HW2_ZheyuanHu

February 22, 2021

1 Homework 2

Name: Zheyuan Hu

NetID: zh2095

1.1 Linear Regression

1.1.1 Q1

[96]:
```
def feature_normalization(train, test):

    for i in range(train.shape[1]):
        x_max = train[:,i].max()
        x_min = train[:,i].min()
        # a is the rescale parameter, b is the shift parameter
        # a, b are trained from the training set
        # if the feature is constant, then do nothing to it
        if x_max == x_min:
            a = 1
            b = 0
        else:
            a = 1/(x_max-x_min)
            b = -x_min

        # apply the transformation on the train and test sets
        train[:,i] = a*(train[:,i]+b)
        test[:,i] = a*(test[:,i]+b)
        train_normalized = train
        test_normalized = test

    return train_normalized, test_normalized
```

1.1.2 Q2

$J(\theta) = \frac{1}{m}\|X\theta - \mathbf{y}\|_2^2$

1.1.3 Q3

$\nabla J(\theta) = \frac{2}{m}(X^\top X\theta - X^\top \mathbf{y})$

1.1.4 Q4

$\theta = \theta - \eta \nabla J(\theta)$

1.1.5 Q5

[97]:
```
def compute_square_loss(X, y, theta):

    m = X.shape[0]
    norm = np.linalg.norm(X.dot(theta) - y)
    loss = norm**2/(m)

    return loss
```

1.1.6 Q6

[98]:
```
def compute_square_loss_gradient(X, y, theta):

    m = X.shape[0]
    grad = 2*(X.T.dot(X).dot(theta) - X.T.dot(y))/m

    return grad
```

1.2 Batch gradient descent

1.2.1 Q7

[99]:
```
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):

    true_gradient = compute_square_loss_gradient(X, y, theta) #The true gradient
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features) #Initialize the gradient we approximate
    #TODO
    for i in range(num_features):
        e_i = np.zeros(num_features)
        e_i[i] = 1
        approx_grad[i] = (compute_square_loss(X, y, theta+epsilon*e_i) -
                          compute_square_loss(X, y, theta-epsilon*e_i)) /
↪(2*epsilon)
    # If the gradient is wrong, then return 0
    if np.linalg.norm(true_gradient - approx_grad) > tolerance:
        indicator = 0
    # if the gradient is corret, then return 1
    else:
```

```
        indicator = 1

    return indicator


#######################################
### Generic gradient checker
def generic_gradient_checker(X, y, theta, objective_func, gradient_func,
                             epsilon=0.01, tolerance=1e-4):
    true_gradient = gradient_func(X, y, theta)
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features)
    #TODO
    for i in range(num_features):
        e_i = np.zeros(num_features)
        e_i[i] = 1
        approx_grad[i] = (objective_func(X, y, theta+epsilon*e_i) -
                         objective_func(X, y, theta-epsilon*e_i)) / (2*epsilon)
    # If the gradient is wrong, then return 0
    if np.linalg.norm(true_gradient - approx_grad) > tolerance:
        indicator = 0
    # if the gradient is corret, then return 1
    else:
        indicator = 1

    return indicator
```

1.2.2 Q8

[100]:
```
def batch_grad_descent(X, y, alpha=0.1, num_step=1000, grad_check=False):

    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step + 1, num_features))  #Initialize theta_hist
    loss_hist = np.zeros(num_step + 1)  #Initialize loss_hist
    theta = np.zeros(num_features)  #Initialize theta
    #TODO
    loss_hist[0] = compute_square_loss(X, y, theta)
    for i in range(num_step):
        grad = compute_square_loss_gradient(X, y, theta)
        if grad_check == True:
            check = grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4)
        theta = theta - alpha*grad
        theta_hist[i+1,:] = theta
        loss = compute_square_loss(X, y, theta)
        loss_hist[i+1] = loss

    return loss_hist, theta_hist
```

1.2.3 Q9

[101]:
```
from skeleton_code import*
```

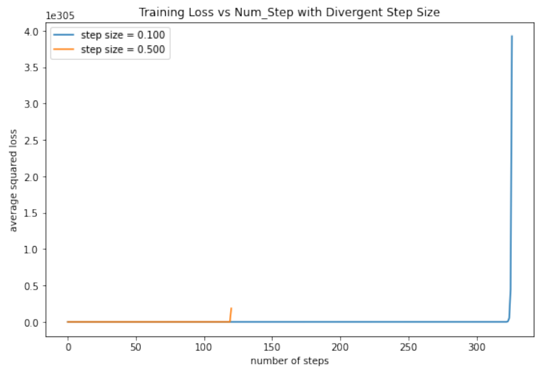
[102]:
```
X_train, y_train, X_test, y_test = load_data()
```

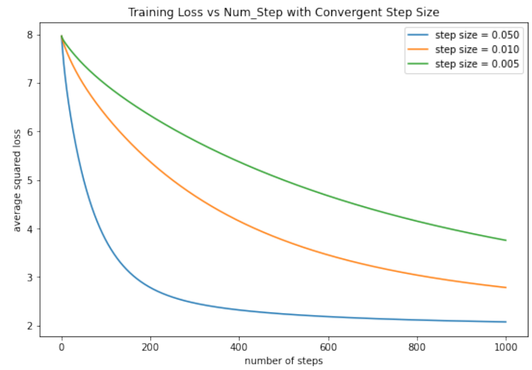
```
loading the dataset
Split into Train and Test
Scaling all to [0, 1]
```

[103]:
```
# noticing that GD diverges when alpha=0,1,0.5,, so we plot a new graph to avoid␣
↪scale conflict
fig, ax = plt.subplots(1,2,figsize=(20,6))
alpha_set = [0.1, 0.5]
num_step = 1000
step_set = np.arange(num_step+1)
for alpha in alpha_set:
    loss_hist, theta_hist = batch_grad_descent(X_train, y_train, alpha,␣
↪num_step, grad_check=False)
    ax[0].plot(step_set, loss_hist, label = 'step size = %0.3f'%alpha)
    ax[0].set_title('Training Loss vs Num_Step with Divergent Step Size')
    ax[0].set_xlabel('number of steps')
    ax[0].set_ylabel('average squared loss')
    ax[0].legend()


alpha_set = [0.05, 0.01, 0.005]
for alpha in alpha_set:
    loss_hist, theta_hist = batch_grad_descent(X_train, y_train, alpha,␣
↪num_step, grad_check=False)
    ax[1].plot(step_set, loss_hist, label = 'step size = %0.3f'%alpha)
    ax[1].set_title('Training Loss vs Num_Step with Convergent Step Size')
    ax[1].set_xlabel('number of steps')
    ax[1].set_ylabel('average squared loss')
    ax[1].legend()

plt.show()
```

```
C:\Users\52673\OneDrive\Desktop\NYU-MSDS\DS 1003 ML\1003
Homeworks\hw2\hw2\skeleton_code.py:84: RuntimeWarning: overflow encountered in
multiply
  grad = 2*(X.T.dot(X).dot(theta) - X.T.dot(y))/m
C:\Users\52673\OneDrive\Desktop\NYU-MSDS\DS 1003 ML\1003
Homeworks\hw2\hw2\skeleton_code.py:203: RuntimeWarning: invalid value
encountered in subtract
  theta = theta - alpha*grad
C:\Users\52673\OneDrive\Desktop\NYU-MSDS\DS 1003 ML\1003
Homeworks\hw2\hw2\skeleton_code.py:84: RuntimeWarning: overflow encountered in
```

```
multiply
  grad = 2*(X.T.dot(X).dot(theta) - X.T.dot(y))/m
C:\Users\52673\OneDrive\Desktop\NYU-MSDS\DS 1003 ML\1003
Homeworks\hw2\hw2\skeleton_code.py:203: RuntimeWarning: invalid value
encountered in subtract
  theta = theta - alpha*grad
```





Summarize:

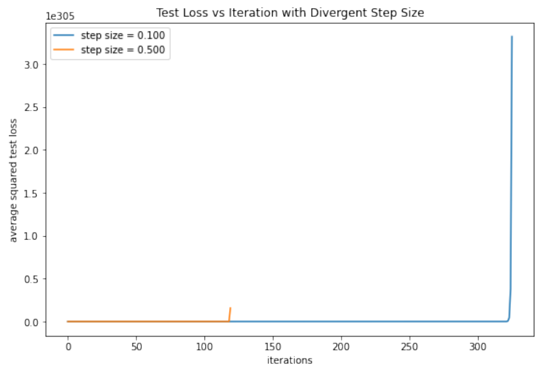
From the above plots, we can see that GD converges at step_size <= 0.1, and diverges at step_size = 0.5. For those convergent step sizes, the larger the step_size is, the faster the GD converges.

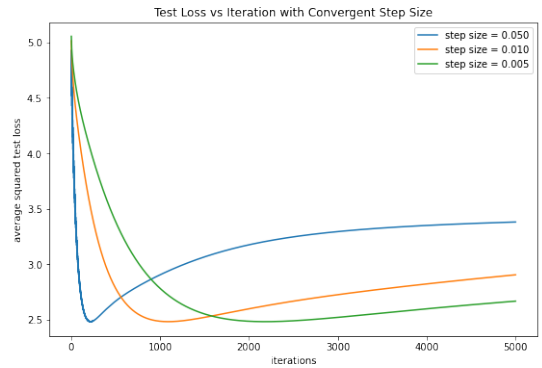
1.2.4 Q10

[106]:
```python
# noticing that GD diverges when alpha=0.5,, so we plot a new graph to avoid␣
↪scale conflict
fig, ax = plt.subplots(1,2,figsize=(20,6))
iteration = 5000
iteration_set = np.arange(iteration)
alpha_set = [0.1, 0.5]
num_features = X_train.shape[1]

for alpha in alpha_set:
    #Initialize theta
    theta = np.zeros(num_features)
    loss_test_set = []

    for i in range(iteration):
        grad = compute_square_loss_gradient(X_train, y_train, theta)
        theta = theta - alpha*grad
        loss_test = compute_square_loss(X_test, y_test, theta)
        loss_test_set.append(loss_test)
    ax[0].plot(iteration_set, loss_test_set, label = 'step size = %0.3f'%alpha)
    ax[0].set_title('Test Loss vs Iteration with Divergent Step Size')
```

```
    ax[0].set_xlabel('iterations')
    ax[0].set_ylabel('average squared test loss')
    ax[0].legend()


alpha_set = [0.05, 0.01, 0.005]
for alpha in alpha_set:
    #Initialize theta
    theta = np.zeros(num_features)
    loss_test_set = []

    for i in range(iteration):
        grad = compute_square_loss_gradient(X_train, y_train, theta)
        theta = theta - alpha*grad
        loss_test = compute_square_loss(X_test, y_test, theta)
        loss_test_set.append(loss_test)

    ax[1].plot(iteration_set, loss_test_set, label = 'step size = %0.3f'%alpha)
    ax[1].set_title('Test Loss vs Iteration with Convergent Step Size')
    ax[1].set_xlabel('iterations')
    ax[1].set_ylabel('average squared test loss')
    ax[1].legend()

plt.show()
```

```
C:\Users\52673\OneDrive\Desktop\NYU-MSDS\DS 1003 ML\1003
Homeworks\hw2\hw2\skeleton_code.py:84: RuntimeWarning: overflow encountered in
multiply
  grad = 2*(X.T.dot(X).dot(theta) - X.T.dot(y))/m
<ipython-input-106-438e472db8df>:15: RuntimeWarning: invalid value encountered
in subtract
  theta = theta - alpha*grad
C:\Users\52673\OneDrive\Desktop\NYU-MSDS\DS 1003 ML\1003
Homeworks\hw2\hw2\skeleton_code.py:84: RuntimeWarning: overflow encountered in
multiply
  grad = 2*(X.T.dot(X).dot(theta) - X.T.dot(y))/m
<ipython-input-106-438e472db8df>:15: RuntimeWarning: invalid value encountered
in subtract
  theta = theta - alpha*grad
```





For those convergent step sizes, the larger the step_size is, the faster θ overfits on the test set.

1.3 Ridge Regression

1.3.1 Q11

Gradient of $J_\lambda(\theta)$:

$\nabla J_\lambda(\theta) = \frac{2}{m}(X^\top X\theta - X^\top \mathbf{y}) + 2\lambda\theta$

The expression for updating :

$\theta = \theta - \eta \nabla J_\lambda(\theta)$

1.3.2 Q12

[107]:
```
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):

    m = X.shape[0]
    grad = 2*(X.T.dot(X).dot(theta) - X.T.dot(y))/m + 2*lambda_reg*theta

    return grad
```

1.3.3 Q13

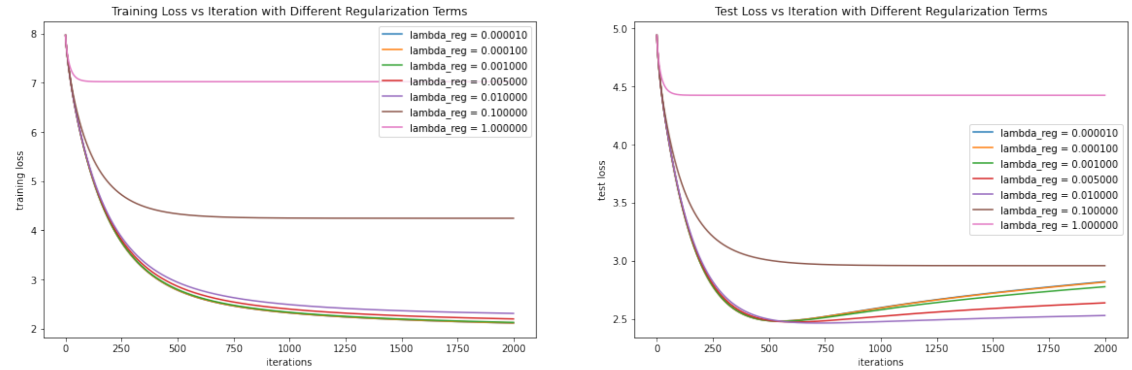
[108]:
```
def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=10**-2, num_step=1000):

    num_instances, num_features = X.shape[0], X.shape[1]
    theta = np.zeros(num_features) #Initialize theta
    theta_hist = np.zeros((num_step+1, num_features)) #Initialize theta_hist
    loss_hist = np.zeros(num_step+1) #Initialize loss_hist
    #TODO
    loss_hist[0] = compute_square_loss(X, y, theta)
    for i in range(num_step):
        grad = compute_regularized_square_loss_gradient(X, y, theta, lambda_reg)
        theta = theta - alpha*grad
```

```
        theta_hist[i+1,:] = theta
        loss = compute_square_loss(X, y, theta)
        loss_hist[i+1] = loss

    return loss_hist, theta_hist
```

1.3.4 Q14

[63]:
```
# plot training average square loss as a function of the training iterations
fig, ax = plt.subplots(1,2,figsize=(20,6))
# here choose alpha=0.02
alpha = 0.02
lamb_set = [1e-5, 1e-4, 1e-3, 5e-3, 1e-2, 1e-1, 1]
iteration = 2000
iter_set = np.arange(iteration+1)
for lambda_reg in lamb_set:
    loss_hist, theta_hist = regularized_grad_descent(X_train, y_train, alpha,␣
 ↪lambda_reg, iteration)
    ax[0].plot(iter_set, loss_hist, label = 'lambda_reg = %f'%lambda_reg)
    ax[0].set_title('Training Loss vs Iteration with Different Regularization␣
 ↪Terms')
    ax[0].set_xlabel('iterations')
    ax[0].set_ylabel('training loss')
    ax[0].legend()


# plot training average square loss as a function of the training iterations
# here choose alpha=0.02
num_features = X_train.shape[1]
iter_set = np.arange(iteration)

for lambda_reg in lamb_set:
    #Initialize theta
    theta = np.zeros(num_features)
    loss_test_set = []
    for i in range(iteration):
        grad = compute_regularized_square_loss_gradient(X_train, y_train, theta,␣
 ↪lambda_reg)
        theta = theta - alpha*grad
        loss_test = compute_square_loss(X_test, y_test, theta)
        loss_test_set.append(loss_test)

    ax[1].plot(iter_set, loss_test_set, label = 'lambda_reg = %f'%lambda_reg)
    ax[1].set_title('Test Loss vs Iteration with Different Regularization Terms')
    ax[1].set_xlabel('iterations')
    ax[1].set_ylabel('test loss')
    ax[1].legend()
```

```
plt.show()
```

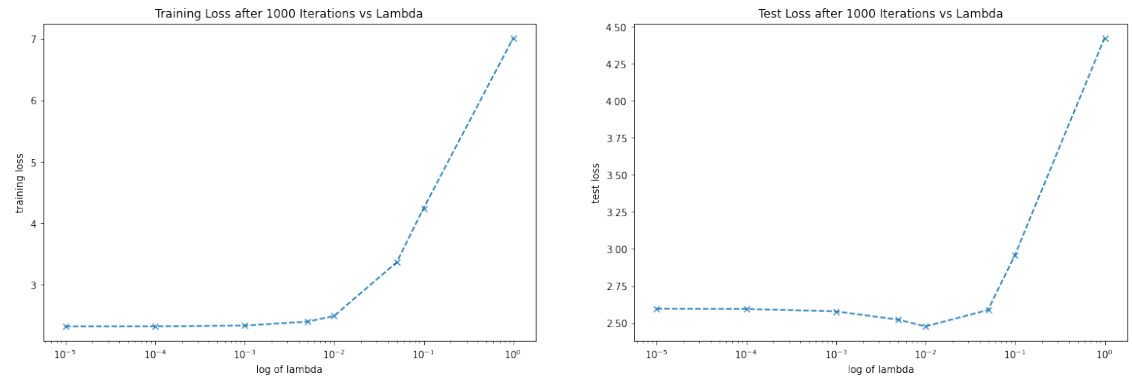


Smaller λ tend to perform better on the training set. However, the smaller the λ is, the more θ overfits on the test set.

1.3.5 Q15

[78]:
```
fig, ax = plt.subplots(1,2,figsize=(20,6))
# here choose alpha=0.02
alpha = 0.02
iteration = 1000
lamb_set = [1e-5, 1e-4, 1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 1]
loss_train_set = []
loss_test_set = []

for lambda_reg in lamb_set:
    loss_hist, theta_hist = regularized_grad_descent(X_train, y_train, alpha,␣
 ↪lambda_reg, iteration)
    loss_train = loss_hist[-1]
    theta = theta_hist[-1]
    loss_test = compute_square_loss(X_test, y_test, theta)
    loss_train_set.append(loss_train)
    loss_test_set.append(loss_test)

ax[0].plot(lamb_set, loss_train_set, 'x--')
ax[0].set_title('Training Loss after %d Iterations vs Lambda'%iteration)
ax[0].set_xscale('log')
ax[0].set_xlabel('log of lambda')
ax[0].set_ylabel('training loss')

ax[1].plot(lamb_set, loss_test_set, 'x--')
ax[1].set_title('Test Loss after %d Iterations vs Lambda'%iteration)
ax[1].set_xscale('log')
```

```
ax[1].set_xlabel('log of lambda')
ax[1].set_ylabel('test loss')

plt.show()
```

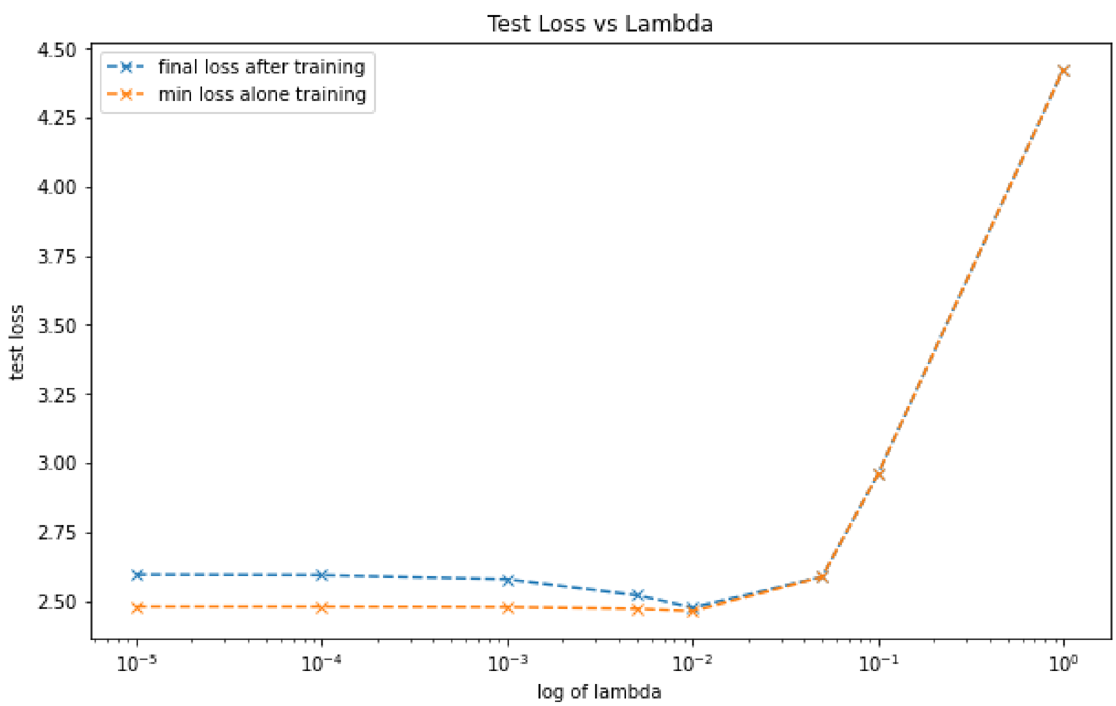


Based on the performance on the test set, I would choose $\lambda = 10^{-2}$

1.3.6 Q16

[94]:
```
fig, ax = plt.subplots(figsize=(10,6))
# here choose alpha=0.02
alpha = 0.02
iteration = 1000
lamb_set = [1e-5, 1e-4, 1e-3, 5e-3, 1e-2, 5e-2, 1e-1,1]
loss_min_set = []
loss_final_set = []
for lambda_reg in lamb_set:
    #Initialize theta
    theta = np.zeros(num_features)
    loss_test_set = []
    for i in range(iteration):
        grad = compute_regularized_square_loss_gradient(X_train, y_train, theta,␣
 ↪lambda_reg)
        theta = theta - alpha*grad
        loss_test = compute_square_loss(X_test, y_test, theta)
        loss_test_set.append(loss_test)

    loss_min = min(loss_test_set)
    loss_min_set.append(loss_min)
    loss_final = loss_test_set[-1]
    loss_final_set.append(loss_final)

ax.plot(lamb_set, loss_final_set, 'x--', label = 'final loss after training')
```

```
ax.plot(lamb_set, loss_min_set, 'x--', label = 'min loss alone training')
ax.set_title('Test Loss vs Lambda')
ax.set_xscale('log')
ax.set_xlabel('log of lambda')
ax.set_ylabel('test loss')
ax.legend()

plt.show()

best_lambda = lamb_set[np.argmin(loss_min_set)]
print('The best lambda is', best_lambda)
```




```
The best lambda is 0.01
```

Yes, I would still select $\lambda = 10^{-2}$, since its corresponding minimum loss is the smallest among all λs

1.3.7 Q17

I would select the θ that is trained with the regularization parameter $\lambda = 10^{-2}$, and is at the iteration where the test loss reaches its minimum.

Based on the previous analysis and graphs, the model with $\lambda = 10^{-2}$ has the best performance, and the reason I choose the θ corresponding to the min test loss during training rather than the θ after the entire training is to prevent the overfitting caused by too many training iterations.

1.4 Logistic regression

1.4.1 Q23

The logistic loss function is:

$$\ell_{logistic} = log(1 + e^{-m}), \text{ where } m = yh_{\theta,b}(x)$$

So the object funtion $L(\theta)$ over $\{x_i, y_i\}_{i=1}^m$ is:

$$L(\theta) = \frac{1}{m} \sum_{i=1}^{m} \log(1 + e^{y_i h_{\theta,b}(x_i)})$$

Want to show that

$$\log(1 + e^{y_i h_{\theta,b}(x_i)}) = \frac{1}{2}(1 + y_i)\log(1 + e^{-h_{\theta,b}(x_i)}) + (1 - y_i)\log(1 + e^{h_{\theta,b}(x_i)})$$

Here y_i is either 1 or -1, for $y_i = 1$,

$$RHS = LHS = \log(1 + e^{h_{\theta,b}(x_i)})$$

similarly, for $y_i = -1$,

$$RHS = LHS = \log(1 + e^{-h_{\theta,b}(x_i)})$$

so the equation holds for both $y_i = 1, -1$

Therefore,

$$L(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (1 + y_i)\log(1 + e^{-h_{\theta,b}(x_i)}) + (1 - y)\log(1 + e^{h_{\theta,b}(x_i)})$$

1.4.2 Q24

The loss function with ℓ_1 regularization term α is:

$$L(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (1 + y_i)\log(1 + e^{-h_{\theta,b}(x_i)}) + (1 - y)\log(1 + e^{h_{\theta,b}(x_i)}) + \alpha\|\theta\|_1$$

where $\|\theta\|_1 = |\theta_1| + |\theta_2| + \cdots + |\theta_n|$ is the ℓ_1 norm

1.4.3 Q25

[109]:
```
def classification_error(clf, X, y):
    ## TODO
    n = X.shape[0]
    y_pred = clf.predict(X)
    err = sum(y_pred != y)/n

    return err
```

```python
X_train, X_test, y_train, y_test = pre_process_mnist_01()

clf = SGDClassifier(loss='log', max_iter=1000,
                    tol=1e-3,
                    penalty='l1', alpha=0.01,
                    learning_rate='invscaling',
                    power_t=0.5,
                    eta0=0.01,
                    verbose=1)
clf.fit(X_train, y_train)

test = classification_error(clf, X_test, y_test)
train = classification_error(clf, X_train, y_train)
print('train: ', train, end='\t')
print('test: ', test)
```

```
-- Epoch 1
Norm: 0.74, NNZs: 313, Bias: -0.007640, T: 9902, Avg. loss: 0.044237
Total training time: 0.03 seconds.
-- Epoch 2
Norm: 0.83, NNZs: 272, Bias: -0.008024, T: 19804, Avg. loss: 0.032060
Total training time: 0.06 seconds.
-- Epoch 3
Norm: 0.88, NNZs: 253, Bias: -0.007908, T: 29706, Avg. loss: 0.030068
Total training time: 0.09 seconds.
-- Epoch 4
Norm: 0.93, NNZs: 242, Bias: -0.007640, T: 39608, Avg. loss: 0.029219
Total training time: 0.12 seconds.
-- Epoch 5
Norm: 0.96, NNZs: 237, Bias: -0.007293, T: 49510, Avg. loss: 0.028547
Total training time: 0.15 seconds.
-- Epoch 6
Norm: 1.00, NNZs: 230, Bias: -0.006902, T: 59412, Avg. loss: 0.028162
Total training time: 0.18 seconds.
-- Epoch 7
Norm: 1.02, NNZs: 225, Bias: -0.006492, T: 69314, Avg. loss: 0.027843
Total training time: 0.21 seconds.
-- Epoch 8
Norm: 1.05, NNZs: 217, Bias: -0.006060, T: 79216, Avg. loss: 0.027601
Total training time: 0.25 seconds.
Convergence after 8 epochs took 0.25 seconds
train:  0.00222177337911533     test:  0.0012300123001230013
```

[112]:
```python
# check if the function returns the same value as 1 - clf.score(X, y)
train = 1 - clf.score(X_train, y_train)
test = 1 - clf.score(X_test, y_test)
```

```
print('train: ', train, end='\t')
print('test: ', test)
```

```
train:  0.0022217733791153327   test:  0.0012300123001229846
```

1.4.4 Q26

[1]:
```
from mnist_classification_source_code import*
```

[2]:
```
X_train, X_test, y_train, y_test = pre_process_mnist_01()

N_train = 100
X_train, y_train = sub_sample(N_train, X_train, y_train)
```

[8]:
```
alpha_set = [1e-4, 2e-4, 5e-4, 1e-3, 2e-3, 5e-3, 1e-2, 2e-2, 5e-2, 1e-1]
err_mean_set = []
err_std_set = []
for alpha in alpha_set:
    err_set = []
    for i in range(10):
        clf = SGDClassifier(loss='log', max_iter=1000,
                    tol=1e-3,
                    penalty='l1', alpha=alpha,
                    learning_rate='invscaling',
                    power_t=0.5,
                    eta0=0.01,
                    verbose=1)

        clf.fit(X_train, y_train)
        err = classification_error(clf, X_test, y_test)
        err_set.append(err)

    err_mean = np.mean(err_set)
    err_std = np.std(err_set)
    err_mean_set.append(err_mean)
    err_std_set.append(err_std)
```

```
-- Epoch 1
Norm: 0.30, NNZs: 607, Bias: 0.006820, T: 100, Avg. loss: 0.156853
Total training time: 0.00 seconds.
-- Epoch 2
Norm: 0.34, NNZs: 607, Bias: 0.007928, T: 200, Avg. loss: 0.074943
Total training time: 0.00 seconds.
-- Epoch 3
Norm: 0.36, NNZs: 607, Bias: 0.008634, T: 300, Avg. loss: 0.060653
Total training time: 0.00 seconds.
-- Epoch 4
```

```
Norm: 0.57, NNZs: 137, Bias: 0.037168, T: 1000, Avg. loss: 0.185514
Total training time: 0.01 seconds.
-- Epoch 11
Norm: 0.58, NNZs: 134, Bias: 0.038187, T: 1100, Avg. loss: 0.184512
Total training time: 0.01 seconds.
-- Epoch 12
Norm: 0.60, NNZs: 133, Bias: 0.039174, T: 1200, Avg. loss: 0.183257
Total training time: 0.01 seconds.
-- Epoch 13
Norm: 0.61, NNZs: 128, Bias: 0.040107, T: 1300, Avg. loss: 0.182418
Total training time: 0.01 seconds.
-- Epoch 14
Norm: 0.62, NNZs: 124, Bias: 0.040990, T: 1400, Avg. loss: 0.181443
Total training time: 0.01 seconds.
-- Epoch 15
Norm: 0.63, NNZs: 118, Bias: 0.041851, T: 1500, Avg. loss: 0.180896
Total training time: 0.01 seconds.
-- Epoch 16
Norm: 0.64, NNZs: 116, Bias: 0.042688, T: 1600, Avg. loss: 0.179650
Total training time: 0.01 seconds.
-- Epoch 17
Norm: 0.65, NNZs: 116, Bias: 0.043492, T: 1700, Avg. loss: 0.178863
Total training time: 0.01 seconds.
-- Epoch 18
Norm: 0.66, NNZs: 113, Bias: 0.044270, T: 1800, Avg. loss: 0.177992
Total training time: 0.01 seconds.
-- Epoch 19
Norm: 0.67, NNZs: 110, Bias: 0.045025, T: 1900, Avg. loss: 0.177324
Total training time: 0.01 seconds.
-- Epoch 20
Norm: 0.67, NNZs: 108, Bias: 0.045758, T: 2000, Avg. loss: 0.176674
Total training time: 0.02 seconds.
-- Epoch 21
Norm: 0.68, NNZs: 107, Bias: 0.046475, T: 2100, Avg. loss: 0.175907
Total training time: 0.02 seconds.
Convergence after 21 epochs took 0.02 seconds
```

[9]:

```
fig, ax = plt.subplots(figsize=(10,6))
ax.errorbar(alpha_set, err_mean_set, err_std_set, marker='s', mfc='red')
ax.set_title('Test Classification Error vs Alpha')
ax.set_xscale('log')
ax.set_xlabel('log of alpha')
ax.set_ylabel('test classification error')

plt.show()
```

1.4.5 Q27

The randomness comes from the SGD algorithm, where in each iteration, we use only 1 point (x_i, y_i) to determine the step direction, and this point is randomly selected.

1.4.6 Q28

[15]:
```
best_alpha = alpha_set[np.argmin(err_mean_set)]

print('The optimal  among the values I tested is:', best_alpha )
```

```
The optimal  among the values I tested is: 0.1
```

1.4.7 Q29

[60]:
```
alpha_set = [1e-4, 2e-4, 5e-4, 1e-3, 2e-3, 5e-3, 1e-2, 2e-2, 5e-2, 1e-1]
theta_set = np.zeros([len(alpha_set),28,28])
scale_set = []
for i in range(len(alpha_set)):
    clf = SGDClassifier(loss='log', max_iter=1000,
                        tol=1e-3,
                        penalty='l1', alpha=alpha_set[i],
                        learning_rate='invscaling',
                        power_t=0.5,
                        eta0=0.01,
```

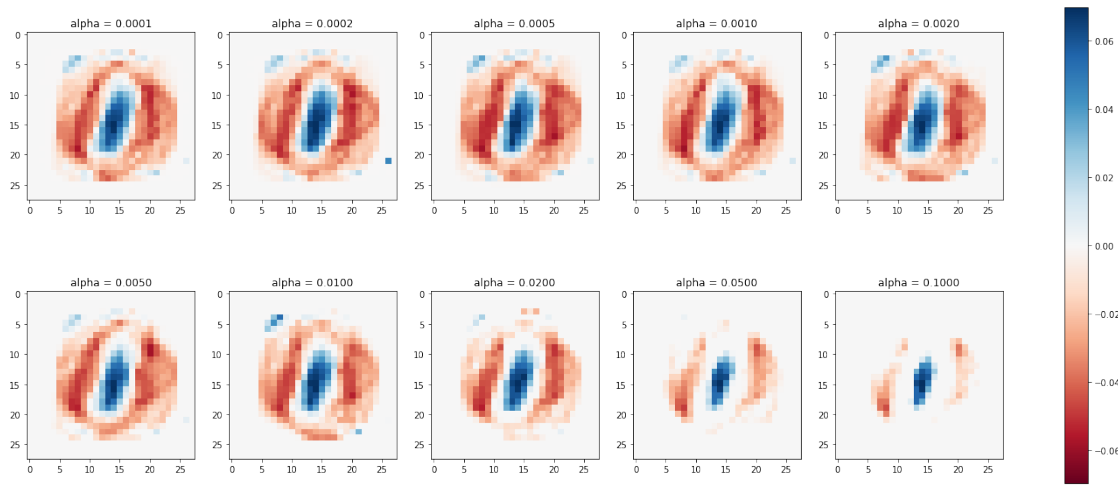
```
                    verbose=1)

    clf.fit(X_train, y_train)
    theta = clf.coef_.reshape(28,28)
    theta_set[i] = theta
    scale = np.abs(clf.coef_).max()
    scale_set.append(scale)
```

```
-- Epoch 1
Norm: 0.30, NNZs: 472, Bias: 0.000684, T: 100, Avg. loss: 0.165201
Total training time: 0.00 seconds.
-- Epoch 2
Norm: 0.34, NNZs: 524, Bias: 0.002083, T: 200, Avg. loss: 0.072415
Total training time: 0.00 seconds.
-- Epoch 3
Norm: 0.36, NNZs: 547, Bias: 0.003033, T: 300, Avg. loss: 0.057848
Total training time: 0.00 seconds.
-- Epoch 4
Norm: 0.38, NNZs: 549, Bias: 0.003678, T: 400, Avg. loss: 0.050057
Total training time: 0.00 seconds.
-- Epoch 5
Norm: 0.39, NNZs: 550, Bias: 0.004147, T: 500, Avg. loss: 0.045036
Total training time: 0.00 seconds.
-- Epoch 6
Norm: 0.40, NNZs: 551, Bias: 0.004540, T: 600, Avg. loss: 0.041395
Total training time: 0.00 seconds.
-- Epoch 7
Norm: 0.41, NNZs: 551, Bias: 0.004879, T: 700, Avg. loss: 0.038595
Total training time: 0.00 seconds.
-- Epoch 8
Norm: 0.42, NNZs: 551, Bias: 0.005172, T: 800, Avg. loss: 0.036348
Total training time: 0.00 seconds.
-- Epoch 9
Norm: 0.43, NNZs: 550, Bias: 0.005432, T: 900, Avg. loss: 0.034494
Total training time: 0.00 seconds.
-- Epoch 10
Norm: 0.44, NNZs: 549, Bias: 0.005658, T: 1000, Avg. loss: 0.032925
Total training time: 0.01 seconds.
-- Epoch 11
Norm: 0.44, NNZs: 547, Bias: 0.005870, T: 1100, Avg. loss: 0.031564
Total training time: 0.01 seconds.
-- Epoch 12
Norm: 0.45, NNZs: 545, Bias: 0.006058, T: 1200, Avg. loss: 0.030376
Total training time: 0.01 seconds.
-- Epoch 13
Norm: 0.45, NNZs: 545, Bias: 0.006231, T: 1300, Avg. loss: 0.029324
Total training time: 0.01 seconds.
-- Epoch 14
```

```
Norm: 0.54, NNZs: 140, Bias: 0.034731, T: 800, Avg. loss: 0.185452
Total training time: 0.01 seconds.
-- Epoch 9
Norm: 0.55, NNZs: 137, Bias: 0.035923, T: 900, Avg. loss: 0.184295
Total training time: 0.01 seconds.
-- Epoch 10
Norm: 0.57, NNZs: 133, Bias: 0.037056, T: 1000, Avg. loss: 0.182209
Total training time: 0.01 seconds.
-- Epoch 11
Norm: 0.58, NNZs: 132, Bias: 0.038123, T: 1100, Avg. loss: 0.181747
Total training time: 0.01 seconds.
-- Epoch 12
Norm: 0.60, NNZs: 130, Bias: 0.039145, T: 1200, Avg. loss: 0.180725
Total training time: 0.01 seconds.
-- Epoch 13
Norm: 0.61, NNZs: 125, Bias: 0.040110, T: 1300, Avg. loss: 0.180367
Total training time: 0.01 seconds.
-- Epoch 14
Norm: 0.62, NNZs: 124, Bias: 0.041043, T: 1400, Avg. loss: 0.179097
Total training time: 0.01 seconds.
-- Epoch 15
Norm: 0.63, NNZs: 122, Bias: 0.041937, T: 1500, Avg. loss: 0.178602
Total training time: 0.01 seconds.
-- Epoch 16
Norm: 0.64, NNZs: 120, Bias: 0.042798, T: 1600, Avg. loss: 0.177770
Total training time: 0.01 seconds.
-- Epoch 17
Norm: 0.65, NNZs: 118, Bias: 0.043627, T: 1700, Avg. loss: 0.177366
Total training time: 0.01 seconds.
-- Epoch 18
Norm: 0.66, NNZs: 117, Bias: 0.044429, T: 1800, Avg. loss: 0.176582
Total training time: 0.01 seconds.
-- Epoch 19
Norm: 0.67, NNZs: 112, Bias: 0.045209, T: 1900, Avg. loss: 0.175977
Total training time: 0.01 seconds.
Convergence after 19 epochs took 0.01 seconds
```

[87]:
```python
alpha_set = [1e-4, 2e-4, 5e-4, 1e-3, 2e-3, 5e-3, 1e-2, 2e-2, 5e-2, 1e-1]
fig, ax = plt.subplots(2,5,figsize=(25,10))
k = 0
for i in range(2):
    for j in range(5):
        scale = scale_set[k]
        theta = theta_set[k]
        im = ax[i,j].imshow(theta, cmap=plt.cm.RdBu, vmax=scale, vmin=-scale)
        ax[i,j].set_title('alpha = %0.4f'%alpha_set[k])
        k += 1
```

```
fig.colorbar(im, ax=ax.ravel().tolist())
plt.show()
```



1.4.8 Q30

Note about θ:

In the visualization graph of the fitted θ, the shape of the blue part representing positive coefficients looks like the number "1", while the red part representing negative coefficients looks like "0", and the position of the blue "1" and red "0" is roughly consistent with the picture of handwritten numbers. That is, for example, if x_i is the vector of handwritten "1", then its non-zero values are similar placed as the positive coefficients in θ, then by multiplying x_i with θ, the value of the preiction gets larger, thus helps predict 1.

Note the effect of the regularization:

We can see that as α goes up, those relatively small coefficients in θ is decresing and tend to approach 0. This shows that the regularization helps identify the important features, and as we penalize more on the complexity, more unimportant features would be excluded and only the important features left to be considered in the prediction model.