In [1]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:

```
train3_f = 'train3_oddYr.txt'
test3_f = 'test3_oddYr.txt'
train5_f = 'train5_oddYr.txt'
test5_f = 'test5_oddYr.txt'

train3 = np.loadtxt(train3_f, dtype=int)
test3 = np.loadtxt(test3_f, dtype=int)
train5 = np.loadtxt(train5_f, dtype=int)
test5 = np.loadtxt(test5_f, dtype=int)
```

In [5]:

```
label_train3 = np.zeros(train3.shape[0])
label_train5 = np.ones(train5.shape[0])
```

In [6]:

```
# concatenate the label and the 64 * 1 vector together for random shuffling
train3 = np.concatenate((train3,label_train3[:,np.newaxis]),axis = 1)
train5 = np.concatenate((train5,label_train5[:,np.newaxis]),axis = 1)
```

In [12]:

```
#extract x and y from training samples
train = np.concatenate((train3,train5))
np.random.shuffle(train)
x_train = train[:,:-1]
y_train = train[:,-1]
print(x_train.shape,y_train.shape)
```

```
(1400, 64) (1400,)
```

In [13]:

```
#initialization of w
w = np.random.randn(64,1) / 100
```

In [14]:

```
# define sigmoid function
def sigmoid(w,x):
    z = np.dot(x,w)
    return(1/(1+np.exp(-z)))
```

For this problem, I will use the ML estimator to estimate the parameters w:

- For the loss function of this binary-classification problem, I use the cross entropy loss;
- The optimization algorithm I use is gradient descent;
- Maximum iterations of this algorithm is 1000, learning rate = 1 / 700;

- Every 10 iterations, the loss and the corresponding error rate will be updated and recorded.

In [15]:

```
alpha = 0.2 / x_train.shape[0]
max_iter = 5000
loss_list = []
error_list = []
for i in range(max_iter):
    prob = sigmoid(w,x_train)
    temp = np.log(prob) * y_train[:,np.newaxis] + np.log(1-prob) * (1-y_train)[:,np.
    loss = -1 / x_train.shape[0] * np.sum(temp,axis = 0)
    if i%10 == 0:
        loss_list.append(loss)
        print('after ',str(i),' iterations, the loss is ',str(loss[0]))
        prob_cur = sigmoid(w,x_train)
        y_cur = np.where(prob_cur > 0.5,1,0)
        error_rate = np.sum(np.absolute(y_train[:,np.newaxis] - y_cur),axis = 0) / x
        error_list.append(error_rate)
        print('error rate is ',str(error_rate[0]))
    temp1 = (y_train[:,np.newaxis] - prob) * x_train
    gradient = np.sum(temp1,axis = 0)
    w = w + alpha * gradient[:,np.newaxis]
loss_list = np.array(loss_list)
error_list = np.array(error_list)
```

```
after  0  iterations, the loss is  0.6925897569617911
error rate is  0.49642857142857144
after  10  iterations, the loss is  0.4482931798418664
error rate is  0.10857142857142857
after  20  iterations, the loss is  0.3591451399418551
error rate is  0.08928571428571429
after  30  iterations, the loss is  0.3123788785022366
error rate is  0.08071428571428571
after  40  iterations, the loss is  0.28301422271796883
error rate is  0.07785714285714286
after  50  iterations, the loss is  0.26259392999591896
error rate is  0.07357142857142857
after  60  iterations, the loss is  0.24742817219668017
error rate is  0.07142857142857142
after  70  iterations, the loss is  0.23563585170006418
error rate is  0.07071428571428572
after  80  iterations, the loss is  0.22615127023652543
error rate is  0.06714285714285714
after  90  iterations, the loss is  0.2183225812085373
```

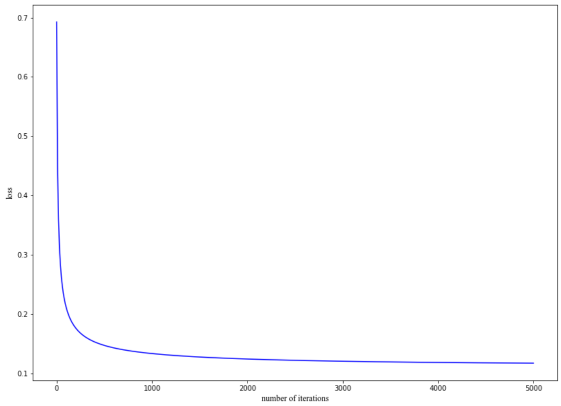
In [29]:

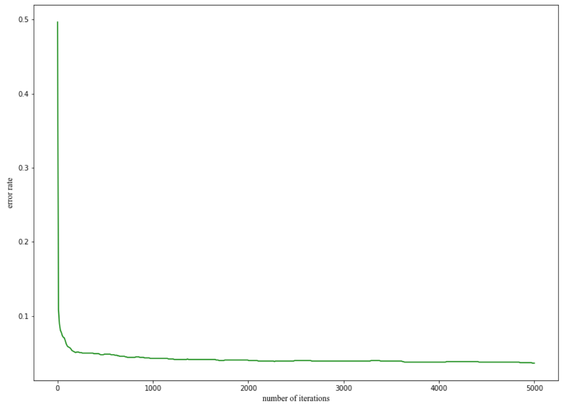
```
x = np.linspace(0,5000,500)
```

As we can see from the plot, as the number of iterations increase, the loss generally decreases and converges. Also the error rate generally decreases along with the increase of iteration number.

In [30]:

```
fig=plt.figure(figsize=(30,10))
fig.add_subplot(1,2,1)
plt.plot(x,loss_list,'b')
plt.xlabel('number of iterations', fontdict={'family' : 'Times New Roman', 'size'
plt.ylabel('loss', fontdict={'family' : 'Times New Roman', 'size'    : 12})
fig.add_subplot(1,2,2)
plt.plot(x,error_list,'g')
plt.xlabel('number of iterations', fontdict={'family' : 'Times New Roman', 'size'
plt.ylabel('error rate', fontdict={'family' : 'Times New Roman', 'size'    : 12})
plt.show()
```





In [31]:

```
# optimized weight vector
for i in range(w.shape[0]):
    print('w%d = %.3f'%(i,w[i][0]),end = '\t')
    if i % 3 == 2:
        print()
```

```
w0 = -0.890     w1 = -1.392     w2 = -1.151
w3 = -1.099     w4 = -0.745     w5 = -0.773
w6 = 0.816      w7 = 1.700      w8 = 0.068
w9 = -0.093     w10 = 0.202     w11 = -0.069
w12 = -0.336    w13 = 0.685     w14 = -1.218
w15 = -1.276    w16 = 3.219     w17 = 1.363
w18 = 1.352     w19 = 0.220     w20 = 0.625
w21 = -1.914    w22 = -2.384    w23 = -2.429
w24 = 0.785     w25 = 0.407     w26 = 0.553
w27 = -0.268    w28 = -0.488    w29 = -2.156
w30 = 0.354     w31 = -0.029    w32 = 0.474
w33 = 1.041     w34 = 0.046     w35 = -0.318
w36 = -0.626    w37 = -0.205    w38 = -0.398
w39 = -0.322    w40 = 1.119     w41 = -0.188
w42 = -0.313    w43 = -0.062    w44 = 0.104
w45 = -0.807    w46 = 0.762     w47 = -1.410
w48 = 1.365     w49 = -0.596    w50 = 1.241
w51 = 0.554     w52 = 0.401     w53 = -0.291
w54 = 0.216     w55 = -1.132    w56 = 0.524
w57 = 0.277     w58 = 0.866     w59 = 1.668
w60 = 0.476     w61 = 0.628     w62 = 0.521
w63 = -0.461
```

In [32]:

```
# test data
label_test3 = np.zeros(test3.shape[0])
label_test5 = np.ones(test5.shape[0])
test3 = np.concatenate((test3,label_test3[:,np.newaxis]),axis = 1)
test5 = np.concatenate((test5,label_test5[:,np.newaxis]),axis = 1)
test = np.concatenate((test3,test5))
np.random.shuffle(test)
x_test = test[:,:-1]
y_test = test[:,-1]
print(x_test.shape,y_test.shape)
```

```
(800, 64) (800,)
```

In [33]:

```
# using the optimized weight vector to predict test data sample
prob_test = sigmoid(w,x_test)
pred_test = np.where(prob_test > 0.5,1,0)
error_rate = np.sum(np.absolute(y_test[:,np.newaxis] - pred_test),axis = 0) / x_test
print("testing error rate ",str(error_rate[0]))
```

```
testing error rate  0.06
```

In [ ]:

```

```