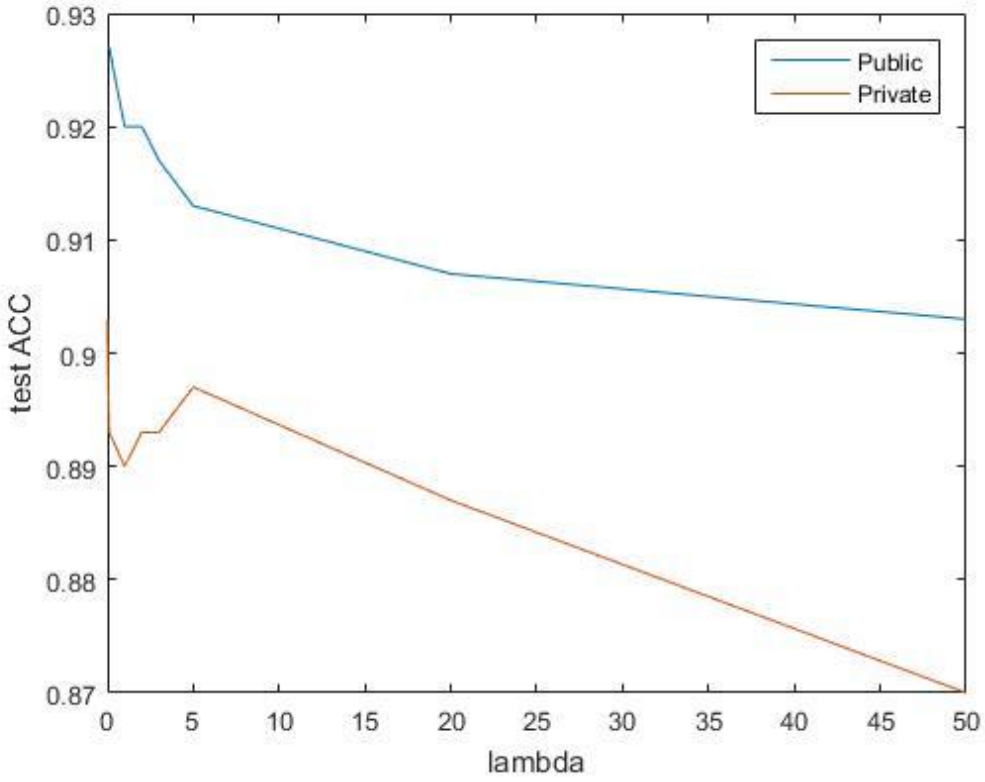


Fig. 2

We can see that lambda greatly affects the accuracy, especially the private set.

Learning rate also greatly affects the convergence of each iteration, based on experimental results, rate = 0.1 yields the best convergence.

---

2. Describe your another method

Methods I tried:

During this homework, I implemented several methods, including Adaboost, NN, and also some expansion of logistic regression to allow features with quadratic orders.

Adaboost was implemented by combining several logistic regression classifiers, yet it didn't seem to have superior performance than logistic regularization. My guess is that some weaker classifier should be chosen, such as some basic decision stump.

Expansion of logistic regression was done by quadratic mapping of the original 57 dimensions of features. I included cross terms among them. Hence, there would be

a total of 57*57 features and 1 bias term. The results are better than logistic regularization, yet slightly worse than NN.

Neural Network (1-layer / 2-layer / 3-layer)

My best result of kaggle's private set was obtained by using a 3-layer NN. Its core structure is shown in Fig. 3

```
for it in range(it_max):
    # Forward propagation
    z1 = X.dot(W1) + b1
    a1 = sigmoid(z1)
    z2 = a1.dot(W2) + b2
    a2 = sigmoid(z2)
    z3 = a2.dot(W3) + b3
    # Soft Max
    exp_scores = np.exp(z3)
    probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    if it % 1000 == 0 or it == it_max -1:
        correct_logprobs = -np.log(probs[range(num), Y])
        data_loss = np.sum(correct_logprobs)
        print "[ %5d"%it,"/",it_max,"]      Loss = %.8f"%(data_loss/num)
    # Backpropagation
    delta3 = probs
    delta3[range(num), Y] -= 1
    dW3 = (a2.T).dot(delta3)
    db3 = np.sum(delta3, axis=0, keepdims=True)
    delta2 = delta3.dot(W3.T) * a2 * (1 - a2)
    dW2 = (a1.T).dot(delta2)
    db2 = np.sum(delta2, axis=0, keepdims=True)
    delta1 = delta2.dot(W2.T) * a1 * (1 - a1)
    dW1 = np.dot(X.T, delta1)
    db1 = np.sum(delta1, axis=0)
    # Regularization
    dW3 += lam * W3
    dW2 += lam * W2
    dW1 += lam * W1
    # Adagrad
    ada_W1 = np.sqrt(np.square(ada_W1) + np.square(dW1))
    ada_b1 = np.sqrt(np.square(ada_b1) + np.square(db1))
    ada_W2 = np.sqrt(np.square(ada_W2) + np.square(dW2))
    ada_b2 = np.sqrt(np.square(ada_b2) + np.square(db2))
    ada_W3 = np.sqrt(np.square(ada_W3) + np.square(dW3))
    ada_b3 = np.sqrt(np.square(ada_b3) + np.square(db3))
    # Gradient descent with Adagrad
    W1 += -rate * dW1/(ada_W1 + eps)
    b1 += -rate * db1/(ada_b1 + eps)
    W2 += -rate * dW2/(ada_W2 + eps)
    b2 += -rate * db2/(ada_b2 + eps)
    W3 += -rate * dW3/(ada_W3 + eps)
    b3 += -rate * db3/(ada_b3 + eps)
```

Fig. 3

The main reference of the implementation of NN is from:

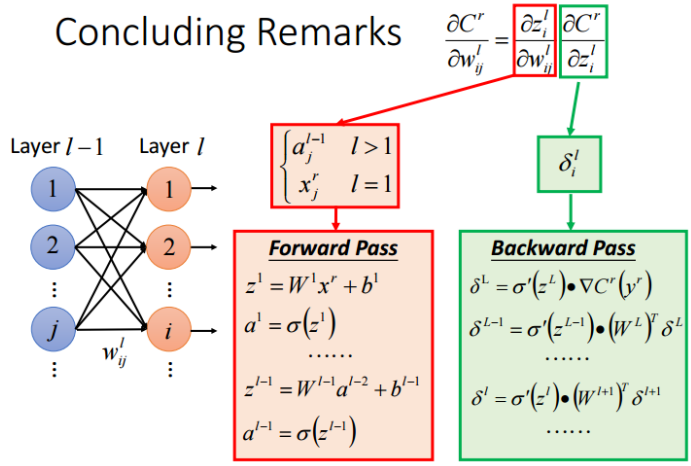
[1] http://speech.ee.ntu.edu.tw/~tlkagk/courses/MLDS_2015_2/Lecture/DNN%20backprop.pdf

[2] http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/

[3] https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

My model shares some similarity with [2], the reason is that it has a proper structure of model handling. Hence, I followed its coding structure to build up my function. The calculation of backpropagation follows mostly from [1], specifically this

diagram:



I also used Adagrad to speed up the convergence of gradient descent. However, after the lecture on 10/28, I learned that it might be better if I used min-batch gradient descent. After some basic try-out, it does turn out to have a faster loss-drop and could escape local-min.

Results:

Even though using a 2-layer NN yielded a 95.3% ACC on the public set, its private set's ACC dropped to 91.0%. The Best Kaggle result was obtained by using a 3-layer NN, each with (11, 5, 3) neurons, and $\lambda = 4$.

Variation:

I also tried doing some **quadratic mapping** of the 57 features, by doing so, I would have 57×2=114 features as input to the NN. It does result in lesser training error but it took much more time to train and have some overfitting problems. Hence, it would require some careful regularization and selection of neuron numbers. However, I failed to find a proper set of parameters that would yield a better validation result than a basic NN.

Notes on Validation:

During this homework, I did a simple **2-fold cross validation**, it was done by dividing the 4000 data set into 2 sets: (3800, 200) and (200, 3800).

During each process, the 3800 data is first to train the model, and the 200 data would be used for validation. The total validation error would be the average over these two processes. If the validation error is good, I would perform a final training, that uses the whole (4000) data to train a proper model. The last step is definitely profitable, especially for using such a small data set. For example, my best public set result was 95.3%, which was obtained by using only the latter 3800 of the data. Yet its private set was worse than the result of using full 4000 data.

Conclusion:

My NN implementations yielded drastically different results for public and private sets, my guess is that there are some other adjustments could be made (for example, mini-batch or probably some neuron pruning).