Sheng Xu

In [1]:

```
from LRGradientDescent import LogisticRegressionGradientDescent as LRGD
import numpy as np
from scipy.special import logsumexp
from scipy.special import expit as sigm #sigmoid function

import os
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
```

In [2]:

```
## Toy problem
    #
    # Logistic regression should be able to perfectly predict all 10 examples
    # five examples have x values within (-2, -1) and are labeled 0
    # five examples have x values within (+1, +2) and are labeled 1
    N = 10
    x_NF = np.hstack([np.linspace(-2, -1, 5), np.linspace(1,2, 5)])[:,np.newaxis]
    y_N = np.hstack([np.zeros(5), 1.0 * np.ones(5)])

    lr = LRGD(
        alpha=0.1, step_size=0.1, init_w_recipe='zeros')

    # Prepare features by inserting column of all 1
    xbias_NG = lr.insert_final_col_of_all_ones(x_NF)
```

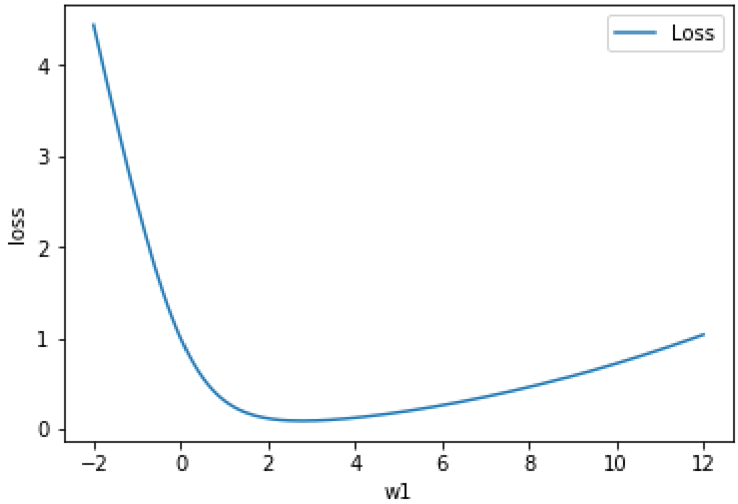
1a

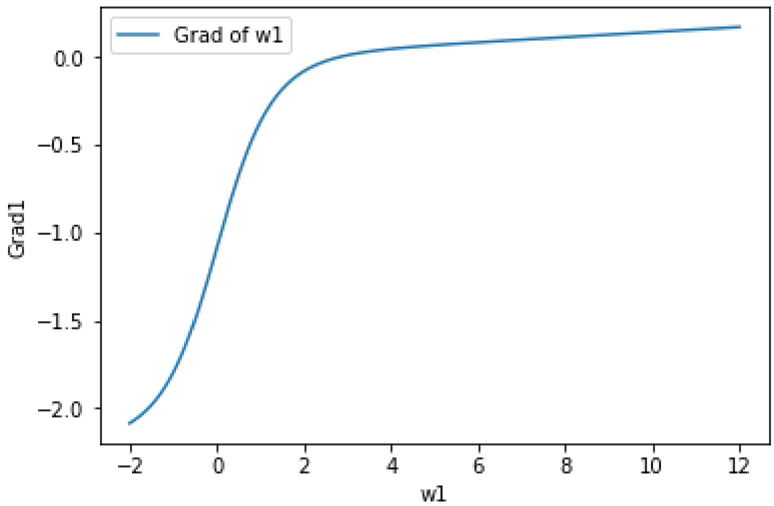
In [3]:

```
    loss=[]; grad=[];
    arr=np.linspace(-2, 12, 141)
    for i in arr:
        w=np.array([i,0])
        loss.append(lr.calc_loss(w, xbias_NG, y_N))
        grad.append(lr.calc_grad(w, xbias_NG, y_N)[0])
    #print (loss, grad)
    id_min_cost=np.argmin(loss)
```

In [4]:

```
plt.plot(arr, loss, label='Loss')
plt.xlabel('w1');
plt.ylabel('loss');
plt.legend();
plt.show();

plt.plot(arr, grad, label='Grad of w1')
plt.xlabel('w1');
plt.ylabel('Grad1');
plt.legend();
plt.show();
```





Yes.
For loss:
We know that when w1 is less than 0, the loss is big because the value of log_loss is big, because $w * x$ is negative, which leads to wrong classification.
On the other hand, when w1 is positive, the loss is bigger because the l2_penalty goes bigger as w1 is bigger.

For Gradient:
From the information of loss, we know: the partial derivative of w1 is negative when $w1 < 0$, it will reach 0 somewhere positive. Then, it goes positive when w1 grows bigger.

The minimum is somewhere between 2 and 3, close to 3.
Here, my estimation is 2.8. (See below)

In [5]:

```
print("best w1 for LR with 1 feature and 0 bias: %.3f" % arr[id_min_cost])
```

```
best w1 for LR with 1 feature and 0 bias: 2.800
```

In [6]:

```
lr.fit(x_NF, y_N)
```

```
Initializing w_G with 2 features using recipe: zeros
Running up to 10000 iters of gradient descent with step_size 0.1
iter    0/10000  loss         1.000000  avg_L1_norm_grad         0.541011
w[0]    0.000 bias    0.000
iter    1/10000  loss         0.888015  avg_L1_norm_grad         0.494016
w[0]    0.108 bias    0.000
iter    2/10000  loss         0.794586  avg_L1_norm_grad         0.451748
w[0]    0.207 bias    0.000
iter    3/10000  loss         0.716373  avg_L1_norm_grad         0.414112
w[0]    0.297 bias    0.000
iter    4/10000  loss         0.650555  avg_L1_norm_grad         0.380787
w[0]    0.380 bias    0.000
iter    5/10000  loss         0.594813  avg_L1_norm_grad         0.351344
w[0]    0.456 bias    0.000
iter    6/10000  loss         0.547278  avg_L1_norm_grad         0.325330
w[0]    0.527 bias    0.000
iter    7/10000  loss         0.506451  avg_L1_norm_grad         0.302308
w[0]    0.592 bias    0.000
iter    8/10000  loss         0.471140  avg_L1_norm_grad         0.281878
```

1b

In [7]:

```
print(" Result for LR with 1 feature and 0 bias: ", lr.trace_w[-1])
```

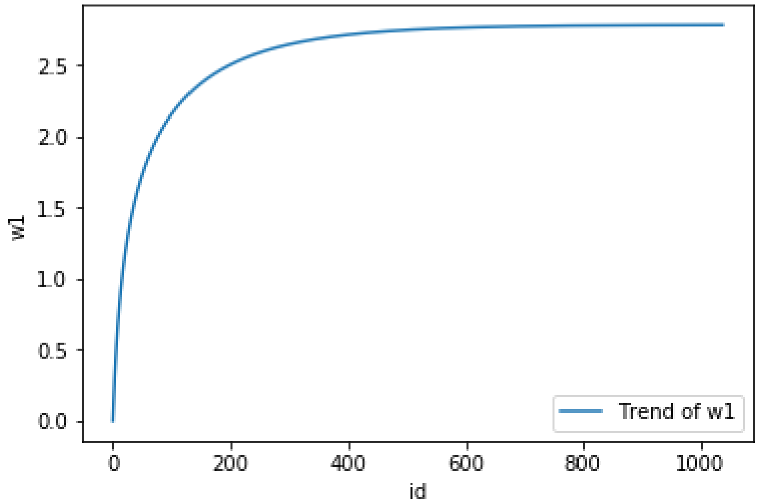
```
 Result for LR with 1 feature and 0 bias:  [2.78265835e+00 1.02172570e-17]
```

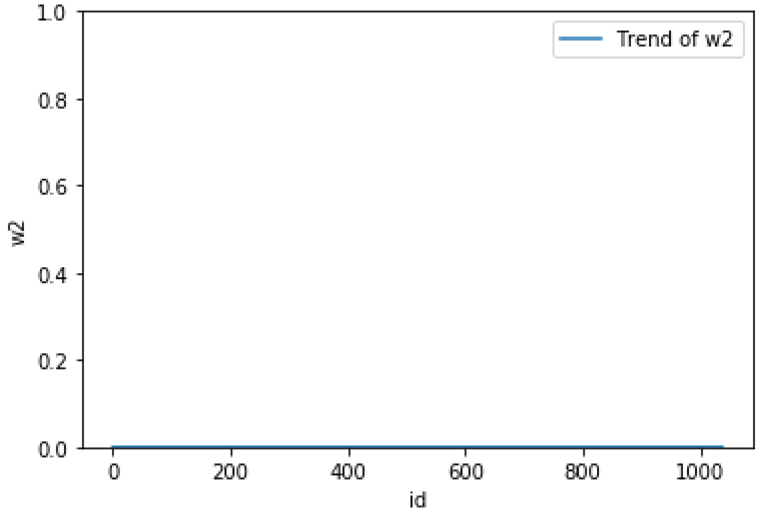
In [8]:

```
matrix=np.matrix(lr.trace_w).T
w1=np.asarray(matrix[0])[-1]
w2=np.asarray(matrix[1])[-1]
idx=np.linspace(0, w1.size-1,w1.size)
```

In [9]:

```
## Draw Picture
plt.plot(idx, w1, label='Trend of w1')
plt.xlabel('id');
plt.ylabel('w1');
plt.legend();
plt.show();

plt.plot(idx, w2, label='Trend of w2')
plt.xlabel('id');
plt.ylabel('w2');
plt.ylim([0.0, 1.0]);
plt.legend();
plt.show();
```





Yes.
w1 approaches 2.78. The converging speed is fast at first, then becomes really slow before w1 reaches the final result.
w2 stays at nearly 0 because it should be 0 from the symetricity of the question.

1c

In [10]:

```
lr2 = LRGD(
        alpha=0.1, step_size=0.1, init_w_recipe='uniform_-1_to_1')
lr2.fit(x_NF, y_N)
```

```
Initializing w_G with 2 features using recipe: uniform_-1_to_1
Running up to 10000 iters of gradient descent with step_size 0.1
iter    0/10000  loss         0.932814  avg_L1_norm_grad         0.579579
w[0]    0.098 bias    0.430
iter    1/10000  loss         0.834339  avg_L1_norm_grad         0.534178
w[0]    0.198 bias    0.415
iter    2/10000  loss         0.751280  avg_L1_norm_grad         0.492499
w[0]    0.290 bias    0.400
iter    3/10000  loss         0.681012  avg_L1_norm_grad         0.454653
w[0]    0.374 bias    0.385
iter    4/10000  loss         0.621302  avg_L1_norm_grad         0.420533
w[0]    0.452 bias    0.372
iter    5/10000  loss         0.570293  avg_L1_norm_grad         0.389904
w[0]    0.523 bias    0.360
iter    6/10000  loss         0.526460  avg_L1_norm_grad         0.362461
w[0]    0.590 bias    0.348
iter    7/10000  loss         0.488565  avg_L1_norm_grad         0.337881
w[0]    0.651 bias    0.337
iter    8/10000  loss         0.455603  avg_L1_norm_grad         0.315844
```

In [11]:

```
print(" Result for LR with 1 feature and 0 bias: ", lr2.trace_w[-1])
```

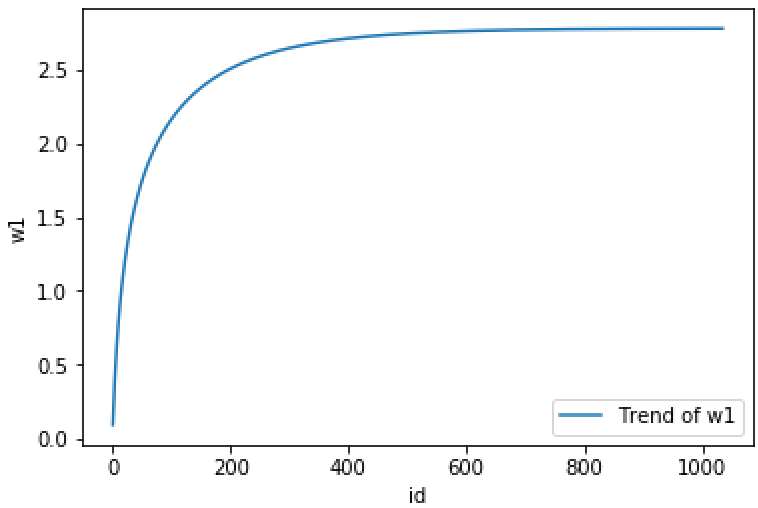
```
 Result for LR with 1 feature and 0 bias:  [2.78266077e+00 9.11289218e-04]
```

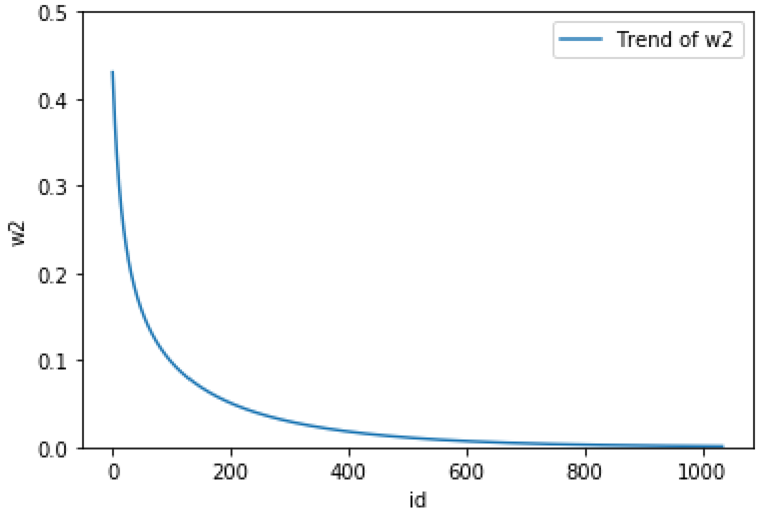
In [12]:

```
matrix=np.matrix(lr2.trace_w).T
w1=np.asarray(matrix[0])[-1]
w2=np.asarray(matrix[1])[-1]
idx=np.linspace(0, w1.size-1,w1.size)
```

In [14]:

```
## Draw Picture
plt.plot(idx, w1, label='Trend of w1')
plt.xlabel('id');
plt.ylabel('w1');
plt.legend();
plt.show();

plt.plot(idx, w2, label='Trend of w2')
plt.xlabel('id');
plt.ylabel('w2');
plt.ylim([0.0, 0.5]);
plt.legend();
plt.show();
```





Yes.
w1 approaches 2.78. The converging speed is fast at first, then becomes really slow before w1 is stablized.
w2 approaches 0. The converging speed is fast at first, then becomes really slow before w2 is stabalized.

In [ ]:

In [1]:

```
from LRGradientDescent import LogisticRegressionGradientDescent as LRGD
from show_images import show_images
import numpy as np
from scipy.special import logsumexp
from scipy.special import expit as sigm #sigmoid function
from numpy import genfromtxt
from matplotlib import pyplot as plt

import os
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns

import sklearn.linear_model
import sklearn.tree
import sklearn.metrics

from scipy.special import expit as sigm
```

In [2]:

```
def calc_TP_TN_FP_FN(ytrue_N, yhat_N):
    ''' Compute counts of four possible outcomes of a binary classifier for evaluation.

    Args
    ----
    ytrue_N : 1D array of floats
        Each entry represents the binary value (0 or 1) of 'true' label of one example
        One entry per example in current dataset
    yhat_N : 1D array of floats
        Each entry represents a predicted binary value (either 0 or 1).
        One entry per example in current dataset.
        Needs to be same size as ytrue_N.

    Returns
    -------
    TP : float
        Number of true positives
    TN : float
        Number of true negatives
    FP : float
        Number of false positives
    FN : float
        Number of false negatives
    '''
    TP = 0.0
    TN = 0.0
    FP = 0.0
    FN = 0.0
    FP_id=[]
    FN_id=[]
    l=ytrue_N.size
    for i in range(0,l):
        if (yhat_N[i]==1):
            if (ytrue_N[i]==1):
                TP=TP+1.0
            else:
                FP=FP+1.0
                FP_id.append(i)
        else:
            if (ytrue_N[i]==0):
                TN=TN+1.0
            else:
                FN=FN+1.0
                FN_id.append(i)
    return TP, TN, FP, FN, FP_id, FN_id

def calc_perf_metrics_for_threshold(ytrue_N, yproba1_N, thresh):
    ''' Compute performance metrics for a given probabilistic classifier and threshold
    '''
    tp, tn, fp, fn, FPSample, FNSample = calc_TP_TN_FP_FN(ytrue_N, yproba1_N >= thresh)
    ## Compute ACC, TPR, TNR, etc.
    acc = (tp + tn) / float(tp + tn + fp + fn + 1e-10)
    tpr = tp / float(tp + fn + 1e-10)
    tnr = tn / float(fp + tn + 1e-10)
    ppv = tp / float(tp + fp + 1e-10)
    npv = tn / float(tn + fn + 1e-10)

    return acc, tpr, tnr, ppv, npv, FPSample, FNSample
```

```
def print_perf_metrics_for_threshold(ytrue_N, yproba1_N, thresh):
    ''' Pretty print perf. metrics for a given probabilistic classifier and threshold
    '''
    acc, tpr, tnr, ppv, npv = calc_perf_metrics_for_threshold(ytrue_N, yproba1_N, thresh)

    ## Pretty print the results
    print("%.3f ACC" % acc)
    print("%.3f TPR" % tpr)
    print("%.3f TNR" % tnr)
    print("%.3f PPV" % ppv)
    print("%.3f NPV" % npv)

def calc_confusion_matrix_for_threshold(ytrue_N, yproba1_N, thresh):
    ''' Compute the confusion matrix for a given probabilistic classifier and threshold

    Args
    ----
    ytrue_N : 1D array of floats
        Each entry represents the binary value (0 or 1) of 'true' label of one example
        One entry per example in current dataset
    yproba1_N : 1D array of floats
        Each entry represents a probability (between 0 and 1) that correct label is positiv
        One entry per example in current dataset
        Needs to be same size as ytrue_N
    thresh : float
        Scalar threshold for converting probabilities into hard decisions
        Calls an example "positive" if yproba1 >= thresh

    Returns
    -------
    cm_df : Pandas DataFrame
        Can be printed like print(cm_df) to easily display results
    '''

    #print(ytrue_N, yproba1_N >= thresh)
    cm = sklearn.metrics.confusion_matrix(ytrue_N, yproba1_N >= thresh)
    cm_df = pd.DataFrame(data=cm, columns=[0, 1], index=[0, 1])
    cm_df.columns.name = 'Predicted'
    cm_df.index.name = 'True'
    return cm_df
```

In [3]:

```
x_all= genfromtxt('data_digits_8_vs_9_noisy/x_train.csv', delimiter=',')[1:]
#xbias_NG = lr.insert_final_col_of_all_ones(x_all)
y_all= genfromtxt('data_digits_8_vs_9_noisy/y_train.csv', delimiter=',')[1:]
```

In [4]:

```
#print(x_all[0:11,0:10])
x_tr=x_all[2000:]
x_va=x_all[:2000]
#print(x_va.shape[0])
y_tr=y_all[2000:]
y_va=y_all[:2000]
```

In [5]:

```
#Moderate Step Size
lrM = LRGD(alpha=10.0, step_size=0.5, num_iterations=10000,init_w_recipe='zeros')
lrM.fit(x_tr, y_tr)
```

```
Initializing w_G with 785 features using recipe: zeros
Running up to 10000 iters of gradient descent with step_size 0.5
iter    0/10000  loss         1.000000  avg_L1_norm_grad         0.024676
w[0]    0.000 bias    0.000
iter    1/10000  loss         0.652834  avg_L1_norm_grad         0.058458
w[0]   -0.001 bias    0.002
iter    2/10000  loss         4.480393  avg_L1_norm_grad         0.167414
w[0]    0.012 bias    0.145
iter    3/10000  loss         9.642480  avg_L1_norm_grad         0.151994
w[0]   -0.025 bias   -0.212
iter    4/10000  loss         1.741418  avg_L1_norm_grad         0.129421
w[0]    0.010 bias    0.151
iter    5/10000  loss         6.221672  avg_L1_norm_grad         0.151786
w[0]   -0.018 bias   -0.121
iter    6/10000  loss         2.955412  avg_L1_norm_grad         0.146658
w[0]    0.016 bias    0.241
iter    7/10000  loss         4.446966  avg_L1_norm_grad         0.147311
w[0]   -0.016 bias   -0.069
iter    8/10000  loss         2.279642  avg_L1_norm_grad         0.115011
```

In [6]:

```
#Small Step Size
lrS = LRGD(alpha=10.0, step_size=0.1, num_iterations=10000,init_w_recipe='zeros')
lrS.fit(x_tr, y_tr)
```

```
Initializing w_G with 785 features using recipe: zeros
Running up to 10000 iters of gradient descent with step_size 0.1
iter    0/10000  loss         1.000000  avg_L1_norm_grad         0.024676
w[0]    0.000 bias    0.000
iter    1/10000  loss         0.870389  avg_L1_norm_grad         0.024782
w[0]   -0.000 bias    0.000
iter    2/10000  loss         0.773694  avg_L1_norm_grad         0.021132
w[0]    0.000 bias    0.009
iter    3/10000  loss         0.700361  avg_L1_norm_grad         0.022997
w[0]   -0.000 bias    0.006
iter    4/10000  loss         0.642664  avg_L1_norm_grad         0.020912
w[0]    0.001 bias    0.016
iter    5/10000  loss         0.595566  avg_L1_norm_grad         0.021399
w[0]   -0.000 bias    0.011
iter    6/10000  loss         0.554936  avg_L1_norm_grad         0.018142
w[0]    0.001 bias    0.021
iter    7/10000  loss         0.520284  avg_L1_norm_grad         0.016834
w[0]    0.000 bias    0.017
iter    8/10000  loss         0.490452  avg_L1_norm_grad         0.012709
```

In [7]:

```
#Large Step Size
lrL = LRGD(alpha=10.0, step_size=1, num_iterations=10000,init_w_recipe='zeros')
lrL.fit(x_tr, y_tr)
```

```
nitializing w_G with 785 features using recipe: zeros
unning up to 10000 iters of gradient descent with step_size 1
ter    0/10000  loss         1.000000  avg_L1_norm_grad         0.024676  w
0]    0.000 bias    0.000
ter    1/10000  loss         0.680851  avg_L1_norm_grad         0.073857  w
0]   -0.002 bias    0.004
ter    2/10000  loss        12.430986  avg_L1_norm_grad         0.168269  w
0]    0.032 bias    0.358
ter    3/10000  loss        16.612874  avg_L1_norm_grad         0.152061  w
0]   -0.043 bias   -0.360
ter    4/10000  loss         6.307122  avg_L1_norm_grad         0.165638  w
0]    0.027 bias    0.366
ter    5/10000  loss        16.392406  avg_L1_norm_grad         0.152081  w
0]   -0.046 bias   -0.340
ter    6/10000  loss         2.325717  avg_L1_norm_grad         0.086335  w
0]    0.024 bias    0.386
ter    7/10000  loss         2.622341  avg_L1_norm_grad         0.089287  w
0]   -0.013 bias    0.026
ter    8/10000  loss         2.359122  avg_L1_norm_grad         0.081969  w
0]    0.027 bias    0.453
ter    9/10000  loss         1.626583  avg_L1_norm_grad         0.057398  w
0]   -0.008 bias    0.112
ter   10/10000  loss         0.688730  avg_L1_norm_grad         0.023235  w
0]    0.019 bias    0.387
ter   11/10000  loss         0.417238  avg_L1_norm_grad         0.004829  w
0]    0.008 bias    0.292
ter   12/10000  loss         0.391655  avg_L1_norm_grad         0.002510  w
0]    0.010 bias    0.317
ter   13/10000  loss         0.379486  avg_L1_norm_grad         0.002436  w
0]    0.008 bias    0.322
ter   14/10000  loss         0.368143  avg_L1_norm_grad         0.002360  w
0]    0.007 bias    0.327
ter   15/10000  loss         0.357537  avg_L1_norm_grad         0.002289  w
0]    0.006 bias    0.333
ter   16/10000  loss         0.347592  avg_L1_norm_grad         0.002224  w
0]    0.005 bias    0.338
ter   17/10000  loss         0.338240  avg_L1_norm_grad         0.002163  w
0]    0.004 bias    0.342
ter   18/10000  loss         0.329419  avg_L1_norm_grad         0.002107  w
0]    0.003 bias    0.347
ter   19/10000  loss         0.321077  avg_L1_norm_grad         0.002055  w
0]    0.002 bias    0.352
ter  100/10000  loss         0.122988  avg_L1_norm_grad         0.000563  w
0]   -0.063 bias    0.554
ter  101/10000  loss         0.122520  avg_L1_norm_grad         0.000556  w
0]   -0.063 bias    0.555
ter  200/10000  loss         0.105269  avg_L1_norm_grad         0.000200  w
0]   -0.072 bias    0.663
ter  201/10000  loss         0.105218  avg_L1_norm_grad         0.000199  w
0]   -0.072 bias    0.664
ter  300/10000  loss         0.102529  avg_L1_norm_grad         0.000109  w
0]   -0.067 bias    0.752
ter  301/10000  loss         0.102514  avg_L1_norm_grad         0.000108  w
0]   -0.067 bias    0.753
ter  400/10000  loss         0.101512  avg_L1_norm_grad         0.000073  w
```

```
0]   -0.061 bias    0.829
ter  401/10000  loss         0.101505  avg_L1_norm_grad        0.000072  w
0]   -0.061 bias    0.830
LERT! Divergence detected. Loss is increasing but should be decreasing!
ecent history of loss values:
ter  488  loss         0.109160
ter  489  loss         0.112070
ter  490  loss         0.115540
ter  491  loss         0.121522
ter  492  loss         0.129152
ter  493  loss         0.144005
ter  494  loss         0.167084
ter  495  loss         0.224993
ter  496  loss         0.388263
ter  497  loss         1.382206

---------------------------------------------------------------------------
alueError                                 Traceback (most recent call last)
ipython-input-7-5b01a4a41bcf> in <module>
      1 #Large Step Size
      2 lrL = LRGD(alpha=10.0, step_size=1, num_iterations=10000,init_w_reci
e='zeros')
----> 3 lrL.fit(x_tr, y_tr)

~\Documents\comp135-19s-assignments-master\project1\LRGradientDescent.py in
it(self, x_NF, y_N)
    238             ## Assess divergence and raise ValueError as soon as it
happens
    239             self.raise_error_if_diverging(
--> 240                 self.trace_steps, self.trace_loss, self.trace_L1_nor
_of_grad)
    241
    242             ## Assess convergence and break early if happens

~\Documents\comp135-19s-assignments-master\project1\LRGradientDescent.py in
aise_error_if_diverging(self, trace_steps, trace_loss, trace_L1_norm_of_gra
)
    536                     trace_steps[-M+ii], trace_loss[-M+ii]))
    537             raise ValueError("Divergence detected. %s. %s." % (
--> 538                 reason_str, hint_str))
    539
    540

alueError: Divergence detected. Loss is increasing but should be decreasin
!. Try a smaller step_size than current value 1.000e+00.
```

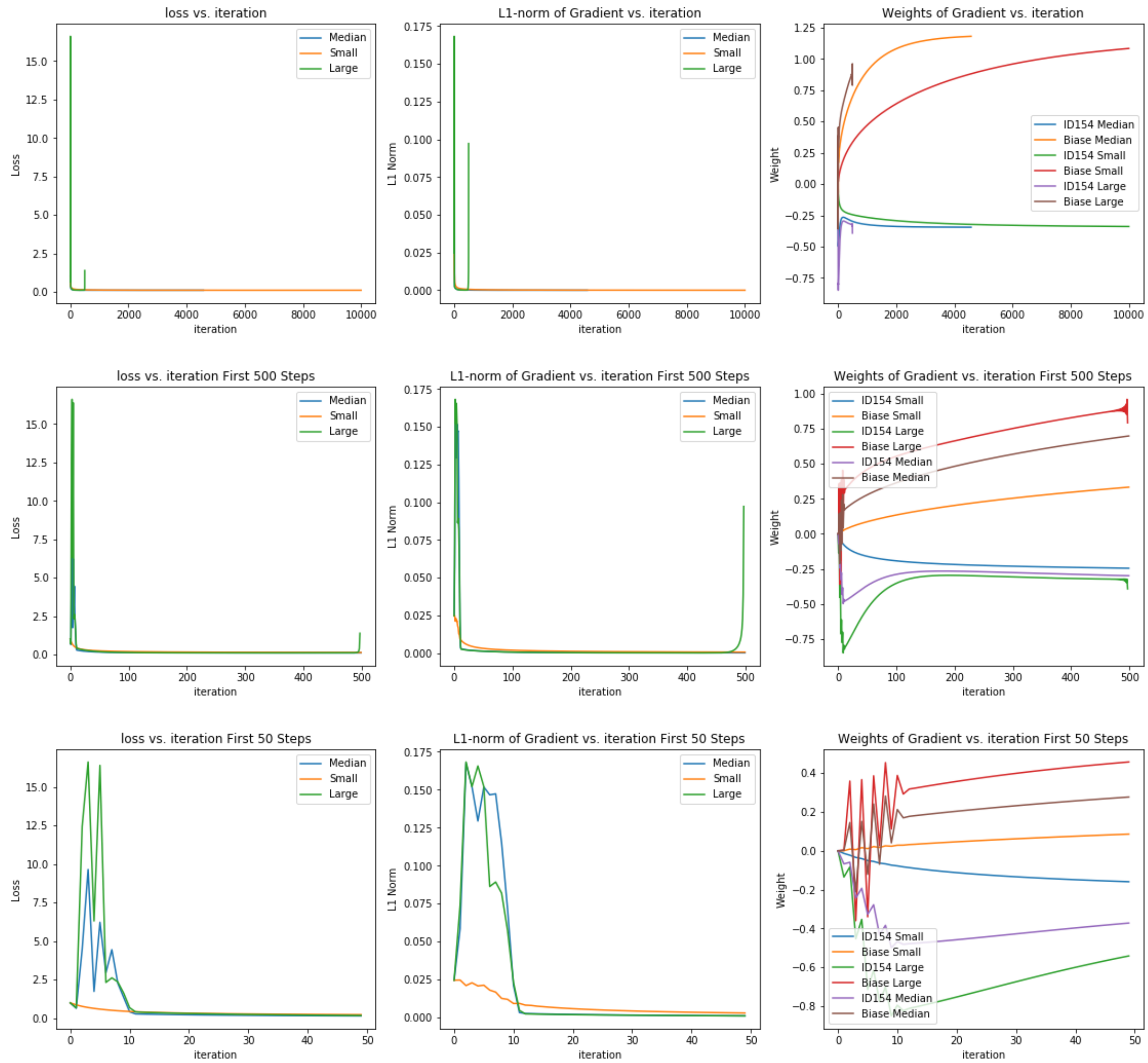
In [8]:

```
#Moderate Step Size
lossM=np.array(lrM.trace_loss)
gdL1M=np.array(lrM.trace_L1_norm_of_grad)
w154M=np.asarray(np.matrix(lrM.trace_w).T[154])[-1]
wbiaM=np.asarray(np.matrix(lrM.trace_w).T[-1] )[-1]
indxM=range(0,lossM.size)

#Small Step Size
lossS=np.array(lrS.trace_loss)
gdL1S=np.array(lrS.trace_L1_norm_of_grad)
w154S=np.asarray(np.matrix(lrS.trace_w).T[154])[-1]
wbiaS=np.asarray(np.matrix(lrS.trace_w).T[-1] )[-1]
indxS=range(0,lossS.size)

#Large Step Size
lossL=np.array(lrL.trace_loss)
gdL1L=np.array(lrL.trace_L1_norm_of_grad)
w154L=np.asarray(np.matrix(lrL.trace_w).T[154])[-1]
wbiaL=np.asarray(np.matrix(lrL.trace_w).T[-1] )[-1]
indxL=range(0,lossL.size)
```

In [9]:

```
fig2a, axes_arr = plt.subplots(nrows=1, ncols=3,figsize=(19,5))

ax1=axes_arr[0]
ax1.set_title('loss vs. iteration'); ax1.set_xlabel('iteration'); ax1.set_ylabel('Loss');

ax1.plot(indxM, lossM, label='Median');
ax1.plot(indxS, lossS, label='Small');
ax1.plot(indxL, lossL, label='Large');
ax1.legend();

ax2=axes_arr[1]
ax2.set_title('L1-norm of Gradient vs. iteration'); ax2.set_xlabel('iteration'); ax2.set_yl
ax2.plot(indxM, gdL1M, label='Median');
ax2.plot(indxS, gdL1S, label='Small');
ax2.plot(indxL, gdL1L, label='Large');
ax2.legend();

ax3=axes_arr[2]
ax3.set_title('Weights of Gradient vs. iteration'); ax3.set_xlabel('iteration'); ax3.set_yl
ax3.plot(indxM, w154M, label='ID154 Median')
ax3.plot(indxM, wbiaM, label='Biase Median')
ax3.plot(indxS, w154S, label='ID154 Small')
ax3.plot(indxS, wbiaS, label='Biase Small')
ax3.plot(indxL, w154L, label='ID154 Large')
ax3.plot(indxL, wbiaL, label='Biase Large')
ax3.legend();

fig2a2, axes_arr2 = plt.subplots(nrows=1, ncols=3,figsize=(19,5))

ax1=axes_arr2[0]
ax1.set_title('loss vs. iteration First 500 Steps'); ax1.set_xlabel('iteration'); ax1.set_y

ax1.plot(indxM[:500], lossM[:500], label='Median');
ax1.plot(indxS[:500], lossS[:500], label='Small');
ax1.plot(indxL, lossL, label='Large');
ax1.legend();

ax2=axes_arr2[1]
ax2.set_title('L1-norm of Gradient vs. iteration First 500 Steps'); ax2.set_xlabel('iterati
ax2.plot(indxM[:500], gdL1M[:500], label='Median');
ax2.plot(indxS[:500], gdL1S[:500], label='Small');
ax2.plot(indxL, gdL1L, label='Large');
ax2.legend();

ax3=axes_arr2[2]
ax3.set_title('Weights of Gradient vs. iteration First 500 Steps'); ax3.set_xlabel('iterati
ax3.plot(indxS[:500], w154S[:500], label='ID154 Small')
ax3.plot(indxS[:500], wbiaS[:500], label='Biase Small')
ax3.plot(indxL, w154L, label='ID154 Large')
ax3.plot(indxL, wbiaL, label='Biase Large')
ax3.plot(indxM[:500], w154M[:500], label='ID154 Median')
ax3.plot(indxM[:500], wbiaM[:500], label='Biase Median')
ax3.legend();

fig2a3, axes_arr3 = plt.subplots(nrows=1, ncols=3,figsize=(19,5))

ax1=axes_arr3[0]
ax1.set_title('loss vs. iteration First 50 Steps'); ax1.set_xlabel('iteration'); ax1.set_yl
```

```
ax1.plot(indxM[:50], lossM[:50], label='Median');
ax1.plot(indxS[:50], lossS[:50], label='Small');
ax1.plot(indxL[:50], lossL[:50], label='Large');
ax1.legend();

ax2=axes_arr3[1]
ax2.set_title('L1-norm of Gradient vs. iteration First 50 Steps'); ax2.set_xlabel('iteratio
ax2.plot(indxM[:50], gdL1M[:50], label='Median');
ax2.plot(indxS[:50], gdL1S[:50], label='Small');
ax2.plot(indxL[:50], gdL1L[:50], label='Large');
ax2.legend();

ax3=axes_arr3[2]
ax3.set_title('Weights of Gradient vs. iteration First 50 Steps'); ax3.set_xlabel('iteratio
ax3.plot(indxS[:50], w154S[:50], label='ID154 Small')
ax3.plot(indxS[:50], wbiaS[:50], label='Biase Small')
ax3.plot(indxL[:50], w154L[:50], label='ID154 Large')
ax3.plot(indxL[:50], wbiaL[:50], label='Biase Large')
ax3.plot(indxM[:50], w154M[:50], label='ID154 Median')
ax3.plot(indxM[:50], wbiaM[:50], label='Biase Median')
ax3.legend();
```



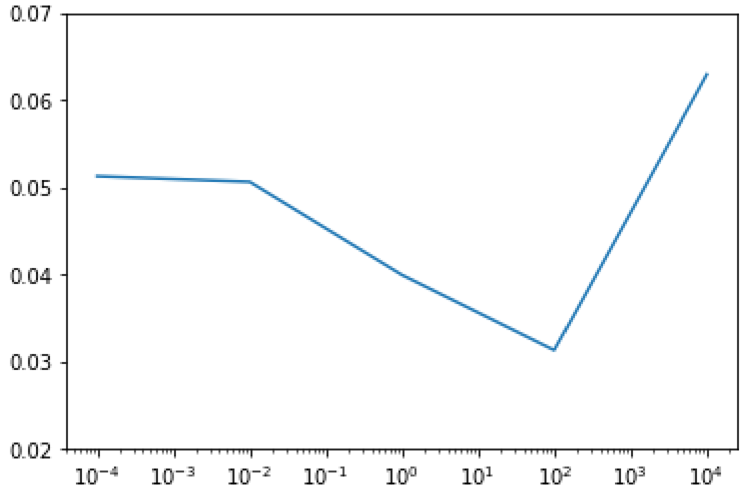
Explanation:

The small step size is 0.1. The large step size is 1. The moderate(median) step size I use is 0.5 (I tried 0.2, 0.5, 0.9. all worked).
As we are using gradient descent, the loss should be decreasing. However, the large step size makes the loss jumps and increases at first and near the desired weight. The median step size also increases the loss at the first few steps. You can also see significant squiggles in the weights for large and median step size at the first few steps.
I would recommend 0.5 to proceed.
The large step size of 1 will give a diverging result.

2B

In [10]:

```
alpha_grid=[0.0001, 0.01, 1, 100, 10000]
err_data= genfromtxt('tmp/all_cv_scores.csv', delimiter=',')[1:]
err=np.asarray(err_data[0:-1,2])
#print(err)
ave_err=[];
for i in range(err.size//3+1):
    ave_err.append(np.mean(err[3*i:3*i+2]))
plt.plot(alpha_grid, ave_err)
plt.xscale('log')
plt.ylim([0.02,0.07])
plt.show()
```



When α is small ($\leq 0, 01$), the error on validation set is large, probabaly because we're over fitting the training set.
When α is appropriate, the error is small. Here $\alpha = 100$ is the smallest.
When α is big (≥ 100), the error is big, probably because we are penalizing large weight and thus make the weight to close to zero. Thus, it's underfitting.
I would choose 100.

2C

In [8]:

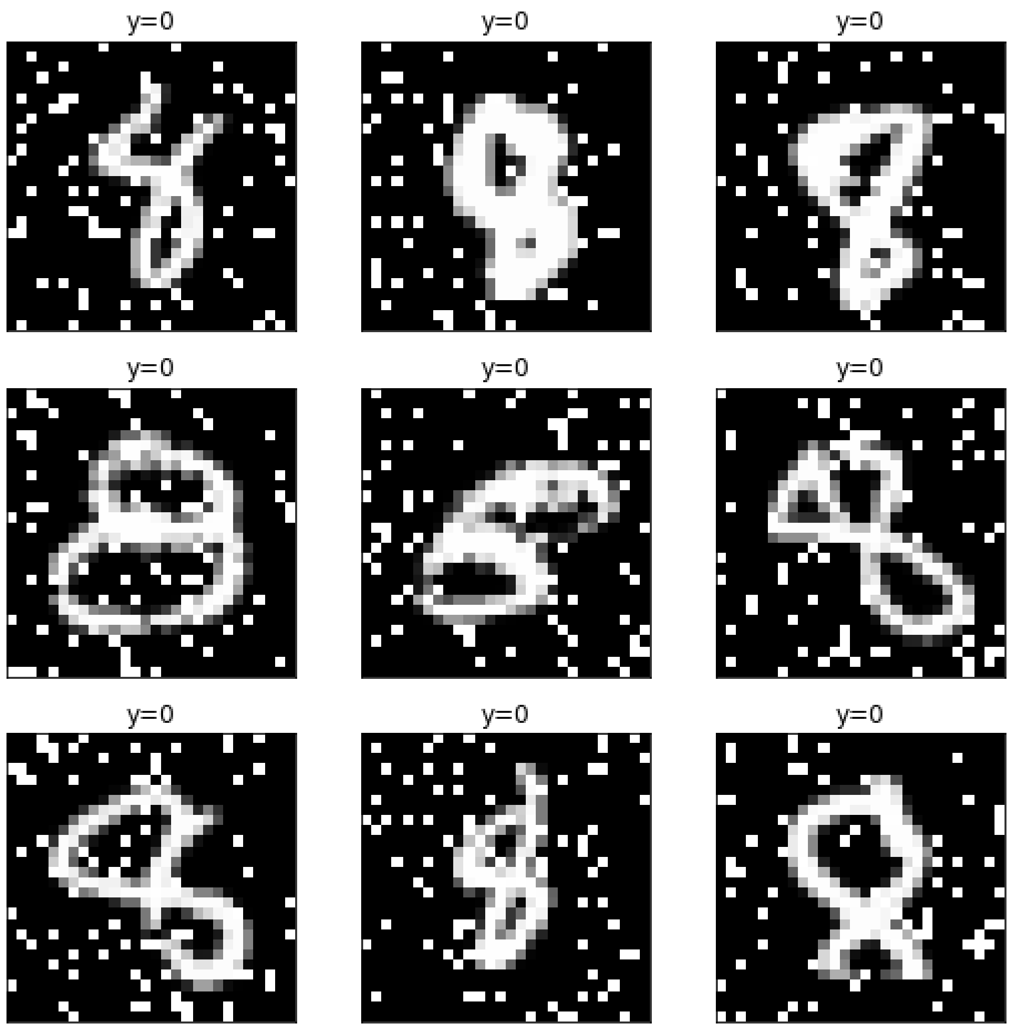
```
num_folder=3;
x_va_F=x_all[:int(np.ceil(x_all.shape[0]/num_folder))]
y_va_F=y_all[:int(np.ceil(x_all.shape[0]/num_folder))]
w_F=genfromtxt('tmp/alpha0100.0000_fold01_weights.txt', delimiter=',')
w=w_F[:-1]
b=w_F[-1]
y_pred=np.asarray(sigm(x_va_F.dot(w)+b)).reshape(-1)
TP,TN,FP,FN, FPSample, FNSample=calc_TP_TN_FP_FN(y_va_F, y_pred>=0.5)
print(calc_confusion_matrix_for_threshold(y_va_F, y_pred, 0.5))
```

```
Predicted     0     1
True
0          1860    62
1            69  1943
```

In [12]:

```
x_FP_9=x_va_F[FPSample]
y_FP_9=y_va_F[FPSample]
x_FN_9=x_va_F[FNSample]
y_FN_9=y_va_F[FNSample]
print("False Pos")
show_images(x_FP_9, y_FP_9)
```

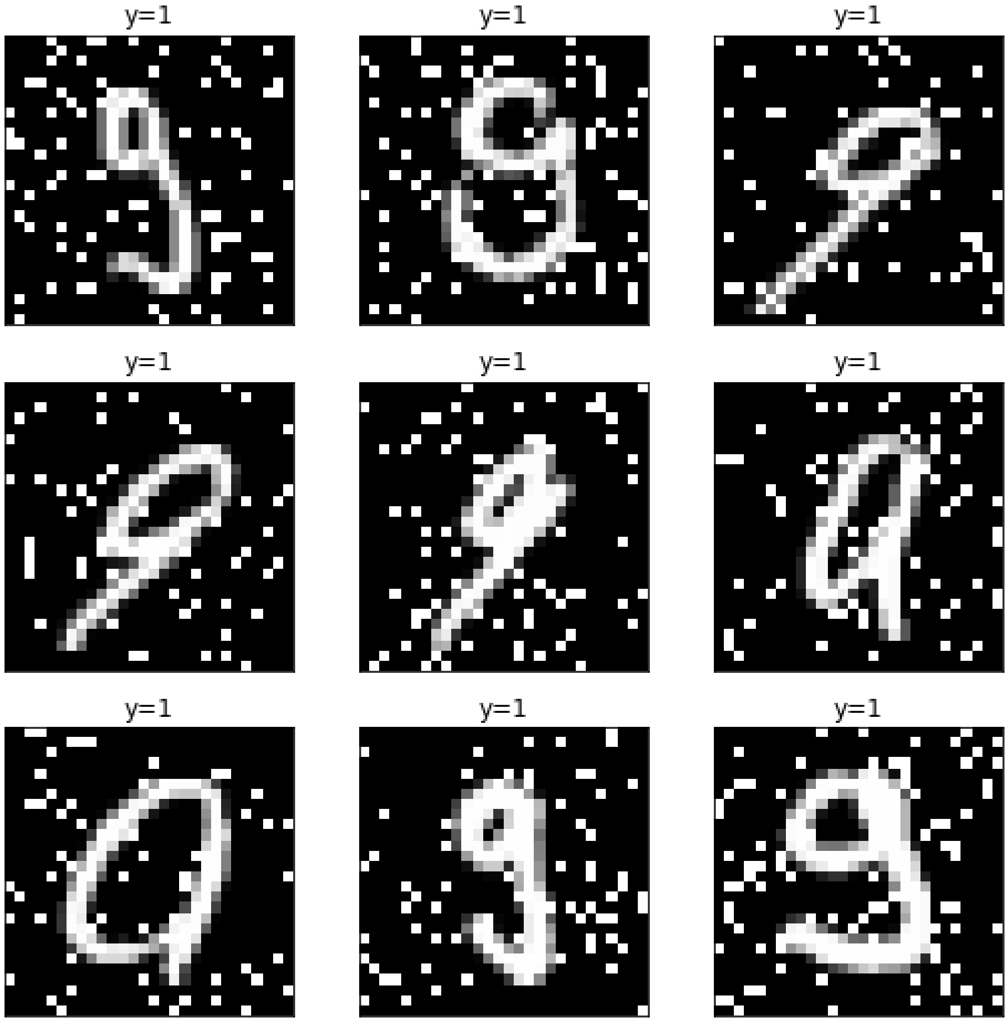
```
False Negative
```



In [13]:

```
print("False Neg")
show_images(x_FN_9, y_FN_9)
```

```
False Positive
```



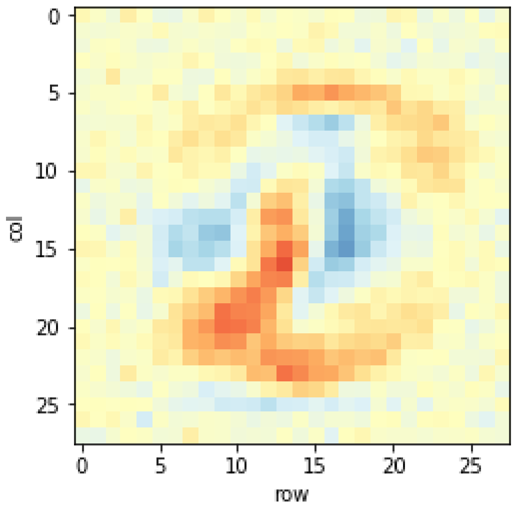
The classifier makes mistakes in some leaned cases (FN row 3 col 1, FP row 2 col 2). It also misclassifies some examples where 9 is written not like a shape of 'q'(FP row 3 col 3). or 8 is written like a "θ"(FN row 2 col 1). It can also get a wrong result when the image is a little incomplete.

The classifier misidentifies when the image has some key factors for the other subject. For example, if 9 is written in a very leaned lower part like "/" but not a straight "|" or has a horizontal line at the botton, it classifies 9 as an 8. This is a linear weighted model, so once those key features outweighted, the model makes the wrong prediction.

2D

In [15]:

```
plt.imshow(w.reshape(28,28), interpolation='nearest', vmin=-0.5, vmax=0.5, cmap='RdYlBu')
plt.xlabel('row');
plt.ylabel('col');
plt.show()
```



The plot shows that the middle area[10:25, 5:22] has strong connection with the classification results. There are pixels of deep blue/red that are connected with the positive/negative classification results.
For positive results (classified as 9), pixels[12:17, 5:10], pixels[10:17, 16:18] are significantly related.
For negative results (classified as 8), pixels[12:17, 12:14], pixels[17:22, 6:12], pixels[20:23, 11:16] are significantly related.

2E

In [14]:

```
lr = LRGD(alpha=100.0, step_size=0.5, num_iterations=10000,init_w_recipe='zeros')
lr.fit(x_all, y_all)
```

```
nitializing w_G with 785 features using recipe: zeros
unning up to 10000 iters of gradient descent with step_size 0.5
ter    0/10000  loss         1.000000  avg_L1_norm_grad         0.024599  w
0]    0.000 bias    0.000
ter    1/10000  loss         0.626452  avg_L1_norm_grad         0.053024  w
0]   -0.000 bias    0.003
ter    2/10000  loss         3.983615  avg_L1_norm_grad         0.166386  w
0]    0.012 bias    0.134
ter    3/10000  loss        10.109441  avg_L1_norm_grad         0.153411  w
0]   -0.024 bias   -0.221
ter    4/10000  loss         1.603762  avg_L1_norm_grad         0.123394  w
0]    0.011 bias    0.144
ter    5/10000  loss         5.980697  avg_L1_norm_grad         0.152966  w
0]   -0.015 bias   -0.116
ter    6/10000  loss         3.571271  avg_L1_norm_grad         0.155634  w
0]    0.019 bias    0.247
ter    7/10000  loss         4.940048  avg_L1_norm_grad         0.150363  w
0]   -0.014 bias   -0.083
ter    8/10000  loss         2.345486  avg_L1_norm_grad         0.114557  w
0]    0.020 bias    0.274
ter    9/10000  loss         1.466452  avg_L1_norm_grad         0.074123  w
0]   -0.005 bias    0.032
ter   10/10000  loss         0.565581  avg_L1_norm_grad         0.025197  w
0]    0.012 bias    0.209
ter   11/10000  loss         0.384964  avg_L1_norm_grad         0.004231  w
0]    0.006 bias    0.157
ter   12/10000  loss         0.371157  avg_L1_norm_grad         0.003117  w
0]    0.006 bias    0.166
ter   13/10000  loss         0.360935  avg_L1_norm_grad         0.003013  w
0]    0.006 bias    0.169
ter   14/10000  loss         0.351390  avg_L1_norm_grad         0.002917  w
0]    0.006 bias    0.171
ter   15/10000  loss         0.342457  avg_L1_norm_grad         0.002825  w
0]    0.005 bias    0.174
ter   16/10000  loss         0.334083  avg_L1_norm_grad         0.002738  w
0]    0.005 bias    0.176
ter   17/10000  loss         0.326218  avg_L1_norm_grad         0.002656  w
0]    0.005 bias    0.179
ter   18/10000  loss         0.318819  avg_L1_norm_grad         0.002578  w
0]    0.004 bias    0.181
ter   19/10000  loss         0.311845  avg_L1_norm_grad         0.002505  w
0]    0.004 bias    0.183
one. Converged after 753 iterations.
```

In [18]:

```
x_test_NF=genfromtxt('data_digits_8_vs_9_noisy/x_test.csv', delimiter=',')[1:]
yproba1_test_N = lr.predict_proba(x_test_NF)[:, 1]
np.savetxt('yproba1_test.txt', yproba1_test_N)
```

```
error_rate           0.033787191124558746
AUROC     0.9949214767299642
The error rate fits with the average error of CV(3.3%). The AUROC fits the ave AUROC of
CV(0.994).
```

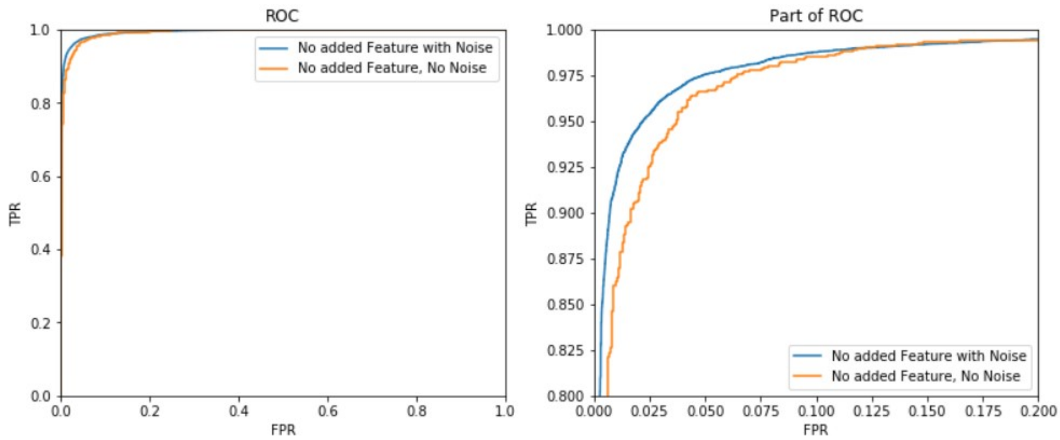
In [ ]:

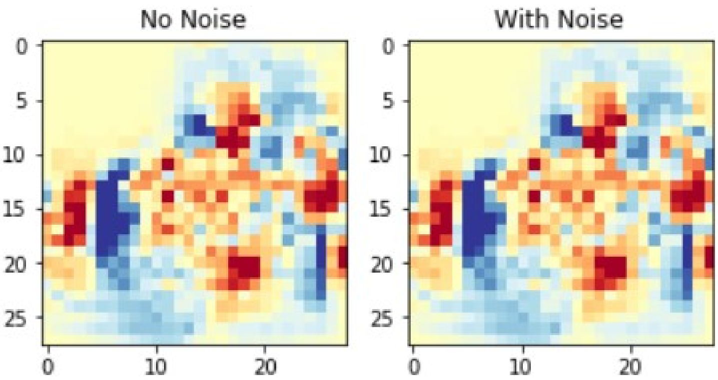
```
```

Problem3:

I used several ways to improve the performance of the model in a few ways:

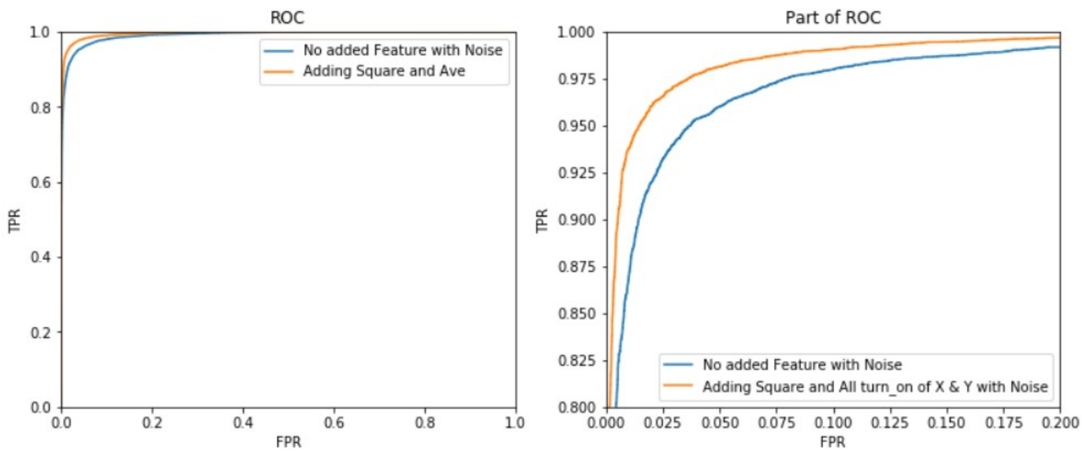
The first thing I did is to add more examples into the the original data set. The procedure is simply creating noises(randomly from 0 to 1) in k(randomly from 1 to 10) pixels for one picture. I did it 10 times for each piece of data and got a data set 10 times larger.



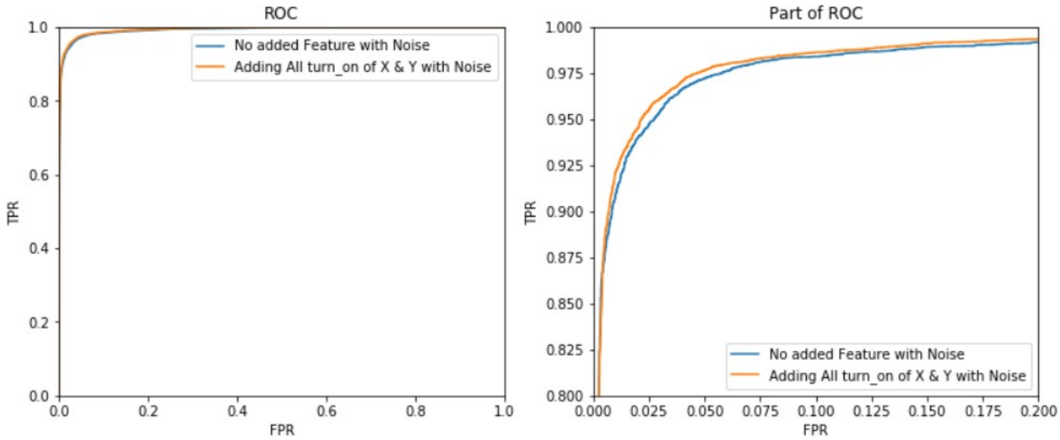


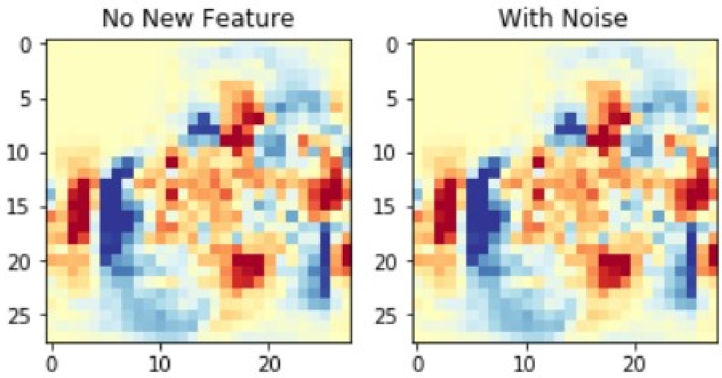
This improves the error rate of the test set from 5.3% to **<u>4%</u>**. And the AUROC is improved to **<u>0.993</u>**. The feature plot doesn't change much, not surprisingly.

Then I trained with the new feature given(ave and x^2). The improvement is not significant. The error rate drop to **0.39** and the AUROC is improved quite a bit to **<u>0.996</u>**.



The model I used is to consider a bright point with its neighbors. How many bright points has one bright neighbor horizontally, how many has 2? The same applies vertically. This 4 features helps to reduce the error rate to **3.6%**, while interestingly the AUROC keeps almost the same at **0.994**.





The feature map of weight doesn’t change so much, but the weight of added features are also comparably large.