Stats 202A - Final Project

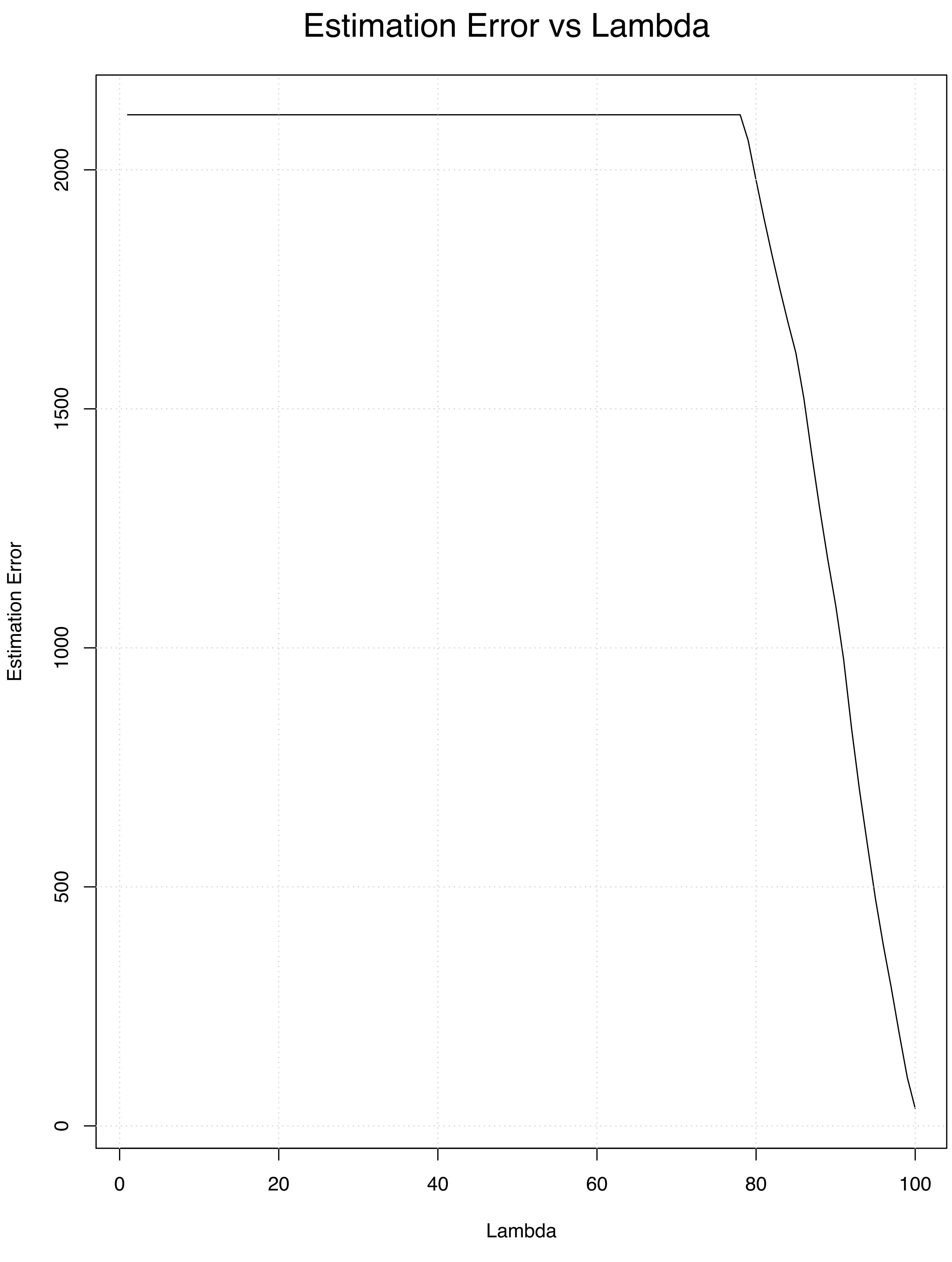
Peter Racioppo

Part 1:
Lasso & Epsilon Boosting in R

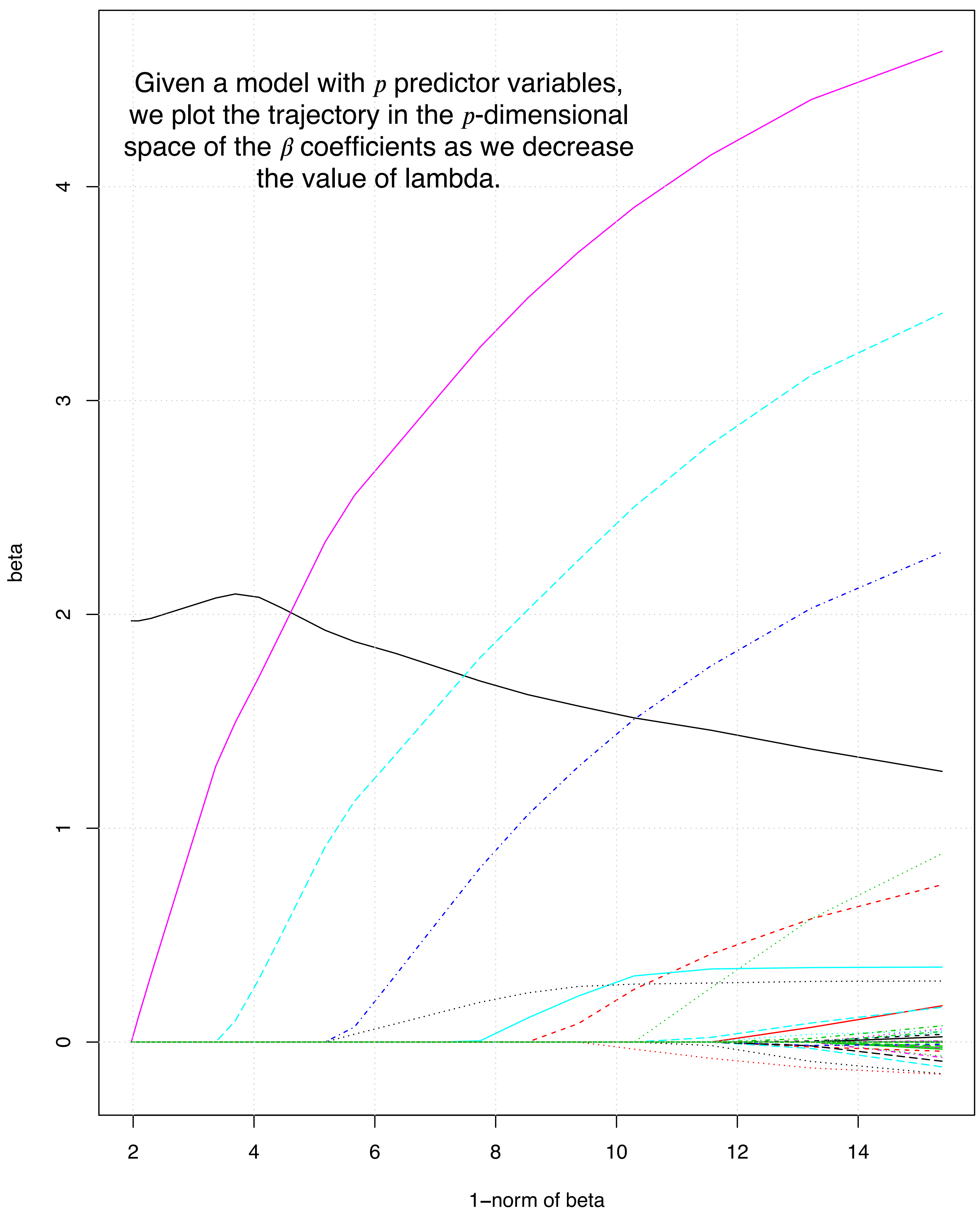
In [ ]:
```
#############################################################
## Stat 202A - Homework 7
## Author: Peter Racioppo
## Date: 12/14/2019
## Description: This script implements the lasso
#############################################################

# 1) Write R code using the included script 'Lasso.R' for computing
# the Lasso solution path using coordinate descent. Please include
# but do not penalize the intercept term (as we did for ridge regression).

# 2) Use epsilon-boosting technique. Compare the difference.

# 2) For Lasso, plot the estimation error over the different values of lambda.

#############################################################
## INSTRUCTIONS: Please fill in the missing lines of code
## only where specified. Do not change function names,
## function inputs or outputs. You can add examples at the
## end of the script (in the "Optional examples" section) to
## double-check your work, but MAKE SURE TO COMMENT OUT ALL
## OF YOUR EXAMPLES BEFORE SUBMITTING.
##
## Very important: Do not use the function "setwd" anywhere
## in your code. If you do, I will be unable to grade your
## work since R will attempt to change my working directory
## to one that does not exist.
#############################################################

#####################################
## Function 1: Lasso Solution Path ##
#####################################

myLasso <- function(X, Y, lambda_all){

  # Find the lasso solution path for various values of
  # the regularization parameter lambda.
  #
  # X: n x p matrix of explanatory variables.
  # Y: n dimensional response vector
  # lambda_all: Vector of regularization parameters. Make sure
  # to sort lambda_all in decreasing order for efficiency.
```

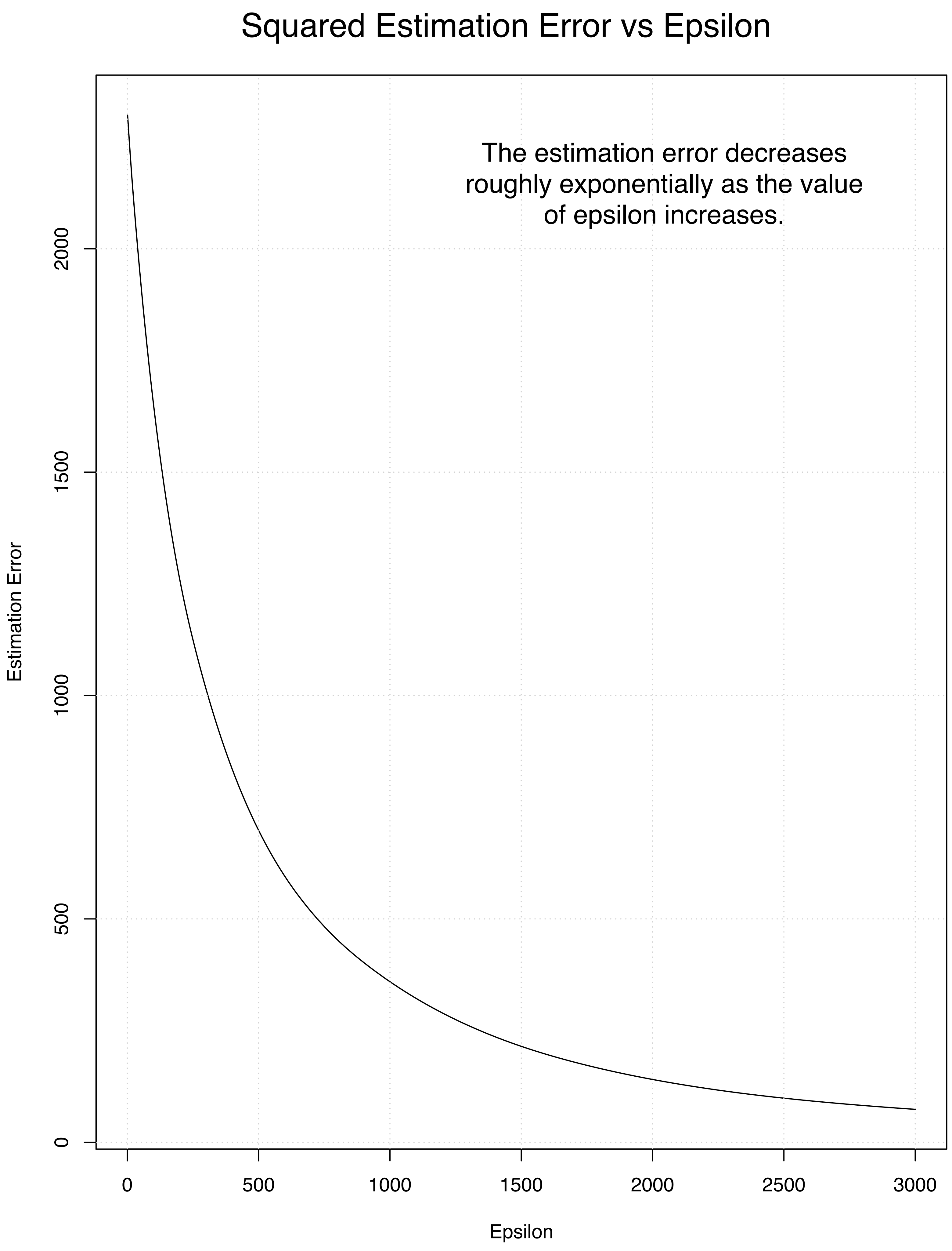
```
#
# Returns a matrix containing the lasso solution vector
# beta for each regularization parameter.

#######################
## FILL IN CODE HERE ##
#######################

n = nrow(X)
X = cbind(rep(1, n), X) # Add a column of ones
p = ncol(X)
S = 10
len = length(lambda_all)
beta_all = matrix(rep(0,len*p), nrow = p) # Initialize beta_all

# Rj = Y - Sum(k~=j)[(Xk*beta_k)]
# beta_j_hat = dot(Rj,Xj)/Xj^2
# beta_j = sign(beta_j_hat) * max(0, abs(beta_j_hat)-lambda/Xj^2)

R = Y # Initialize R
beta = rep(0,p) # Initialize beta

# X^2
SS = rep(0, p)
for (j in 1:p)
  SS[j] = sum(X[,j]^2)

for (l in 1:len) {
  lambda = lambda_all[l]
  for (t in 1:S)
    db = sum(R * X[,1]) / SS[1] # beta_j_hat ??
    b = beta[1] + db # Add beta_j_hat to beta ??
    # Set beta_j = sign(beta_j_hat)*max(0,abs(beta_j_hat)): ??
    b = sign(b) * max(0, abs(b))
    db = b - beta[1] # dbeta_k = new beta - old beta
    # dR_i = - X_i * dbeta_k
    R = R - X[,1] * db
    beta[1] = b # Update beta to hold old value
    for (k in 2:p) {
      # R = R + X[,k] * beta[k]
      # db = sum(R * X[,k]);
      # beta[k] = sign(db) * max(0, (abs(db)-lambda)/SS[k])
      # R = R - X[,k] * beta[k]
```

```
        db = sum(R * X[,k]) / SS[k]
        b = beta[k] + db
        # Beta_hat_lambda. Soft thresholding:
        b = sign(b) * max(0, abs(b)-lambda/SS[k])
        db = b - beta[k]
        R = R - X[,k] * db # Update R
        beta[k] = b
      }
    beta_all[,l] = beta
  }

  ## Function should output the matrix beta_all, the
  ## solution to the lasso regression problem for all
  ## the regularization parameters.
  ## beta_all is (p+1) x length(lambda_all)
  return(beta_all)

}

###################################
## Function 2: Epsilon Boosting ##
###################################

EpsilonBoosting <- function(X, Y){

  # The stagewise regression iterates the following steps.
  # Given the current R = Y - sum(pj=1 Xj*betaj), find j with the maximal <R, Xj>.
  # Then update beta_j <- beta_j + eps*<R, Xj> for a small eps. This is similar to
  # matching pursuit but is much less greedy. Such an update will change R and
  # reduce <R, Xj>, until another Xj catches up. So overall, the algorithm ensures
  # that all of the selected Xj have the same <R, Xj>, which is the case with the
  # algorithm in the above two sections. The stagewise regression is also called
  # epsilon-boosting.
  #######################

  n = nrow(X) # Num samples
  X = cbind(rep(1, n), X) # Add a column of ones
  p = ncol(X) # Num features
  T = 3000 # Num steps
  epsilon = 0.0001 # Epsilon (step size)

  beta = matrix(rep(0, p), nrow = p) # Initialize beta
  # db = matrix(rep(0, p), nrow = p)
  beta_all = matrix(rep(0, p*T), nrow = p) # Beta history vector

  for (t in 1:T)
  {
    P = X*0 # Initialize P
    for (j in 1:p){
      P[,j] = X[,j]*beta[j] # P = Xj*betaj
    }
    R = Y - rowSums(P) # R = Y - sum(Xj*betaj)
    RX = rep(0, p) # Initialize RX
    for (j in 2:p){
      RX[j] = sum(R * X[,j]) # RX[j] = <R,Xj>
    }
    db = max(RX) # Max over j of <R,Xj>
    j_max = which.max(RX) # j that maximizes <R,Xj>
    beta[j_max] = beta[j_max] + epsilon*db # beta_j <- beta_j + eps*<R, Xj>
    beta_all[,t] = beta # Update history vector for t_th step
  }
  return(beta_all)
}

test <- function() {
  # Generate data:
  set.seed(10086)
  n = 50 # Num samples
  p = 200 # Num features
  lambda_all = (100:1)*10 # Lambda

  X = matrix(rnorm(n*p), nrow = n) # Random data
  beta_true = matrix(rep(0,p), nrow = p) # True values of beta
  beta_true[1:5] = 1:5 # True values of beta
  Y = 1 + X %*% beta_true + rnorm(n) # Compute Y
  beta_all <- myLasso(X, Y, lambda_all) # Estimate beta

  X = cbind(rep(1, n), X); # Add a column of ones
  Y_hat = X %*% beta_all; # Estimate Y
  estimation_error = rep(0,100) # Initialize estimation error
  for (i in 1:100)
    estimation_error[i] = sum((Y-Y_hat[,i])^2) # Squared estimation error
  matplot(t(matrix(rep(1,p+1),nrow=1)%*%abs(beta_all)), t(beta_all), type = 'l',xlab="1-norm of beta",y
  grid()

  matplot(estimation_error,type = 'l',xlab="Lambda",ylab="Estimation Error")
  grid()
}

test2 <- function() {
  # Generate data:
  set.seed(10086)
  n = 50 # Num samples
  p = 200 # Num features
  X = matrix(rnorm(n*p), nrow = n) # Random data
  beta_true = matrix(rep(0,p), nrow = p) # True values of beta
  beta_true[1:5] = 1:5 # True values of beta
  Y = 1 + X %*% beta_true + rnorm(n) # Compute Y
  beta_all <- EpsilonBoosting(X, Y) # Estimate beta

  X = cbind(rep(1, n), X); # Add a column of ones
  Y_hat = X %*% beta_all; # Estimate Y
  estimation_error = rep(0,3000) # Initialize estimation error
  for (i in 1:3000)
    estimation_error[i] = sum((Y-Y_hat[,i])^2) # Squared estimation error
  matplot(t(matrix(rep(1,p+1),nrow=1)%*%abs(beta_all)), t(beta_all), type = 'l',xlab="1-norm of beta",y
  grid()

  matplot(estimation_error,type = 'l',xlab="Epsilon",ylab="Estimation Error")
  grid()
}

test()
test2()
```

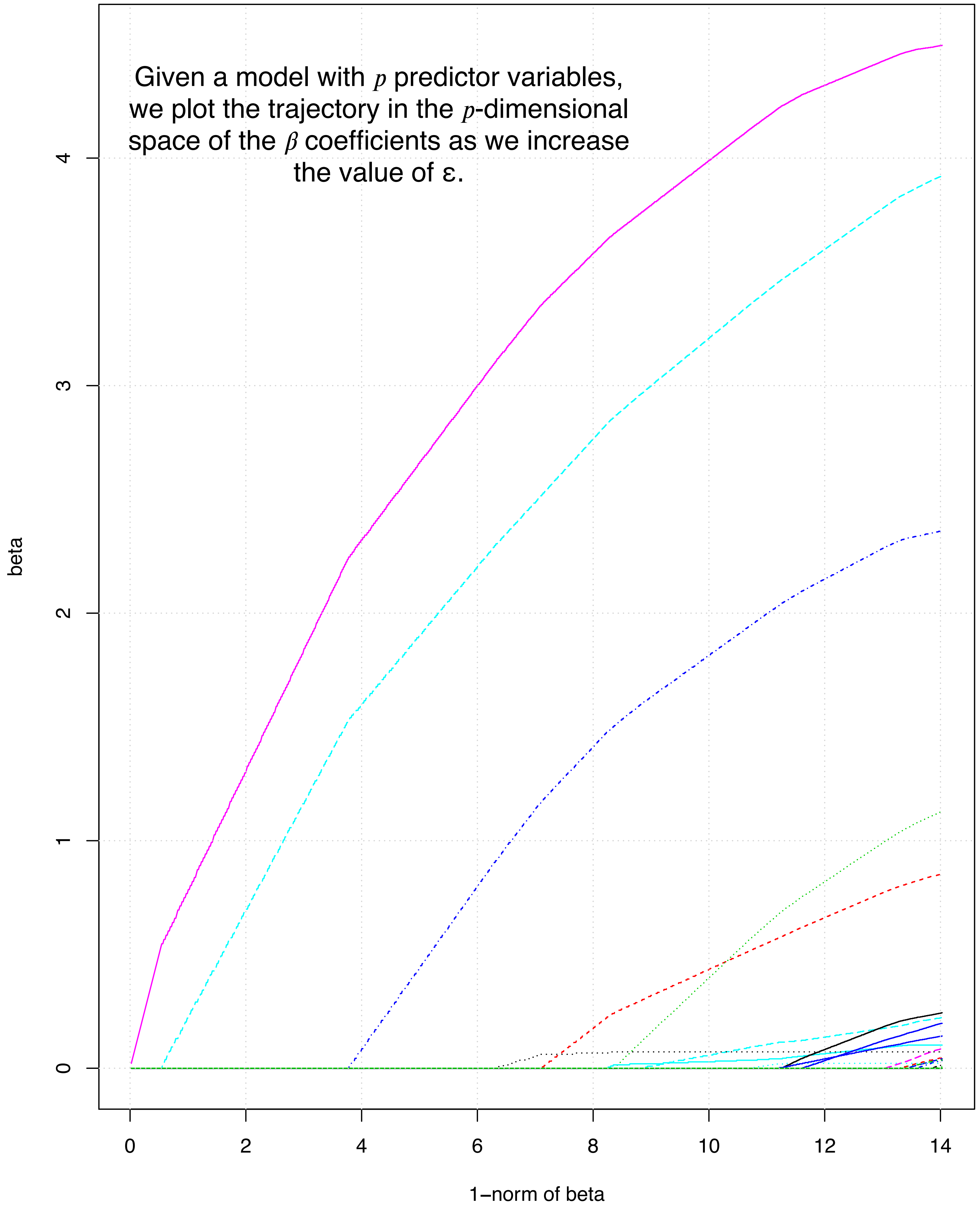

Estimation Error vs Lambda
Estimation Error
Lambda
0
500
1000
1500
2000
0
20
40
60
80
100

Solution Paths for LASSO




Squared Estimation Error vs Epsilon
The estimation error decreases roughly exponentially as the value of epsilon increases.
Estimation Error
0
500
1000
1500
2000
Epsilon
0
500
1000
1500
2000
2500
3000

Solution Paths for Epsilon-Boosting



Given a model with p predictor variables, we plot the trajectory in the p-dimensional space of the β coefficients as we increase the value of ε.

Part 2:
Neural Networks in Python

Fully-Connected Neural Nets

In this exercise we will implement fully-connected networks on MNIST datasets. For each layer we will implement a `forward` and a `backward` function. The `forward` function will receive inputs, weights, and other parameters and will return both an output and a `cache` object storing data needed for the backward pass, like this:

```
def layer_forward(x, w):
  """ Receive inputs x and weights w """
  # Do some computations ...
  z = # ... some intermediate value
  # Do some more computations ...
  out = # the output

  cache = (x, w, z, out) # Values we need to compute gradients

  return out, cache
```

The backward pass will receive upstream derivatives and the `cache` object, and will return gradients with respect to the inputs and weights, like this:

```
def layer_backward(dout, cache):
  """
  Receive derivative of loss with respect to outputs and cache,
  and compute derivative with respect to inputs.
  """
  # Unpack cache values
  x, w, z, out = cache

  # Use values in cache to compute derivatives
  dx = # Derivative of loss with respect to x
  dw = # Derivative of loss with respect to w

  return dx, dw
```

After implementing a bunch of layers this way, we will be able to easily combine them to build classifiers with different architectures.

In addition to implementing fully-connected networks of arbitrary depth, we will also explore different update rules for optimization, and introduce

In [3]:
```
# As usual, a bit of setup
from __future__ import print_function
import time
import numpy as np
import matplotlib.pyplot as plt
from stats202a.classifiers.fc_net import *
from stats202a.data_utils import get_mnist_data
from stats202a.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from stats202a.solver import Solver
from stats202a.layers import *

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

Download data

you need to download the MNIST datasets. Run the following bash in the `stats202a/datasets` directory: `./get_datasets.sh` (for windows, run `./get_datasets.cmd`)

In [62]:
```
# Your code doesn't work, as usual.

# Load the (preprocessed) MNIST data.
# The second dimension of images indicated the number of channel. For black and white images in MNIS
T, channel=1.
# data = get_mnist_data()
# # ./get_datasets.sh
# for k, v in list(data.items()):
#   print(('%s: ' % k, v.shape))

import tensorflow as tf
# data = tf.keras.datasets.mnist.load_data()

mnist = tf.keras.datasets.mnist
(X_train, y_train), (X_test, y_test) = mnist.load_data()
data = {
  "X_train": X_train,
  "y_train": y_train,
  "X_val": X_test,
  "y_val": y_test}

for k, v in list(data.items()):
  print(('%s: ' % k, v.shape))
```

```
('X_train: ', (60000, 28, 28))
('y_train: ', (60000,))
('X_val: ', (10000, 28, 28))
('y_val: ', (10000,))
```

Fully-connected layer: foward

Open the file `stats202a/layers.py` and implement the `fc_forward` function.

Once you are done you can test your implementaion by running the following:

In [5]:
```
# Test the fc_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape), output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = fc_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around 1e-9.
print('Testing fc_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing fc_forward function:
difference:  9.769847728806635e-10
```

```
In [6]: # Test the fc_backward function
        np.random.seed(231)
        x = np.random.randn(10, 2, 3)
        w = np.random.randn(6, 5)
        b = np.random.randn(5)
        dout = np.random.randn(10, 5)

        dx_num = eval_numerical_gradient_array(lambda x: fc_forward(x, w, b)[0], x, dout)
        dw_num = eval_numerical_gradient_array(lambda w: fc_forward(x, w, b)[0], w, dout)
        db_num = eval_numerical_gradient_array(lambda b: fc_forward(x, w, b)[0], b, dout)

        _, cache = fc_forward(x, w, b)
        dx, dw, db = fc_backward(dout, cache)

        # The error should be around 1e-10
        print('Testing fc_backward function:')
        print('dx error: ', rel_error(dx_num, dx))
        print('dw error: ', rel_error(dw_num, dw))
        print('db error: ', rel_error(db_num, db))
```

```
Testing fc_backward function:
dx error:  5.399100368651805e-11
dw error:  9.904211865398145e-11
db error:  2.4122867568119087e-11
```

Fully-connected layer: backward

Now implement the `fc_backward` function and test your implementation using numeric gradient checking.

ReLU layer: forward

Implement the forward pass for the ReLU activation function in the `relu_forward` function and test your implementation using the following:

In [11]:
```
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,         ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
                        [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

# Compare your output with ours. The error should be around 5e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

ReLU layer: backward

Now implement the backward pass for the ReLU activation function in the `relu_backward` function and test your implementation using numeric gradient checking:

In [12]:
```
np.random.seed(231)
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be around 3e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.2756349136310288e-12
```

"Sandwich" layers

There are some common patterns of layers that are frequently used in neural nets. For example, fc/conv layers are frequently followed by a ReLU nonlinearity. To make these common patterns easy, we define several convenience layers in the file `stats202a/layer_utils.py` .

Implement the `fc_relu_forward` and `fc_relu_backward` functions, and run the following to numerically gradient check the backward pass:

In [14]:
```
from stats202a.layer_utils import fc_relu_forward, fc_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = fc_relu_forward(x, w, b)
dx, dw, db = fc_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: fc_relu_forward(x, w, b)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: fc_relu_forward(x, w, b)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: fc_relu_forward(x, w, b)[0], b, dout)

print('Testing affine_relu_forward:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward:
dx error:  6.750562121603446e-11
dw error:  8.162015570444288e-11
db error:  7.826724021458994e-12
```

Loss layers: Softmax

Now implement the softmax loss in the `softmax_loss` function.

The softmax loss is in the following form: $L_i = -\log(exp(x_{iy_i})/\sum_j(exp(x_{ij})))$x_i$istheoutputofthetopfclayerforinputimage$i,y_i $isthetruelabelof$x_i, $and$j$ is the index of category. To avoid overflow, you may follow the trick in [https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/ (https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/)](https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/) to compute the 'logsumexp' operation.

You can make sure that the implementations are correct by running the following:

In [15]:
```
np.random.seed(231)
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x, verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be 2.3 and dx error should be 1e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing softmax_loss:
loss:  2.3025458445007376
dx error:  8.234144091578429e-09
```

Two-layer network

First we implement a two-layer network with only one hidden layer. We will use the class TwoLayerNet in the file `stats202a/classifiers/fc_net.py` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays.

Besides softmax loss, we add another L2 regularization loss: $||W||_2^2$, where W is the weights of all layers. Bias are not included. We use a parameter `self.reg` to control the strength of regularization.

Open the file `stats202a/classifiers/fc_net.py` and complete the implementation of the `TwoLayerNet` class. This class will serve as a model for the other networks you will implement in this assignment, so read through it to make sure you understand the API. You can run the cell below to test your implementation.

In [19]:
```
np.random.seed(231)
N, D, H, C = 3, 5, 50, 7
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)

std = 1e-3
model = TwoLayerNet(input_dim=D, hidden_dim=H, num_classes=C)

print('Testing test-time forward pass ... ')
model.params['W1'] = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
model.params['b1'] = np.linspace(-0.1, 0.9, num=H)
model.params['W2'] = np.linspace(-0.3, 0.4, num=H*C).reshape(H, C)
model.params['b2'] = np.linspace(-0.9, 0.1, num=C)
X = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.loss(X)
correct_scores = np.asarray(
  [[11.53165108,  12.2917344,   13.05181771,  13.81190102,  14.57198434, 15.33206765,  16.09215096],
   [12.05769098,  12.74614105,  13.43459113,  14.1230412,   14.81149128, 15.49994135,  16.18839143],
   [12.58373087,  13.20054771,  13.81736455,  14.43418138,  15.05099822, 15.66781506,  16.2846319 ]])
scores_diff = np.abs(scores - correct_scores).sum()
assert scores_diff < 1e-6, 'Problem with test-time forward pass'

print('Testing training loss (no regularization)')
y = np.asarray([0, 5, 1])
loss, grads = model.loss(X, y)
correct_loss = 3.4702243556
assert abs(loss - correct_loss) < 1e-10, 'Problem with training-time loss'


model.reg = 1.0
loss, grads = model.loss(X, y)
correct_loss = 26.5948426952
assert abs(loss - correct_loss) < 1e-10, 'Problem with regularization loss'

for reg in [0.0, 0.7]:
    print('Running numeric gradient check with reg = ', reg)
    model.reg = 0
    loss, grads = model.loss(X, y)

    for name in sorted(grads):
        f = lambda _: model.loss(X, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name], verbose=False)
        print('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
```

```
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 1.22e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
Running numeric gradient check with reg =  0.7
W1 relative error: 1.22e-08
W2 relative error: 3.48e-10
b1 relative error: 6.55e-09
b2 relative error: 4.33e-10
```

Solver

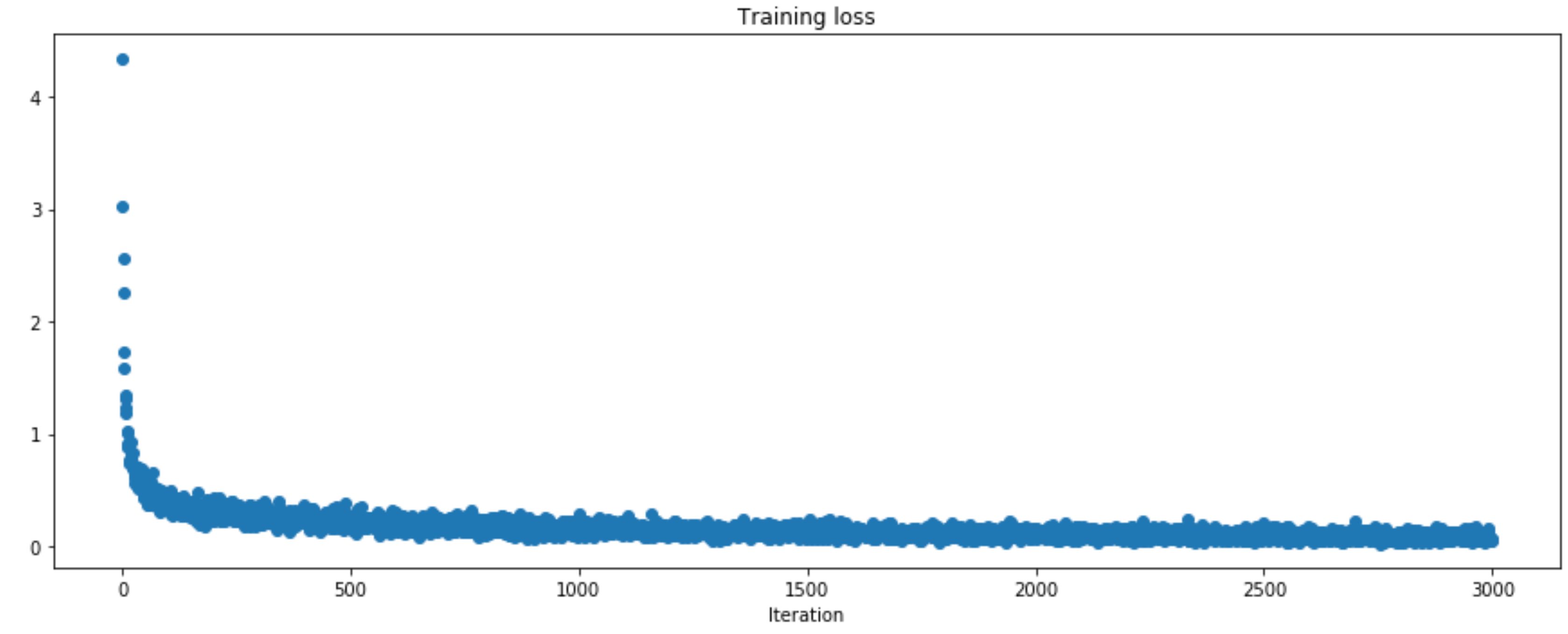
We use a separate class to define the training process.

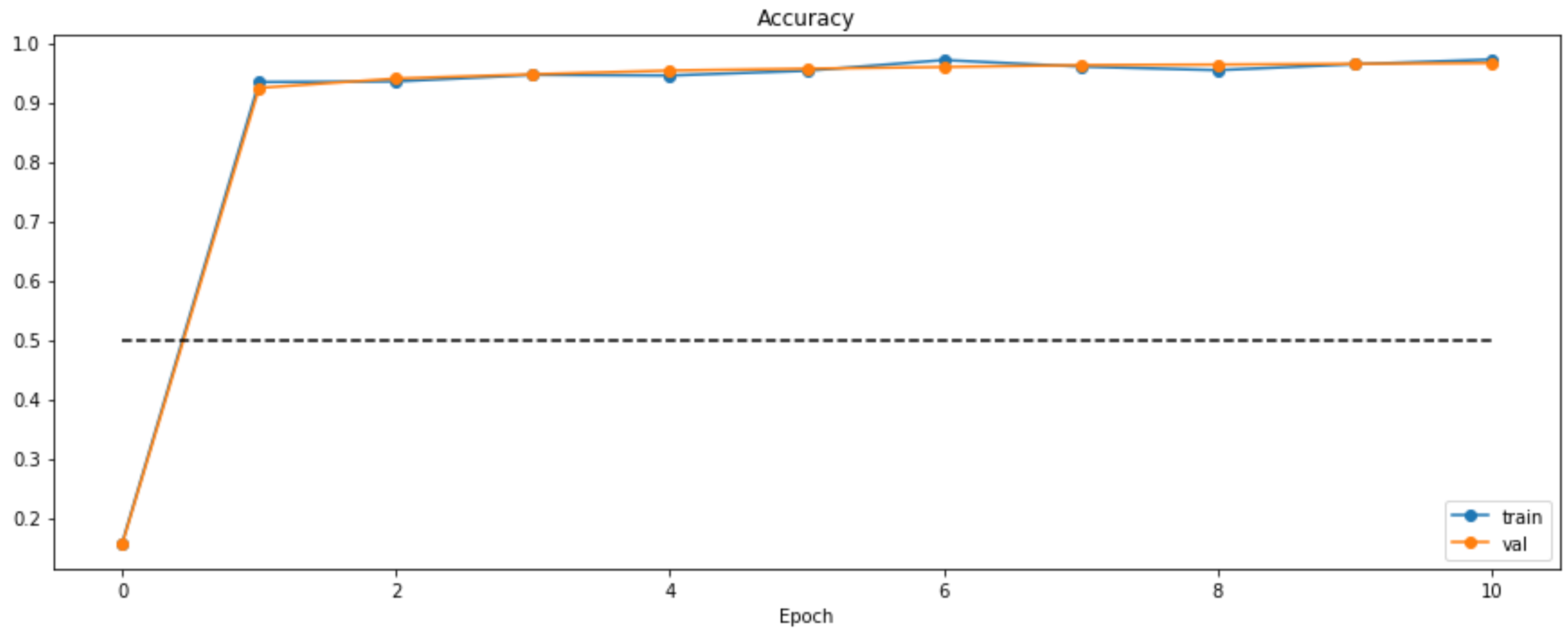
Open the file `stats202a/solver.py` and read through it to familiarize yourself with the API. You can use a `Solver` instance to train a `TwoLayerNet` that achieves at least `97%` accuracy on the validation set. Just run the code.

In [60]:
```
model = TwoLayerNet()
solver = None

solver = Solver(model, data,
              update_rule='sgd',
              optim_config={
                'learning_rate': 1e-3,
              },
              lr_decay=0.95,
              num_epochs=10, batch_size=200,
              print_every=100)
solver.train()
```

```
(Iteration 1 / 3000) loss: 3.639049
(Epoch 0 / 10) train acc: 0.184000; val_acc: 0.206300
(Iteration 101 / 3000) loss: 0.342776
(Iteration 201 / 3000) loss: 0.304021
(Epoch 1 / 10) train acc: 0.912000; val_acc: 0.923400
(Iteration 301 / 3000) loss: 0.371488
(Iteration 401 / 3000) loss: 0.294997
(Iteration 501 / 3000) loss: 0.167035
(Epoch 2 / 10) train acc: 0.956000; val_acc: 0.939600
(Iteration 601 / 3000) loss: 0.122843
(Iteration 701 / 3000) loss: 0.219473
(Iteration 801 / 3000) loss: 0.199194
(Epoch 3 / 10) train acc: 0.958000; val_acc: 0.949200
(Iteration 901 / 3000) loss: 0.141391
(Iteration 1001 / 3000) loss: 0.146663
(Iteration 1101 / 3000) loss: 0.104937
(Epoch 4 / 10) train acc: 0.952000; val_acc: 0.954400
(Iteration 1201 / 3000) loss: 0.092791
(Iteration 1301 / 3000) loss: 0.136841
(Iteration 1401 / 3000) loss: 0.110019
(Epoch 5 / 10) train acc: 0.961000; val_acc: 0.960100
(Iteration 1501 / 3000) loss: 0.108234
(Iteration 1601 / 3000) loss: 0.104657
(Iteration 1701 / 3000) loss: 0.056309
(Epoch 6 / 10) train acc: 0.967000; val_acc: 0.960400
(Iteration 1801 / 3000) loss: 0.116699
(Iteration 1901 / 3000) loss: 0.070217
(Iteration 2001 / 3000) loss: 0.172099
(Epoch 7 / 10) train acc: 0.968000; val_acc: 0.962400
(Iteration 2101 / 3000) loss: 0.155883
(Iteration 2201 / 3000) loss: 0.055427
(Iteration 2301 / 3000) loss: 0.060005
(Epoch 8 / 10) train acc: 0.966000; val_acc: 0.965500
(Iteration 2401 / 3000) loss: 0.126344
(Iteration 2501 / 3000) loss: 0.098041
(Iteration 2601 / 3000) loss: 0.067354
(Epoch 9 / 10) train acc: 0.964000; val_acc: 0.965200
(Iteration 2701 / 3000) loss: 0.082672
(Iteration 2801 / 3000) loss: 0.083628
(Iteration 2901 / 3000) loss: 0.058326
(Epoch 10 / 10) train acc: 0.979000; val_acc: 0.966700
```

In [40]:
```
# Run this cell to visualize training loss and train / val accuracy

plt.subplot(2, 1, 1)
plt.title('Training loss')
plt.plot(solver.loss_history, 'o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(solver.train_acc_history, '-o', label='train')
plt.plot(solver.val_acc_history, '-o', label='val')
plt.plot([0.5] * len(solver.val_acc_history), 'k--')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```


Training loss
4
3
2
1
0
0
500
1000
1500
2000
2500
3000
Iteration


Accuracy
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10
Epoch
train
val

Multilayer network (Optional)

Next you will implement a fully-connected network with an arbitrary number of hidden layers.

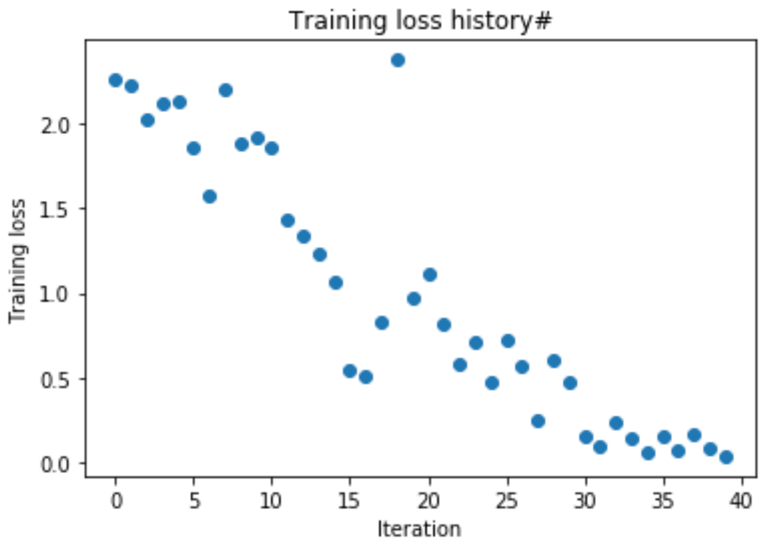
Read through the `FullyConnectedNet` class in the file `stats202a/classifiers/fc_net.py`.

Implement the initialization, the forward pass, and the backward pass. For the moment don't worry about implementing dropout or batch normalization; we will add those features soon.

As a sanity check, make sure you can overfit a small dataset of 50 images. First we will try a three-layer network with 100 units in each hidden layer. You will need to tweak the learning rate and initialization scale, but you should be able to overfit and achieve 100% training accuracy within 20 epochs.

In [61]:
```
# TODO: Use a three-layer Net to overfit 50 training examples.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

small_data['X_train'].shape

weight_scale = 1e-2
learning_rate = 1e-2
model = FullyConnectedNet([100, 100],
              weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history#')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

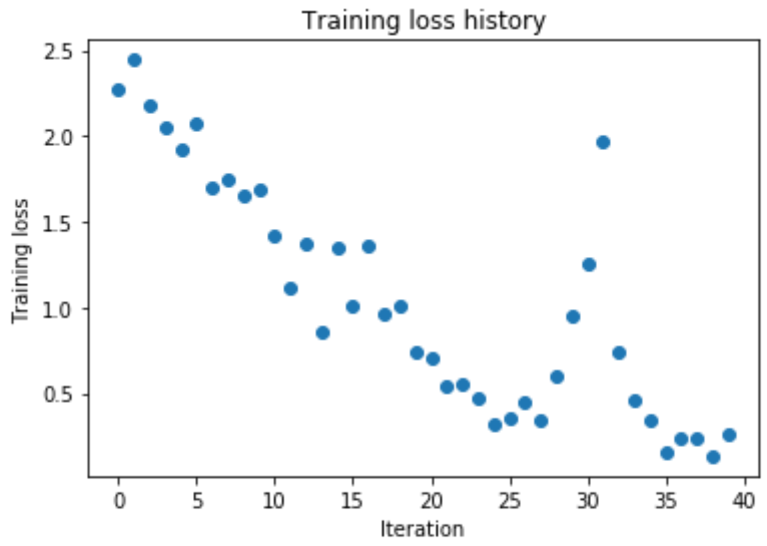
```
(Iteration 1 / 40) loss: 2.259765
(Epoch 0 / 20) train acc: 0.320000; val_acc: 0.207900
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.207300
(Epoch 2 / 20) train acc: 0.340000; val_acc: 0.238300
(Epoch 3 / 20) train acc: 0.500000; val_acc: 0.345000
(Epoch 4 / 20) train acc: 0.420000; val_acc: 0.290200
(Epoch 5 / 20) train acc: 0.600000; val_acc: 0.394700
(Iteration 11 / 40) loss: 1.859239
(Epoch 6 / 20) train acc: 0.660000; val_acc: 0.371100
(Epoch 7 / 20) train acc: 0.840000; val_acc: 0.480400
(Epoch 8 / 20) train acc: 0.800000; val_acc: 0.469900
(Epoch 9 / 20) train acc: 0.300000; val_acc: 0.172200
(Epoch 10 / 20) train acc: 0.620000; val_acc: 0.425500
(Iteration 21 / 40) loss: 1.111941
(Epoch 11 / 20) train acc: 0.820000; val_acc: 0.498400
(Epoch 12 / 20) train acc: 0.720000; val_acc: 0.449200
(Epoch 13 / 20) train acc: 0.860000; val_acc: 0.500000
(Epoch 14 / 20) train acc: 0.840000; val_acc: 0.554000
(Epoch 15 / 20) train acc: 0.980000; val_acc: 0.584700
(Iteration 31 / 40) loss: 0.159379
(Epoch 16 / 20) train acc: 0.980000; val_acc: 0.576400
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.593500
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.559700
(Epoch 19 / 20) train acc: 0.960000; val_acc: 0.551500
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.615700
```



Now try to use a five-layer network with 100 units on each layer to overfit 50 training examples. Again you will have to adjust the learning rate and weight initialization, but you should be able to achieve 100% training accuracy within 20 epochs.

In [59]:
```
# TODO: Use a five-layer Net to overfit 50 training examples.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

learning_rate = 1e-2
weight_scale = 4e-2
model = FullyConnectedNet([100, 100, 100, 100],
                weight_scale=weight_scale, dtype=np.float64)
solver = Solver(model, small_data,
                print_every=10, num_epochs=20, batch_size=25,
                update_rule='sgd',
                optim_config={
                  'learning_rate': learning_rate,
                }
         )
solver.train()

plt.plot(solver.loss_history, 'o')
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.show()
```

```
(Iteration 1 / 40) loss: 2.276575
(Epoch 0 / 20) train acc: 0.100000; val_acc: 0.074300
(Epoch 1 / 20) train acc: 0.240000; val_acc: 0.124300
(Epoch 2 / 20) train acc: 0.300000; val_acc: 0.157800
(Epoch 3 / 20) train acc: 0.580000; val_acc: 0.246200
(Epoch 4 / 20) train acc: 0.720000; val_acc: 0.282100
(Epoch 5 / 20) train acc: 0.620000; val_acc: 0.291500
(Iteration 11 / 40) loss: 1.416126
(Epoch 6 / 20) train acc: 0.520000; val_acc: 0.287800
(Epoch 7 / 20) train acc: 0.660000; val_acc: 0.312700
(Epoch 8 / 20) train acc: 0.820000; val_acc: 0.362400
(Epoch 9 / 20) train acc: 0.820000; val_acc: 0.402200
(Epoch 10 / 20) train acc: 0.840000; val_acc: 0.421400
(Iteration 21 / 40) loss: 0.705214
(Epoch 11 / 20) train acc: 0.900000; val_acc: 0.409400
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.479000
(Epoch 13 / 20) train acc: 0.880000; val_acc: 0.391200
(Epoch 14 / 20) train acc: 0.920000; val_acc: 0.400200
(Epoch 15 / 20) train acc: 0.660000; val_acc: 0.297100
(Iteration 31 / 40) loss: 1.254005
(Epoch 16 / 20) train acc: 0.760000; val_acc: 0.365200
(Epoch 17 / 20) train acc: 0.980000; val_acc: 0.487300
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.512500
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.509200
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.514600
```



Part 3:
Tensorflow and Pytorch

(a) Tensorflow

In [ ]:
```
# Official tutorials: https://www.tensorflow.org/tutorials
# keras totutial for MNIST: https://www.tensorflow.org/tutorials/quickstart/beginner
```

In [2]:
```
import numpy as np
import tensorflow as tf
```

In [3]:
```
# Build an easy calculator
# Placeholders allow us to not provide the data in
# advance for operations and computational graphs.
a = tf.placeholder(dtype=tf.float32, shape=[3,3])
b = tf.placeholder(dtype=tf.float32, shape=[3,3])
c = a+b
d = tf.matmul(a, b)
print(a)
print(b)
print(c)
print(d)
```

```
Tensor("Placeholder:0", shape=(3, 3), dtype=float32)
Tensor("Placeholder_1:0", shape=(3, 3), dtype=float32)
Tensor("add:0", shape=(3, 3), dtype=float32)
Tensor("MatMul:0", shape=(3, 3), dtype=float32)
```

In [4]:
```
sess = tf.Session() # A session allows us to execute computations on graphs
a_input = np.array([[1,1,1],[2,2,2],[3,3,3]])
b_input = np.array([[1,2,3],[1,2,3],[1,2,3]])
my_feed_dict = {a: a_input, b: b_input}
# sess.run performs matrix addition and multipication, as defined above on
# the values specified in the feed_dict, a_input and b_input.
# The session also allocates memory to store the current value of the variable.
res = sess.run([c,d], feed_dict=my_feed_dict)
print(res[0])
print(res[1])
```

```
[[2. 3. 4.]
 [3. 4. 5.]
 [4. 5. 6.]]
[[ 3.  6.  9.]
 [ 6. 12. 18.]
 [ 9. 18. 27.]]
```

In [15]:
```
# From the TensorFlow documentation:
# Calling tf.Variable() adds several operations to the graph:
#   * A variable op that holds the variable value.
#   * An initializer op that sets the variable to its initial value.
#   * The ops for the initial value.
e = tf.Variable(0.0)

# tf.assign(ref,value) outputs a tensor that holds the new value of
# the tensor 'ref' after the value has been assigned.
e_add = tf.assign(e, e+1)
```

In [16]:
```
# print(sess.run(e))

# Running this code gives the following error:
# FailedPreconditionError: Attempting to use uninitialized value Variable
# As we can see, the value of a variable is only valid within a session.
```

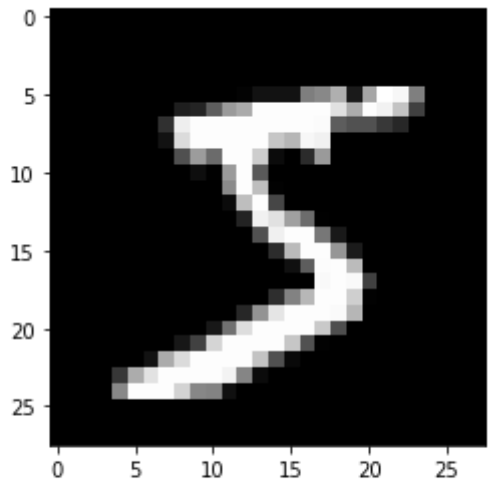
In [17]:
```
# tf.global_variables_initializer(): Returns an Op that initializes global variables.
sess.run(tf.global_variables_initializer())
print(sess.run(e))
sess.run(e_add)
print(sess.run(e))
```

```
0.0
1.0
```

In [111]:
```
# build an easy neuron network
# load in the data
mnist = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Convert the samples from integers to floating-point numbers:
x_train, x_test = x_train / 255.0, x_test / 255.0
print(x_train.shape)
print(y_train.shape)
```

```
(60000, 28, 28)
(60000,)
```

In [29]:
```
import matplotlib.pyplot as plt
plt.imshow(x_train[0], cmap='gray')
plt.show()
```



In [30]:
```
# define structure: 784-->256-->10
input_img = tf.placeholder(dtype=tf.float32, shape=[None, 28*28], name='input')
labels = tf.placeholder(dtype=tf.int32, shape=[None], name='label')
h1 = tf.layers.dense(input_img, units=256) # Add a fully-connected layer
h1 = tf.nn.relu(h1) # Add a relu activation
h2 = tf.layers.dense(h1, units=10) # Add a fully-connected layer
# tf.nn.softmax() performs the equivalent of:
# softmax = tf.exp(logits) / tf.reduce_sum(tf.exp(logits), axis)
# The softmax function is commonly used in the final layer of a neural network-based classifier.
output = tf.nn.softmax(h2) # Use the softmax function

print(h1.shape)
print(h2.shape)
print(output.shape)
print(labels.shape)
```

```
(?, 256)
(?, 10)
(?, 10)
(?,)
```

In [31]:
```
# Define loss and optimizer:

# tf.nn.sparse_softmax_cross_entropy_with_logits:
# Measures the probability error in discrete classification tasks
# in which the classes are mutually exclusive.
loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=labels, logits=output, name='loss')

# tf.train.GradientDescentOptimizer(): defines an optimizer that implements
# the gradient descent algorithm.
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01)

# optimizer.minimize(): Implements compute_gradients() and then updates the
# var_list using apply_gradients().
update = optimizer.minimize(loss)
```

In [97]:
```
# init = tf.initialize_all_variables()

# #
# with tf.Session() as sess:
#     sess.run(init)

#     # Training cycle
#     for epoch in np.arange(training_epochs):
#         avg_cost = 0.
#         total_batch = int(num_examples/batch_size)
#         # Loop over all batches
#         for i in np.arange(total_batch):
#             cur_input = np.reshape(x_train[i:i+10], (10, 784)) # Flatten training data
#             cur_label = y_train[i:i+10] # Image labels
#             my_feed_dict = {input_img:cur_input, labels:cur_label}
#             _,c = sess.run([output, update], feed_dict=my_feed_dict)

#     # Test model
#     correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y_test, 1))
#     # Calculate accuracy
#     accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
#     print("Accuracy:", accuracy.eval(x_test, y_test))
```

In [203]:
```
sess = tf.Session()
sess.run(tf.global_variables_initializer())
cur_input = np.reshape(x_train[0:10], (10, 784))
cur_label = y_train[0:10]
my_feed_dict = {input_img:cur_input, labels:cur_label}
pred,_ = sess.run([output, update], feed_dict=my_feed_dict)
print(pred[0])
# print(np.size(pred))
pred_new = sess.run(output, feed_dict=my_feed_dict)
print(pred_new[0])
# print(np.size(pred_new))

# correct_pred = tf.equal(tf.argmax(pred, 1), tf.argmax(y_test, 1))
# acc = tf.reduce_mean(tf.cast(correct_pred, "float"))
# print("Accuracy:", acc.eval({input_img: x_test, labels: y_test}))
```

```
[0.13398984 0.04544385 0.13270596 0.14252584 0.0579179  0.06027409
 0.14276955 0.10662371 0.05689097 0.12085825]
[0.13653363 0.04713428 0.13327356 0.14421006 0.05956748 0.06402764
 0.13740341 0.10266164 0.05586866 0.11931965]
```

In [187]:
```
# USING KERAS:

from __future__ import absolute_import, division, print_function, unicode_literals

batch_size = 10000
num_epochs = 5
loss ='sparse_categorical_crossentropy'
optimizer = 'adam' # Use Adam optimizer. Improves final accuracy by ~7%.

# batch = np.random.shuffle([x_train,y_train])

# Shuffle the data and take the first 10,000 images:
c = np.c_[x_train.reshape(len(x_train), -1), y_train.reshape(len(y_train), -1)]
a2 = c[:, :x_train.size//len(x_train)].reshape(x_train.shape)
b2 = c[:, x_train.size//len(x_train):].reshape(y_train.shape)

# Reshape the data for passing to 2D convolution:
x_train_b = a2[0:batch_size]
y_train_b = b2[0:batch_size]

# Model 1 Structure:
# Fully Connected -> ReLu -> Dropout -> Fully Connected -> Softmax
model1 = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model1.compile(optimizer=optimizer,
              loss=loss,
              metrics=['accuracy'])

history1 = model1.fit(x_train_b, y_train_b, epochs=num_epochs, shuffle = 'True')

model1.evaluate(x_test,  y_test, verbose=2)

loss1 = history1.history['loss']
acc1 = history1.history['acc']
```

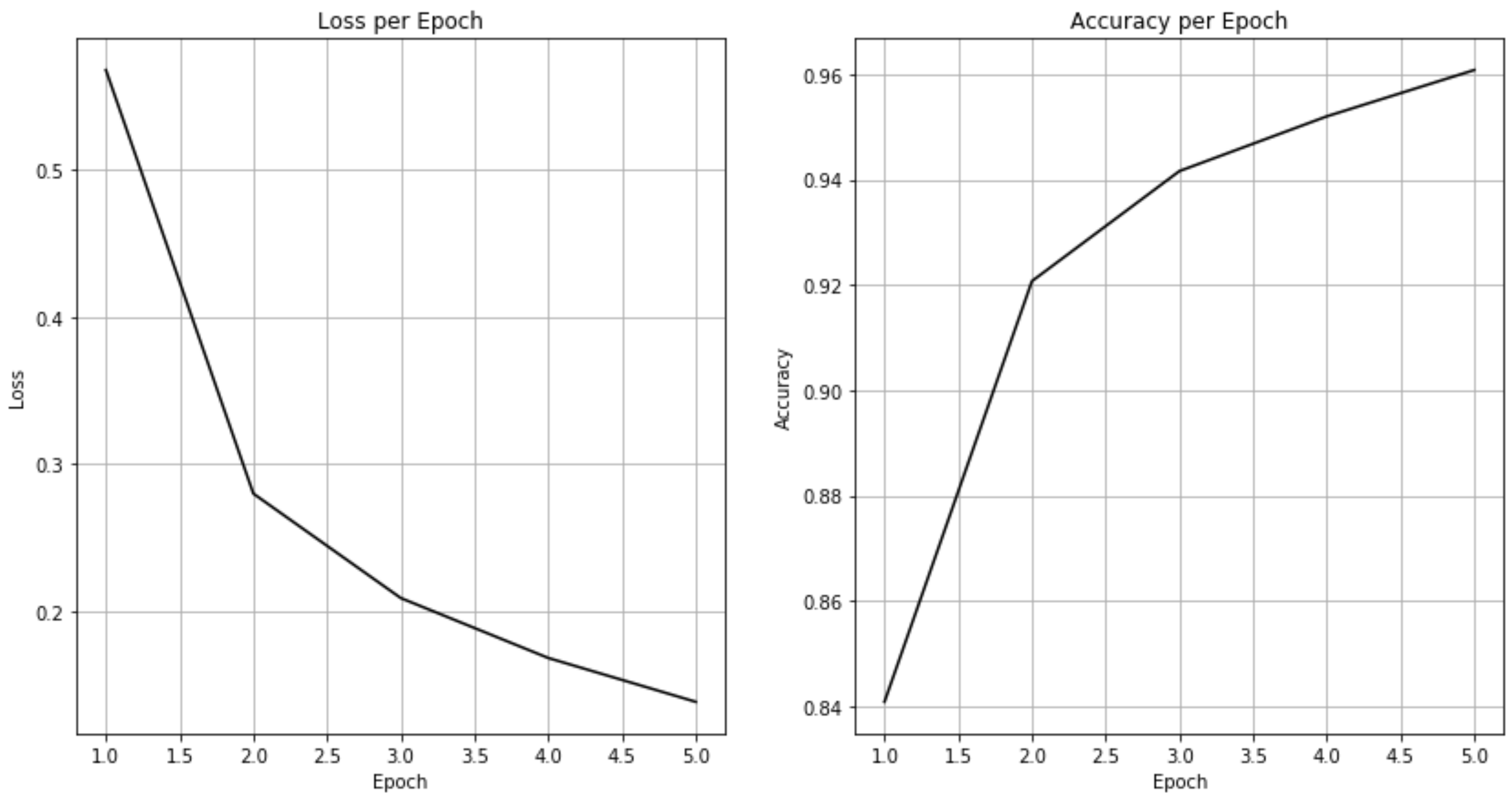
```
Train on 10000 samples
Epoch 1/5

10000/10000 [==============================] - 5s 474us/sample - loss: 0.5672 - acc: 0.8408s - loss:
```

```
10000/10000 [==============================] - 3s 171us/sample - loss: 0.5072 - acc: 0.8108 - loss:
0.5875 - acc: 0.
Epoch 2/5
10000/10000 [==============================] - 3s 328us/sample - loss: 0.2800 - acc: 0.9207
Epoch 3/5
10000/10000 [==============================] - 3s 297us/sample - loss: 0.2092 - acc: 0.9416
Epoch 4/5
10000/10000 [==============================] - 3s 328us/sample - loss: 0.1686 - acc: 0.9520
Epoch 5/5
10000/10000 [==============================] - 4s 360us/sample - loss: 0.1388 - acc: 0.9608
10000/10000 - 1s - loss: 0.1884 - acc: 0.9419
```

In [194]:

```
import matplotlib.pyplot as plt

def plot_la(num_epochs, loss, acc):
    epoch_v = np.linspace(1,num_epochs,num_epochs)

    ## Plot of the loss vs epoch:
    fig, (ax1, ax2) = plt.subplots(1, 2, sharey=False, figsize = (14,7))
    # fig, ax = plt.subplots(1, 1, figsize = (7,7))
    ax1.plot(epoch_v, loss, '-', color = "black")
    ax1.set_title('Loss per Epoch')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.grid()

    ## Plot of the accuracy vs epoch:
    ax2.plot(epoch_v, acc, '-', color = "black")
    ax2.set_title('Accuracy per Epoch')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy')
    ax2.grid()
    plt.show()
```

```
In [195]: plot_la(num_epochs, loss1, acc1)
```



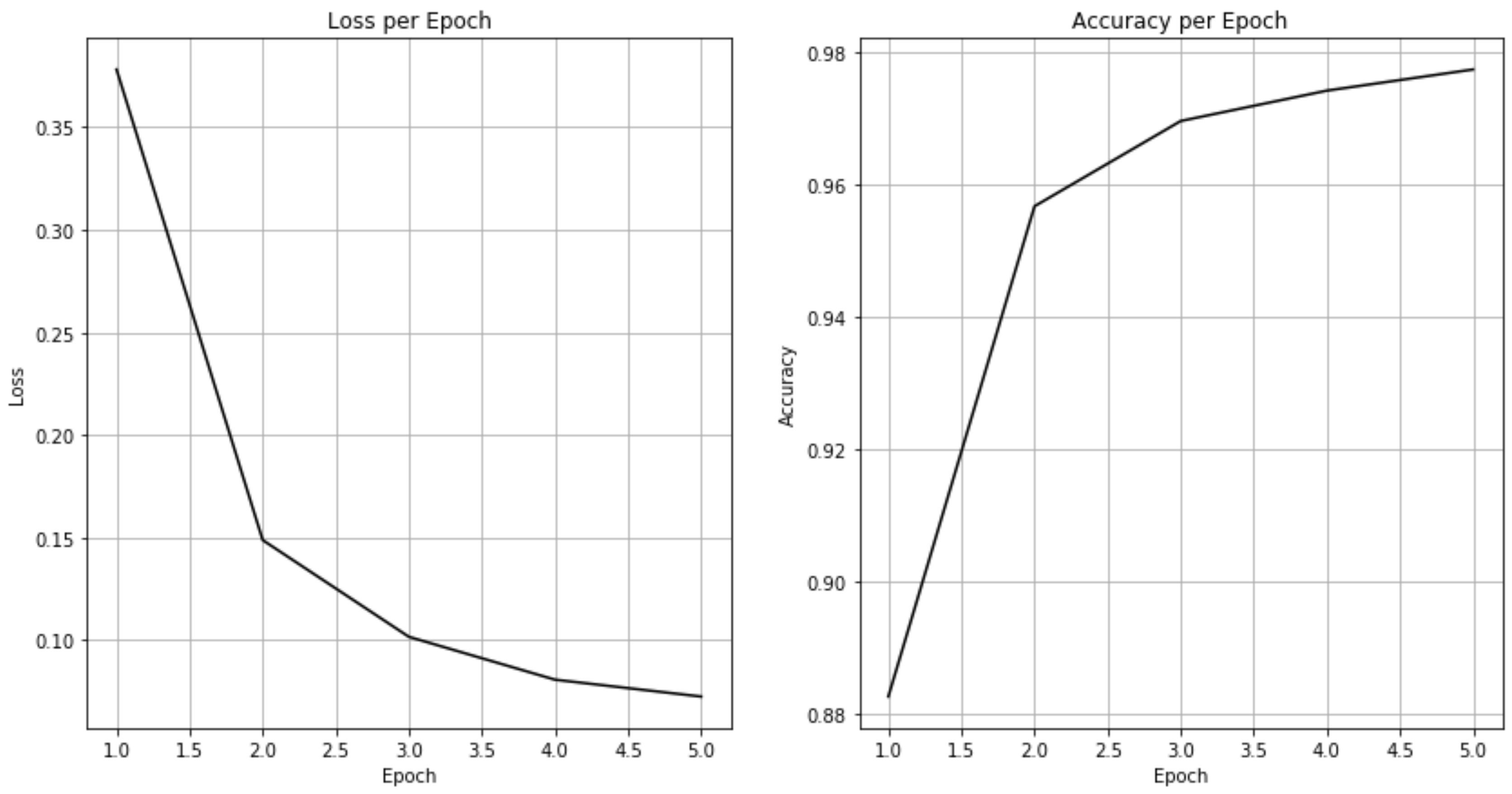
(Fully Connected Network)

In [199]:
```
batch_size = 10000
num_epochs = 5
loss ='sparse_categorical_crossentropy'
optimizer = 'adam'

# Shuffle the data and take the first 10,000 images:
c = np.c_[x_train.reshape(len(x_train), -1), y_train.reshape(len(y_train), -1)]
a2 = c[:, :x_train.size//len(x_train)].reshape(x_train.shape)
b2 = c[:, x_train.size//len(x_train):].reshape(y_train.shape)
x_train_b = a2[0:batch_size]
y_train_b = b2[0:batch_size]

# Reshape the data for passing to 2D convolution:
x_train_b = np.reshape(x_train_b, (batch_size, 28, 28, 1))
x_test_b = np.reshape(x_test, (int(np.size(x_test)/784), 28, 28, 1))

# Model 2 Structure:
# (2D Convolution -> ReLu -> Batch Norm -> Max Pool -> Dropout)x2 -> Fully Connected
# -> ReLu -> Batch Norm -> Dropout -> Fully Connected --> Softmax
model2 = tf.keras.models.Sequential([
  tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation=tf.nn.relu, input_shape=(28, 28, 1)),
  tf.keras.layers.BatchNormalization(),
  tf.keras.layers.MaxPool2D((2, 2)),
  tf.keras.layers.Dropout(0.20),
  tf.keras.layers.Conv2D(64, (3, 3), activation=tf.nn.relu, padding='same'),
  tf.keras.layers.BatchNormalization(),
  tf.keras.layers.MaxPool2D(pool_size=(2, 2)),
  tf.keras.layers.Dropout(0.25),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.BatchNormalization(),
  tf.keras.layers.Dropout(0.30),
  tf.keras.layers.Dense(10, activation='softmax')
])

model2.compile(optimizer=optimizer,
              loss=loss,
              metrics=['accuracy'])

history2 = model2.fit(x_train_b, y_train_b, epochs=num_epochs, shuffle = 'True')

model2.evaluate(x_test_b,  y_test, verbose=2)
```

```
loss2 = history2.history['loss']
acc2 = history2.history['acc']
```

```
Train on 10000 samples
Epoch 1/5
10000/10000 [==============================] - 36s 4ms/sample - loss: 0.3782 - acc: 0.8825
Epoch 2/5
10000/10000 [==============================] - 35s 3ms/sample - loss: 0.1487 - acc: 0.9567
Epoch 3/5
10000/10000 [==============================] - 35s 4ms/sample - loss: 0.1016 - acc: 0.9696
Epoch 4/5
10000/10000 [==============================] - 35s 4ms/sample - loss: 0.0807 - acc: 0.9742
Epoch 5/5
10000/10000 [==============================] - 33s 3ms/sample - loss: 0.0725 - acc: 0.9774
10000/10000 - 11s - loss: 0.0632 - acc: 0.9800
```

In [200]:
```
plot_la(num_epochs, loss2, acc2)
```



In [ ]:
```
# Testing accuracy: 98%. Accuracy improved by about 4%.
```

(Convolutional Network)

(b) Pytorch

In [1]:
```
#how to use google colab: https://pytorch.org/tutorials/beginner/colab.html
#pytorch tutorial: https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
import torch
import torch.nn.functional as F
from torchvision import datasets, transforms
import numpy as np
```

In [48]:
```
# define the network structure
class fc_net(torch.nn.Module):
    def __init__(self, num_in, num_out):
        super(fc_net, self).__init__()
        self.h1 = torch.nn.Linear(in_features=num_in, out_features=256)  # Fully-connected layer
        self.h2 = torch.nn.Linear(in_features=256, out_features=num_out) # Fully-connected layer
    def forward(self, inputs):
        a1 = F.relu(self.h1(inputs)) # ReLu activation
        #     print('Lets see a1')
        #     print(a1[0, 0:10])
        a2 = F.softmax(self.h2(a1),dim=-1) # Softmax activation
        return a2
```

In [3]:
```
# use data_loader to load_in data
train_data = datasets.MNIST('./', train=True, download=True, transform=transforms.Compose([transforms.To
train_loader = torch.utils.data.DataLoader(train_data, batch_size=10, shuffle=False)
cur_x, cur_y = next(iter(train_loader))
print(cur_x.size()) # x_train for current iteration
print(cur_y.size()) # y_train for current iteration
```

```
0it [00:00, ?it/s]

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz (http://yann.lecun.com/exdb/mn
ist/train-images-idx3-ubyte.gz) to ./MNIST/raw/train-images-idx3-ubyte.gz

100%|██████████▉| 9904128/9912422 [00:25<00:00, 378711.05it/s]

Extracting ./MNIST/raw/train-images-idx3-ubyte.gz to ./MNIST/raw


0it [00:00, ?it/s]
  0%|          | 0/28881 [00:00<?, ?it/s]

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz (http://yann.lecun.com/exdb/mn
ist/train-labels-idx1-ubyte.gz) to ./MNIST/raw/train-labels-idx1-ubyte.gz

32768it [00:00, 117099.58it/s]

0it [00:00, ?it/s]

Extracting ./MNIST/raw/train-labels-idx1-ubyte.gz to ./MNIST/raw
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz (http://yann.lecun.com/exdb/mni
st/t10k-images-idx3-ubyte.gz) to ./MNIST/raw/t10k-images-idx3-ubyte.gz


  0%|          | 0/1648877 [00:00<?, ?it/s]
  1%|▏         | 24576/1648877 [00:00<00:06, 234340.71it/s]
  3%|▎         | 57344/1648877 [00:00<00:07, 224313.26it/s]
  5%|▌         | 90112/1648877 [00:00<00:07, 220499.36it/s]
  7%|▋         | 122880/1648877 [00:00<00:06, 239669.75it/s]
  9%|▉         | 155648/1648877 [00:00<00:06, 242588.47it/s]
 11%|█▏        | 188416/1648877 [00:00<00:05, 252671.88it/s]
 14%|█▍        | 229376/1648877 [00:01<00:05, 269898.17it/s]
 16%|█▋        | 270336/1648877 [00:01<00:04, 286168.01it/s]
 19%|█▉        | 319488/1648877 [00:01<00:04, 311459.47it/s]
 22%|██▏       | 360448/1648877 [00:01<00:04, 298281.34it/s]
 24%|██▍       | 393216/1648877 [00:01<00:04, 289927.54it/s]
 26%|██▋       | 434176/1648877 [00:01<00:04, 300132.79it/s]
 29%|██▉       | 483328/1648877 [00:01<00:03, 329531.81it/s]
 32%|███▏      | 532480/1648877 [00:01<00:03, 362678.46it/s]
 35%|███▌      | 581632/1648877 [00:02<00:02, 380483.59it/s]
 38%|███▊      | 622592/1648877 [00:02<00:02, 354508.56it/s]
 42%|████▏     | 688128/1648877 [00:02<00:02, 398109.84it/s]
 45%|████▍     | 737280/1648877 [00:02<00:02, 406582.61it/s]
 48%|████▊     | 786432/1648877 [00:02<00:02, 400930.25it/s]
 51%|█████     | 843776/1648877 [00:02<00:01, 435134.76it/s]
 54%|█████▍    | 892928/1648877 [00:02<00:01, 441088.49it/s]
 57%|█████▋    | 942080/1648877 [00:02<00:01, 401173.03it/s]
 61%|██████    | 1007616/1648877 [00:03<00:01, 431119.77it/s]
 64%|██████▍   | 1056768/1648877 [00:03<00:01, 428409.77it/s]
 68%|██████▊   | 1114112/1648877 [00:03<00:01, 447108.76it/s]
 71%|███████   | 1163264/1648877 [00:03<00:01, 451376.60it/s]
 75%|███████▍  | 1228800/1648877 [00:03<00:00, 490839.37it/s]
 78%|███████▊  | 1286144/1648877 [00:03<00:00, 480042.84it/s]
 81%|████████  | 1335296/1648877 [00:03<00:00, 474807.75it/s]
 84%|████████▍ | 1392640/1648877 [00:03<00:00, 478806.59it/s]
 89%|████████▉ | 1466368/1648877 [00:04<00:00, 492165.30it/s]
 92%|█████████▏| 1523712/1648877 [00:04<00:00, 501065.51it/s]
 96%|█████████▌| 1581056/1648877 [00:04<00:00, 506141.79it/s]
 99%|█████████▉| 1638400/1648877 [00:04<00:00, 440198.64it/s]

Extracting ./MNIST/raw/t10k-images-idx3-ubyte.gz to ./MNIST/raw



0it [00:00, ?it/s]

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz (http://yann.lecun.com/exdb/mni
st/t10k-labels-idx1-ubyte.gz) to ./MNIST/raw/t10k-labels-idx1-ubyte.gz



8192it [00:00, 34959.43it/s]

Extracting ./MNIST/raw/t10k-labels-idx1-ubyte.gz to ./MNIST/raw
Processing...
Done!
torch.Size([10, 1, 28, 28])
torch.Size([10])

9920512it [00:39, 378711.05it/s]
```

```
1654784it [00:22, 440198.64it/s]
```

In [4]:
```
cur_x = torch.reshape(cur_x, (10, 28*28))
model = fc_net(num_in=28*28, num_out=10)
loss = torch.nn.CrossEntropyLoss() # Cross entropy loss
optimizer = torch.optim.SGD(model.parameters(), lr=0.1) # Stochastic gradient descent
preds = model.forward(cur_x) #
cur_loss = loss(preds, cur_y)
optimizer.zero_grad()
cur_loss.backward() # Backprop
optimizer.step() # Step of optimizer
new_preds = model.forward(cur_x)
print(preds[0])
print(new_preds[0])
```

```
Lets see a1
tensor([0.0269, 0.0000, 0.0969, 0.1192, 0.0074, 0.0000, 0.0000, 0.0415, 0.1895,
        0.2509], grad_fn=<SliceBackward>)
Lets see a1
tensor([0.0234, 0.0000, 0.0966, 0.1182, 0.0175, 0.0000, 0.0000, 0.0432, 0.1860,
        0.2472], grad_fn=<SliceBackward>)
tensor([0.0935, 0.0959, 0.0962, 0.1083, 0.1061, 0.1088, 0.0930, 0.1009, 0.1017,
        0.0955], grad_fn=<SelectBackward>)
tensor([0.0937, 0.0970, 0.0960, 0.1090, 0.1061, 0.1104, 0.0921, 0.1001, 0.1006,
        0.0949], grad_fn=<SelectBackward>)
```

In [5]:
```
# convert between numpy and tensor
pred_array = preds[0].detach().numpy()
print(pred_array)
pred_tensor = torch.from_numpy(pred_array).float()
print(pred_tensor)
```

```
[0.09353404 0.09589709 0.09619637 0.10834455 0.10609396 0.10876822
 0.09299684 0.10093759 0.10171144 0.09551988]
tensor([0.0935, 0.0959, 0.0962, 0.1083, 0.1061, 0.1088, 0.0930, 0.1009, 0.1017,
        0.0955])
```

In [ ]:
```
#######################################################################################################
```

In [116]:
```
# use data_loader to load_in data
train_data = datasets.MNIST('./', train=True, download=True, transform=transforms.Compose([transforms.To
train_loader = torch.utils.data.DataLoader(train_data, batch_size=10, shuffle=False)
cur_x, cur_y = next(iter(train_loader))
print(cur_x.size()) # x_train for current iteration
print(cur_y.size()) # y_train for current iteration

batch_size = 100
test_data = datasets.MNIST('./', train=False, download=True, transform=transforms.Compose([transforms.To
test_loader = torch.utils.data.DataLoader(test_data,batch_size=batch_size, shuffle=False)
test_x, test_y = next(iter(test_loader))
print(test_x.size()) # x_test for current iteration
print(test_y.size()) # y_test for current iteration
```

```
torch.Size([10, 1, 28, 28])
torch.Size([10])
torch.Size([100, 1, 28, 28])
torch.Size([100])
```

In [259]:
```
# Linear Network

# Network Structure:
# Fully Connected -> ReLu -> Fully Connected -> Softmax
class fc_net2(torch.nn.Module):
    def __init__(self, num_in, num_out):
        super(fc_net2, self).__init__()
        self.h1 = torch.nn.Linear(in_features=num_in, out_features=256)  # Fully-connected layer
        self.h2 = torch.nn.Linear(in_features=256, out_features=num_out) # Fully-connected layer

    def forward(self, inputs):
        a1 = F.relu(self.h1(inputs)) # ReLu activation
        a2 = F.softmax(self.h2(a1),dim=-1) # Softmax activation
        return a2

# Train the model:
def train(model,loss,train_loader,optimizer,epoch):
    model.train()
    print('Train Epoch: ', epoch)
    for batch_idx, (cur_x, cur_y) in enumerate(train_loader):
        cur_x = torch.reshape(cur_x, (10, 28*28))
        optimizer.zero_grad() # Zero out gradients
        output = model.forward(cur_x) # Propagate forward through network
        cur_loss = loss(output, cur_y) # Compute loss at current iteration
        cur_loss.backward() # Backprop
        optimizer.step() # Step of optimizer

        if batch_idx % 1000 == 0:
            print('    Iteration: ', batch_idx, ' Loss: ', round(cur_loss.item(),5))

# Test the model:
def test(model,loss,test_loader):
    model.eval()
    print('Test Set:')

    test_loss = 0
    correct = 0
    with torch.no_grad():
        for batch_idx, (cur_x, cur_y) in enumerate(test_loader):
            cur_x = torch.reshape(cur_x, (100, 28*28))
            output = model.forward(cur_x)
            test_loss = loss(output, cur_y)
            preds = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += preds.eq(cur_y.view_as(preds)).sum().item()
            acc = correct/len(test_loader.dataset)

        test_loss /= len(test_loader.dataset)
        print('    Average loss: ', format(test_loss.item(),"10.2E") , ' Accuracy: ', 100.*acc, '%')
    return(test_loss, acc)
```

In [262]:
```
# Run the model:
model = fc_net2(num_in=28*28, num_out=10)
loss = torch.nn.CrossEntropyLoss() # Cross entropy loss
# opt = torch.optim.SGD(model.parameters(), lr=0.1) # SGD
opt = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.4) # SGD with momentum
# opt = torch.optim.Adam(model.parameters(), lr=0.1) # Adam

num_epochs = 5
loss_v = np.zeros(num_epochs)
acc_v = np.zeros(num_epochs)
for epoch in np.arange(num_epochs)+1:
    train(model,loss,train_loader,opt,epoch)
    [test_loss, acc] = test(model,loss,test_loader)
    loss_v[epoch-1] = test_loss
    acc_v[epoch-1] = acc
```

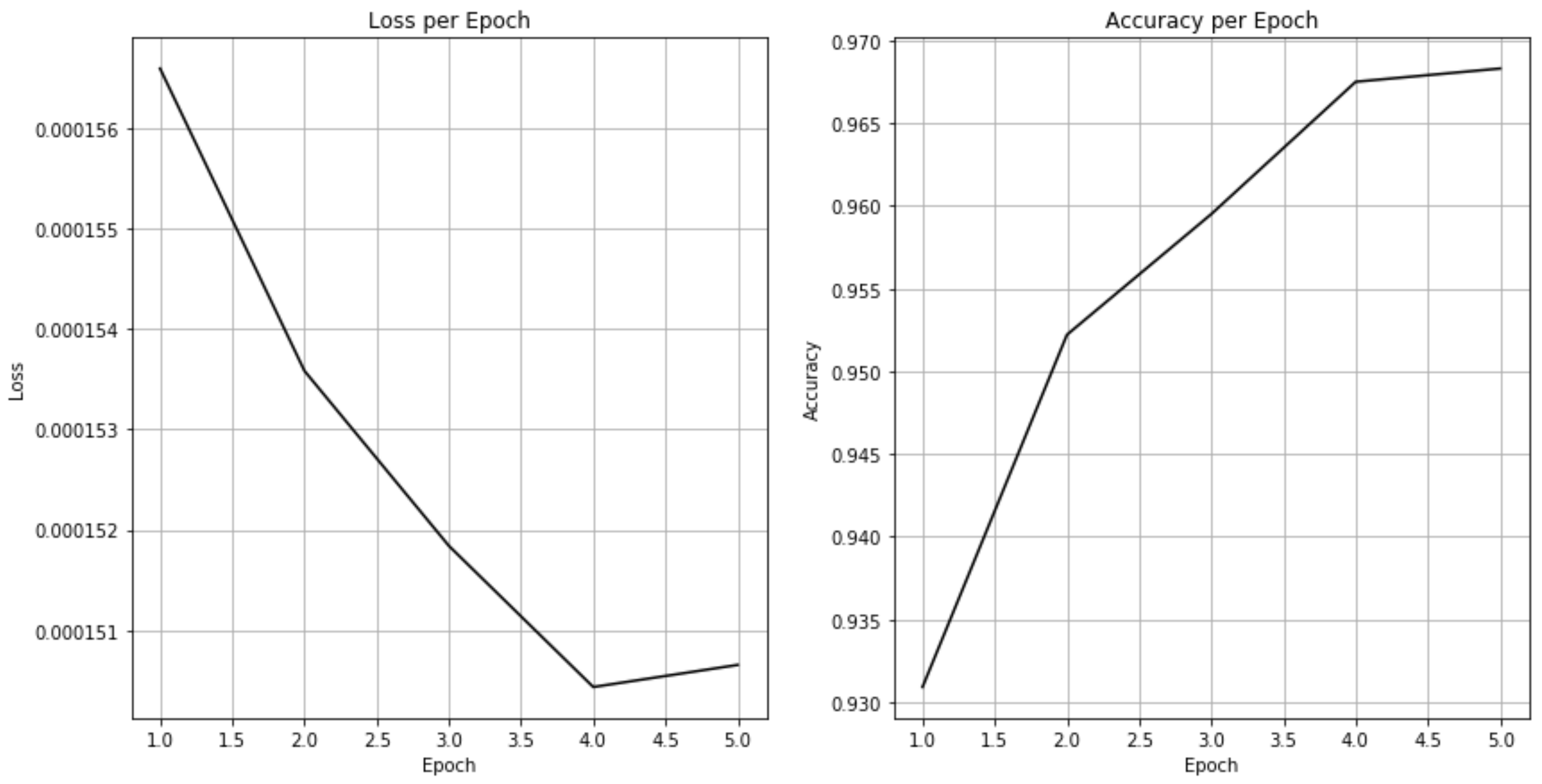
```
Train Epoch:  1
    Iteration:  0  Loss:  2.30048
    Iteration:  1000  Loss:  1.58543
    Iteration:  2000  Loss:  1.60427
    Iteration:  3000  Loss:  1.46302
    Iteration:  4000  Loss:  1.51104
    Iteration:  5000  Loss:  1.54892
Test Set:
    Average loss:    1.57E-04  Accuracy:  93.08999999999999 %
Train Epoch:  2
    Iteration:  0  Loss:  1.46353
    Iteration:  1000  Loss:  1.59995
    Iteration:  2000  Loss:  1.52683
    Iteration:  3000  Loss:  1.46962
    Iteration:  4000  Loss:  1.46201
    Iteration:  5000  Loss:  1.49187
Test Set:
    Average loss:    1.54E-04  Accuracy:  95.22 %
Train Epoch:  3
    Iteration:  0  Loss:  1.46147
    Iteration:  1000  Loss:  1.5685
    Iteration:  2000  Loss:  1.52554
    Iteration:  3000  Loss:  1.51249
    Iteration:  4000  Loss:  1.46235
    Iteration:  5000  Loss:  1.47619
Test Set:
    Average loss:    1.52E-04  Accuracy:  95.95 %
```

```
Train Epoch:  4
    Iteration:  0  Loss:  1.46146
    Iteration:  1000  Loss:  1.47153
    Iteration:  2000  Loss:  1.49725
    Iteration:  3000  Loss:  1.4626
    Iteration:  4000  Loss:  1.46242
    Iteration:  5000  Loss:  1.47561
Test Set:
    Average loss:    1.50E-04  Accuracy:  96.75 %
Train Epoch:  5
    Iteration:  0  Loss:  1.46139
    Iteration:  1000  Loss:  1.56915
    Iteration:  2000  Loss:  1.46777
    Iteration:  3000  Loss:  1.47214
    Iteration:  4000  Loss:  1.46146
    Iteration:  5000  Loss:  1.4641
Test Set:
    Average loss:    1.51E-04  Accuracy:  96.83 %
```

In [263]:

```python
import matplotlib.pyplot as plt

# Plot function:
def plot_la(num_epochs, loss, acc):
    epoch_v = np.linspace(1,num_epochs,num_epochs)

    ## Plot of the loss vs epoch:
    fig, (ax1, ax2) = plt.subplots(1, 2, sharey=False, figsize = (14,7))
    # fig, ax = plt.subplots(1, 1, figsize = (7,7))
    ax1.plot(epoch_v, loss, '-', color = "black")
    ax1.set_title('Loss per Epoch')
    ax1.set_xlabel('Epoch')
    ax1.set_ylabel('Loss')
    ax1.grid()

    ## Plot of the accuracy vs epoch:
    ax2.plot(epoch_v, acc, '-', color = "black")
    ax2.set_title('Accuracy per Epoch')
    ax2.set_xlabel('Epoch')
    ax2.set_ylabel('Accuracy')
    ax2.grid()
    plt.show()
```

```
In [264]: plot_la(num_epochs, loss_v, acc_v)
```



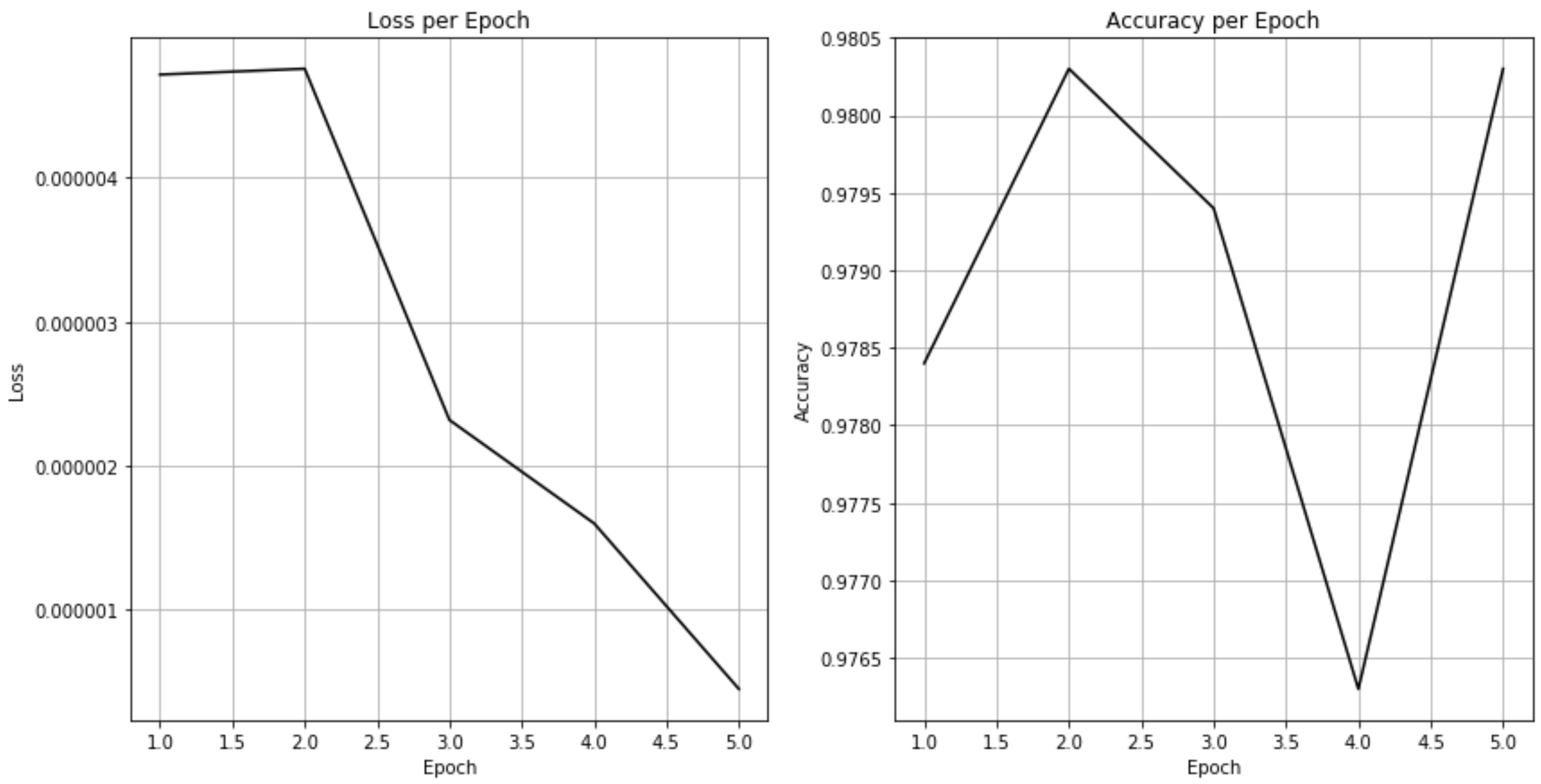
(Fully Connected Network)

In [270]:
```
# Convolutional Network

# Network Structure:
    # (2D Convolution -> ReLu -> Max Pool)*2 ->
    # Fully Connected -> ReLu -> Fully Connected -> Softmax
class ConvNet(torch.nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.conv1 = torch.nn.Conv2d(1, 4, 3)
        self.conv2 = torch.nn.Conv2d(4, 8, 3)
        self.drop1 = torch.nn.Dropout2d(0.25)
        self.drop2 = torch.nn.Dropout2d(0.35)
        self.fc1 = torch.nn.Linear(200, 128)
        self.fc2 = torch.nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, (2, 2))
#         x = self.drop1(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, (2,2))
#         x = self.drop2(x)
        num_features = np.prod(x.size()[1:])
        x = x.view(-1, num_features)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
#         x = F.softmax(x,dim=1)
        x = F.log_softmax(x, dim=1)
        return x

# Train the model
def train(model,loss,train_loader,optimizer,epoch):
    model.train()
    print('Train Epoch: ', epoch)
    for batch_idx, (cur_x, cur_y) in enumerate(train_loader):
        cur_x = torch.reshape(cur_x, (10,1,28,28))
        optimizer.zero_grad() # Zero out gradient
        output = model.forward(cur_x) # Propagate forward through network
        cur_loss = loss(output, cur_y) # Compute loss at current iteration
        cur_loss.backward() # Backprop
        optimizer.step() # Step of optimizer

        # Print:
        if batch_idx % 1000 == 0:
            print('    Iteration: ', batch_idx, ' Loss: ', round(cur_loss.item(),5))

# Test the network
def test(model,loss,test_loader):
    model.eval()
    print('Test Set:')

    test_loss = 0
    correct = 0
    with torch.no_grad():
        for batch_idx, (cur_x, cur_y) in enumerate(test_loader):
            cur_x = torch.reshape(cur_x, (100,1,28,28))
            output = model.forward(cur_x)
            test_loss = loss(output, cur_y)
            preds = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += preds.eq(cur_y.view_as(preds)).sum().item()
            acc = correct/len(test_loader.dataset)

        test_loss /= len(test_loader.dataset)
        print('    Average loss: ', format(test_loss.item(),"10.2E") , ' Accuracy: ', 100.*acc, '%')
    return(test_loss, acc)
```

In [271]:
```python
# Run the model:
model = ConvNet()
loss = torch.nn.CrossEntropyLoss() # Cross entropy loss
opt = torch.optim.SGD(model.parameters(), lr=0.1) # SGD with momentum

num_epochs = 5
loss_v = np.zeros(num_epochs)
acc_v = np.zeros(num_epochs)
for epoch in np.arange(num_epochs)+1:
    train(model,loss,train_loader,opt,epoch)
    [test_loss, acc] = test(model,loss,test_loader)
    loss_v[epoch-1] = test_loss
    acc_v[epoch-1] = acc
```

```
Train Epoch:  1
    Iteration:  0  Loss:  2.31596
    Iteration:  1000  Loss:  0.0558
    Iteration:  2000  Loss:  0.24577
    Iteration:  3000  Loss:  0.00249
    Iteration:  4000  Loss:  0.01135
    Iteration:  5000  Loss:  0.00636
Test Set:
    Average loss:    4.72E-06  Accuracy:  97.84 %
Train Epoch:  2
    Iteration:  0  Loss:  0.00148
    Iteration:  1000  Loss:  0.02163
    Iteration:  2000  Loss:  0.00878
    Iteration:  3000  Loss:  0.00223
    Iteration:  4000  Loss:  0.01746
    Iteration:  5000  Loss:  0.00238
Test Set:
    Average loss:    4.76E-06  Accuracy:  98.03 %
Train Epoch:  3
    Iteration:  0  Loss:  0.00071
    Iteration:  1000  Loss:  0.00259
    Iteration:  2000  Loss:  0.05724
    Iteration:  3000  Loss:  0.00098
    Iteration:  4000  Loss:  0.00859
    Iteration:  5000  Loss:  0.00139
Test Set:
    Average loss:    2.32E-06  Accuracy:  97.94 %
Train Epoch:  4
    Iteration:  0  Loss:  0.00035
    Iteration:  1000  Loss:  0.00131
    Iteration:  2000  Loss:  0.1831
    Iteration:  3000  Loss:  0.00577
    Iteration:  4000  Loss:  0.1079
    Iteration:  5000  Loss:  0.00202
Test Set:
    Average loss:    1.60E-06  Accuracy:  97.63 %
Train Epoch:  5
    Iteration:  0  Loss:  0.00015
    Iteration:  1000  Loss:  0.00283
    Iteration:  2000  Loss:  0.00074
    Iteration:  3000  Loss:  1e-05
    Iteration:  4000  Loss:  0.00115
    Iteration:  5000  Loss:  0.03427
Test Set:
    Average loss:    4.48E-07  Accuracy:  98.03 %
```

In [272]:
```
plot_la(num_epochs, loss_v, acc_v)
```



(Convolutional Network)

Part 4:
Homework Memo

HOMEWORK 1: Sampling in Python & R

In this assignment, I created a uniform pseudorandom number generator on the interval [0,1] using the linear congruential method. I set a seed as the last 4 digits of the current time in milliseconds and used the recursive update rule $X_{i+1} = (aX_i + b) \bmod M$, where M is a large integer, in this case $2^{31} - 1$. I checked the function visually by plotting a histogram of the uniformly distributed pseudo-random samples X_i, using 100,000 samples and 100 bins. I also plotted a scatterplot of (X_i, X_i+1) for all i in the sample set. This two-dimensional plot is a visual check that adjacent points in the distribution are uncorrelated.

I next created a function to sample from an exponential distribution, using the inversion method. In the inversion method, we first generate a uniform distribution U on [0,1] and then let $X = F^{-1}(U)$, where F is the probability distribution we wish to sample from. I checked the function visually by plotting a histogram with 200 bins of 100,000 sampled points, which can be seen to follow an exponential distribution.

Thirdly, I built a function to sample from a 2-dimensional Gaussian distribution, with mean (0,0) and standard deviation of 1, using the polar method. In the polar method, we sample from the normal distribution by first generating independent uniform distributions U_1, U_2 on [0,1], then computing $\theta = 2\pi U_1$ and $R = \sqrt{-2\log(1 - U_2)}$, and then letting $X = R\cos(\theta)$ and $Y = R\sin(\theta)$. I checked the function visually by plotting a scatterplot of the distribution of 100,000 samples in two dimensions, and plotting a histogram of 100,000 samples of $T = R^2/2$, where R is the distance of a point from the origin.

Fourthly, I wrote code for Monte Carlo computation of π, by generating (X_t, Y_t) from the unit square $[0, 1]^2$, and computing the frequency that the points fall below $x^2 + y^2 = 1$. I then used a Monte Carlo method to compute the volume of a d-dimensional unit ball, for $d = 2, 3, 5, 10$, and compared the results to the theoretical values.

Functions:

sample_uniform(low=0, high=1)

> Samples pseudorandom number from a uniform distribution from low to high, using the linear congruential method. Plots scatter plot.
> Output: None

sample_exponential(k=1)

> Samples from an exponential distribution, using the inversion method. Plots histogram.
> Output: None

sample_normal(mean=0,var=1)

> Samples from a normal distribution, using the polar method. Plots scatter plot, histogram.
> Output: None

monte_carlo(d=2)

> Samples from two independent uniform distribution and computes a Monte Carlo computation of π. Computes the volume of a ball of dimension d. Print results.
> Output: None

HOMEWORK 2: Metropolis & Gibbs Sampling in Python and R

I implemented a Monte Carlo Markov Chain (MCMC) sampling method to sample a target distribution $\pi(x)$. At time step t and state Xt, we generate $Xt+1$ by modifying Xt, so that $P(Xt+1 = y \mid Xt = x, Xt-1, ..., X0) = K(x, y)$, where $K(x, y)$ is the transition probability, and where we assume the Markov Property: $P(Xt+1 = y \mid Xt = x, Xt-1, ..., X0) = P(Xt+1 = y \mid Xt = x)$. The Metropolis Algorithm is an MCMC algorithms which is implemented as follows: at time t, let the current state be $Xt = x$. We generate $U \sim$ uniform[0,1]. If $U \leq \pi(y)/\pi(x)$, then we let $Xt+1 = y$, otherwise we let $Xt+1 = x$. I used the Metropolis algorithm to sample from $N(0, 1)$. The proposal distribution at x is $y \sim$ Uniform$[x\text{-}c, x+c]$. I ran 1,000 parallel chains with $x_0 \sim$ uniform$[a, b]$ and made a movie for the change of the histogram of x_t. I also experimented with different values of $[a, b]$ and c.

I next implemented a Gibbs sampler (another MCMC algorithm). The Gibbs sampler samples from a multivariate distribution $\pi(x)$, where $x = (x_1,...,x_k,...,x_d)$. A step k, the algorithm updates the state $x_k \sim \pi(x_k|x_{-k})$, where $x-k$ denotes the current values of all the other components. In the case of a bivariate normal distribution, letting (Xt, Yt) be the values of (X, Y) at iteration t, at iteration $t + 1$ we sample $X_{t+1} \sim N(\rho Y_t, 1 - \rho^2)$, and then sample $Y_{t+1} \sim N(\rho X_{t+1}, 1 - \rho^2)$. I ran a visual check by running 1000 parallel chains from the same starting point and making a movie of the scatterplot of (x_t, y_t). I also ran a single chain for T steps, discarding the first B steps, and plotted the scatterplot of the footsteps for the rest of the steps. Finally, I experimented with different values of ρ, and demonstrated that the sampled density collapses to a point when $|\rho| = 1$.

<u>Functions:</u>

sample_uniform(size=1000, low=0, high=1)

    Samples from a uniform distribution.
    Ouput: An array of uniform random values

sample_normal_chain(x0, c, chain_length=100, mean=0, var=1)

    Returns multiple chains by Metropolis sampling from N(mean, var).
    For every train, the proposal distribution at x is $y \sim$ Uniform$[x\text{-}c, x+c]$.
    Output: An array of sampled values

metropolis_simulation (num_chain=1000, chain_length=100, mean=0, var=1)

    Simulates the metropolis algorithm with different settings. Shows a movie.
    Output: None

gibbs_sample(x0, y0, rho, num_chain=1000, chain_length=100, mean=0, var=1)

    Returns multiple chains by Gibbs sampling.
    Output: Array of Gibbs samples

gibbs_simulation()

    Parameters are inside function. Simulates Gibbs sampling with different ρ and plots.
    Output: None

HOMEWORK 3: The Sweep Operator

I implemented a sweep operator function in R, which takes an input matrix A and value k and outputs a swept matrix B, with the pivot element A_{kk}. The sweep operator is a fundamental operator in regression which performs elementary row operators on matrix equations. The algorithm can be executed efficiently in Python or R by vectorizing the operations in the following manner: We define the pivot element $m = A_{kk}$, and then compute the negative outer product of A's kth column $A_{:k}$ and row $A_{k:}$ divided by m: $B = A - (A_{:k} \otimes A_{k:})/m$. We then overwrite the kth row of B as $B_{k:} = A_{k:}/m$, we overwrite the kth row of B as $B_{:k} = -B_{:k}/m$, and we overwrite the kkth element of B as $B_{kk} = 1/m$. This ordering of operations allows the matrix B to overwritten in place without the use of placeholder variables.

The sweep operator allows us to efficiently compute least squares estimates and build regression models. Given a data matrix X and label vector Y, we seek to build a model of the form $Y = X\beta + \varepsilon$. By "sweeping in" or "sweeping out" certain rows of X^TX, the sweep operator allows us to simultaneously update our estimates of the regression coefficients, the error sum of squares, and the Moore-Penrose pseudoinverse of the data matrix. We first construct the uncorrected sum of squares and cross-products matrix M (the USSCP matrix), which in block form contains the submatrices X^TX, X^TY, Y^TX, and Y^TY. We then sweep M along the rows of X^TX. Our regression coefficient estimates are the upper right block of the swept matrix.

Functions:
myLinearRegression(X, Y)
Computes linear regression coefficient estimates, given data X and labels Y.
Output: β estimates

HOMEWORK 4: QR Decomposition & Linear Regression

I built a function, *myQR*, which given an input matrix A, computes an upper triangular matrix T and a unitary matrix U such that $A = UTU^*$ is the Schur decomposition of A. The function $A_0 = A$ and $U_0 = I$, computes the QR decomposition of A_{k-1} as $A_{k-1} = Q_kR_k$ (Recall that the QR decomposition produces an orthogonal matrix Q and an upper triangular matrix R), and then applies the recursion relations $A_k = R_kQ_k$ and $U_k = U_{k-1}Q_k$. In the limit that k goes to infinity, A_k and U_k approach T and U, respectively.

To improve the speed of the algorithm, we first reduce the matrix A to Hessenberg form, that is a matrix whose elements below the lower off-diagonal are zero. The Hessenberg form is preserved by the QR algorithm described above, and introducing the lower off-diagonal zeros results in significant computational savings. A matrix can be reduced to Hessenberg form using Givens Rotations or Householder Reflections. If A is $n\times n$, $n-1$ Givens Rotations are required to transform the matrix to upper diagonal form. After transforming A to an upper Hessenberg matrix H, the algorithm performs the recursion in the previous paragraph by employing Givens Rotations to overwrite H with $H' = RQ$, where $H = QR$ is the QR factorization of H.

Alternatively, we can use Householder Reflections. A Householder Reflector is a matrix of the form $P = I - 2uu^*$, $\|u\| = 1$. Householder Reflectors are Hermitian and unitary. We can reduce a matrix A to Hessenberg form by repeated application of Householder Reflections. We first perform the recursion $P_k = I_k \oplus (I_{n-\mathrm{k}} - 2u_k u_k^*)$ and then update $A = AP_k$ and $U = P_k U$.

Finally, we can use our QR decomposition algorithm to efficiently perform linear regression. Given input data matrix X and labels Y, we first compute the QR decomposition of X. We then solve the equation $R\beta = Q^{\mathrm{T}}Y$. Since R is upper triangular, this equation can be efficiently solving from the bottom row of the matrix to the top. In particular, we solve for the last (the pth) element of β as $\beta_p = (Q^{\mathrm{T}}Y)_p/R_{pp}$ and then solve the recursion relation: $\beta_i = [(Q^{\mathrm{T}}Y)_i - R_{i,i+1:p}\, \beta_{i+1:p}]/R_{ii}$.

Functions:
givens(a,b)
    Computes Givens rotation matrix.
    Output: Given rotation matrix.
myQR(A)
    Performs QR decomposition on the matrix A.
    Ouput: A list containing the matrices Q and R.
myLinearRegression(X, Y)
    Perform the linear regression on data matrix X and label vector Y.
    Output: Estimated β and mean squared error.

HOMEWORK 5: Eigen-Decomposition and PCA

QR decomposition can be used to efficiently compute eigenvectors and eigenvalues. The algorithm is as follows: Initialize $A_i = A$ and $U_i = I$. Compute the QR decomposition of $A_i = QR$ and recursively update A_i and U_i using the relations $A_i = RQ$ and $U_i = U_iQ$. As the number of recursions tends toward infinity, A_i and U_i converge, respectively, to a diagonal matrix of the eigenvalues D and a matrix Q of eigenvectors.

Using our eigen-decomposition, we can efficiently compute a principal component analysis (PCA). Given a data matrix $X \in \mathrm{R}^{n\times p}$, we first compute μ, an array of the means along the columns of X, and then compute $B = X - X_\mu$, where $X_\mu = 1^{\mathrm{T}}\mu$. We then compute the QR decomposition of $C = B^{\mathrm{T}}B/(n-1)$ and compute $Z = XQ$.

Functions:
myEigen_QR(A, numIter = 1000)
    Computes eigenvectors & eigenvalues for A using myQR.
    Output: A list of eigenvectors and eigenvalues
myPCA(X)
    Performs PCA on matrix X using myEigen_QR().
    Output: A basis matrix Q and data matrix Z, such that $X = ZQ^{\mathrm{T}}$.

HOMEWORK 6: Logistic Regression, Adaboost, & XGBoost

Using the *myQR* function for QR decomposition from Homework 5, I built a function *myLogisticSolution* to perform logistic regression. Given input data X and labels Y, the function runs the Newton-Raphson algorithm to compute estimates of β in a logistic model, with β

initialized as a zero vector. At each iteration, the algorithm first computes the variance matrix $V_m = \text{diag}(p(1-p))$, where $p = \exp(X\beta)/(1+\exp(X\beta))$ and then computes β with the recursion relation $\beta_{i+1} = \beta_i + (X^T V_m X)^{-1}(X^T Y - p)$. We can compute the inverse of $M = X^T V_m X$ efficiently by using the *myQR* function to compute the QR decomposition of M. Then, $M^{-1} = R^{-1}Q^T$. We can compute R^{-1} recursively, using the relation $R^{-T}_{ik} = -(R^T_{i,k:(i-1)}R^{-T}_{k:(i-1),k})/R^T_{ii}$. The function *myLogisticSolution* inputs X and Y and outputs the estimates of β.

I next implemented Adaptive Boosting (AdaBoost) in the *myAdaboost* function. The AdaBoost algorithm constructs a boost classifier of the form $F_T(x) = \sum_{t=1}^{T} f_t(x)$, where the f_t are weak classifiers that take inputs x and return predicted classes. The algorithm functions as follows: at each iteration, the function *ensemble* constructs an ensemble of linear classifiers $\{k_i\}$ with slope and y-intercept values sampled from a uniform distribution. The algorithm chooses the classifier k_m that minimizes the total weighted error $\sum_{y_i \neq k_m(x_i)} w_i^{(m)}$, uses this to compute the error rate $\varepsilon_m = \sum_{y_i \neq k_m(x_i)} w_i^{(m)} / \sum_{i=1}^{N} w_i^{(m)}$, and then calculates the classifier weight $\alpha_m = \frac{1}{2}\ln\left(\frac{1-\varepsilon_m}{\varepsilon_m}\right)$. Finally, the classifier C_{m-1} is boosted to $C_m = C_{m-1} + \alpha_m k_m$.

I then implemented Extreme Gradient Boosting (XGBoost) in the *myXGBoost* function. This function divides the plane into four randomly drawn quadrants. It then constructs a one-layer decision tree on the quadrants. As in AdaBoost, we add a decision tree to the main classifier by determining the weight that would minimize a loss function.

Functions:

myLogisticSolution(X, Y)

  Performs logistic regression on input data matrix X and label vector Y.

  Output: Estimated β coefficients.

myAdaboost(x1, x2, y)

  Performs Adaptive Boosting. Plots the classification results and prints accuracy.

  Output: None

  grad(X,Y,W,b)

    Calculates the gradient.

    Output: Returns db, the gradient with respect to b.

  gradient_descent(X,Y,W,b,l_rate,num_iter)

    Performs gradient descent on b.

    Output: Updated b.

  ensemble(X,Y,range,n_ensmb)

    Creates a random ensemble of weak linear classifiers.

    Output: Ct, a list of W, b, and α values for each classifier in the ensemble.

  predict(X,Y,C)

    Given data matrix X and classifier C, this function predicts the labels.

    Output: Yp, an array of predicted labels.

  L_exp(X,Y,C,At,s=1)

    Computes the exponential loss function.

    Output: Exponential loss.

  Choose(X,Y,C,Ct)

    Chooses the weak classifier that minimizes the total weighted error.

Output: At, a list of the *W* and *b* values for the chosen weak classifier.

New_Weight(X,Y,C,At)

Calculates the new weight for the newly chosen weak classifier.

Output: Weight for the newly chosen weak classifier.

Update_Classifier(X,Y,C,At)

Updates the main classifier with the newly chosen weak classifier.

Output: C, a list of list of *W*, *b*, and α values for the updated classifier.

myXGBoost(x1, x2, y)

Performs the Extreme Gradient Boosting algorithm. Plots results and prints accuracy.

Output: None

Loss(y,yp)

Computes the loss function.

Output: The loss, L = sum(1–y*yp)

ChooseQuad(x1,x2,x1t,x2t,y)

Choose point quadrant.

Output: Classifier parameters.

DecisionTree(x1,x2,y,n_ensmb)

Generates a decision tree using *ChooseQuad*.

Output: Classifier parameters

predict(x1,x2,C)

Given x1, x2, and classifier C, this function predicts the labels.

Output: Yp, an array of predicted labels.

LineSearch(x1,x2,y,yp,rp)

Performs a line search on gamma.

Output: Updated gamma.

HOMEWORK 7: *k*-Fold Cross Validation

I used the *KFold* function in *sklearn.model_selection* to batch cancer mortality data into training and testing sets for 5-fold cross validation. I then used the inbuilt *xgb.XGBClassifier* to train an Extreme Gradient Boosting algorithm on the training data. Using a max depth of 3, the model achieved a mean accuracy of 96.7%, with a standard deviation of 2.2%. I then performed a grid search on the max depth and minimum child weight hyperparameters. That is, I trained the model for all combinations of max_depth $\in$ {3,5,7} and min_child_weight $\in$ {0.1, 1, 5}. Of the nine combinations, I the grid search returned an optimal maximum depth of 3 and minimum child weight of 1. Using these optimal parameters, I used the inbuilt *clf.fit.importances* function to compute the F score of the data features.

Functions:

XGB(X,y,max_depth,min_child_weight)

Performs 5-fold validation for cancer mortality data. Print the mean and standard deviation of the 5-fold validation accuracy

Ouput: None

GridXGB(X,y,max_depth,min_child_weight)

Performs grid search for parameters max_depth and min_child_weight.

Prints the grid search mean test score for each parameter combination.

Output: None

XGB_importances(X,y,max_depth,min_child_weight)

Plots the feature importance of the best model.

Output: None

HOMEWORK 8: Support Vector Machine

A set of points $\{v_1, ..., v_N\}$ with binary labels $s_i = \pm 1$, is said to be linearly separable if it is possible to find a hyperplane that strictly separates the two classes. In this case, a linear programming approach can be taken to find a separating hyperplane. To allow for nonlinearly separable data, we introduce the hinge loss, and, since the margin size is proportional to $1/||w||^2$, introduce a term proportional to $||w||^2$ to maximize the margin size. The general support-vector machine thus minimizes the cost function:

$$\frac{1}{n}\sum_{i=1}^{n} \max\{0,1 - y_i(wx_i - b)\} + \lambda\|w\|^2$$

This cost function is convex (and in fact, quadratic), and can be efficiently solved with techniques from quadratic programming. However, more recent approaches instead rely on sub-gradient descent or coordinate descent. Sub-gradient methods are effective in case of nondifferentiable functions (including the max() function, which is not differentiable at zero). I implemented a sub-gradient method in which we take a step in the direction of the partial derivative of a single component of the cost function at a time. In coordinate descent, in contrast, we perform a line search on a single component, continuing until we have reached an optimum for that component, and then continue to the next component, iterating until the cost levels. For convex cost functions, both methods are guaranteed to converge to the global optimum, given a sufficiently small step size.

Introducing the variable $\zeta_i = \max\{0,1 - y_i(wx_i - b)\}$ (the *i*th component of the hinge loss), the minimization problem can be written in primal form:

$$\frac{1}{n}\sum_{i=1}^{n} \zeta_i + \lambda\|w\|^2$$
$$s.t.\ y_i(w \cdot x_i - b) \geq 1 - \zeta_i,$$
$$\zeta_i \geq 0, \qquad \forall i \in [1, n]$$

The dual problem is:

$$\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j K(x_i, x_j)\alpha_i\alpha_j$$
$$s.t. \sum_{i=1}^{n} y_i\alpha_i = 0,\ \ 0 \leq \alpha_i \leq C,\ \ \forall i \in [1, n]$$

Here, K is a kernel function, which in the linear case is simply $K(x_i, x_j) = x_i \cdot x_j$. The above problems can be generalized to nonlinear classification by introducing a nonlinear kernel, e.g. a polynomial kernel $K(x_i, x_j) = (x_i \cdot x_j + \gamma)^d$, a Gaussian radial basis function kernel $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2, \gamma > 0$, or a hyperbolic tangent kernel $K(x_i, x_j) = \tanh(\kappa x_i \cdot x_j + c), \kappa > 0, c < 0$.

I implemented two methods for solving the dual form of the optimization problem. Firstly, I augmented the cost function with the constraints using Lagrange multipliers, resulting in a new cost function, to which one can apply a sub-gradient descent / coordinate descent algorithm.

$$\min_{\alpha}\left(-\sum_{i=1}^{n}\alpha_i + \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i y_j K(x_i, x_j)\alpha_i\alpha_j + \lambda_1\sum_{i=1}^{n} y_i\alpha_i - \lambda_2\sum_{i=1}^{n} y_i\alpha_i + \nu_1(\alpha - C) - \nu_2\alpha\right)$$

An alternative approach is to use the Sequential Minimal Optimization (SMO) algorithm on the dual problem. The SMO algorithm selects the values for two components of α and optimizes the objective value jointly for both values. It then computes b using the calculated values of α, iterating until α converges. I implemented the following version of SMO: iterating over all α_i, if α_i does not satisfy the KKT conditions, select an α_j at random and jointly optimize the constrained maximization problem on α_i and α_j, using the algorithm in: http://cs229.stanford.edu/materials/smo.pdf.

Functions:

Kernel(u,v,gamma=0.1,coef0=0,type="linear")

Computes the kernel, given input vectors and parameters.

Output: Kernel

dK(w,xi,gamma,coef0,type="linear")

Computes the derivative of the kernel, given input vectors and parameters.

Ouput: Kernel derivative

HingeLoss(y,w,x,lambda)

Computes hinge loss, given input y, w, x, and λ.

Ouput: Hinge loss

SubGrad (yi,w,xi,lambda)

Computes the sub-gradient: $dL = -y_i\, dK(w, x_i) + \lambda w$

Ouput: Subgradient

SGD(y,w,x,lambda,scale,iter)

Performs sub-gradient descent on random, individual components using *SubGrad*.

Output: Updated w

CoordDesc(y,x,lambda,scale,iter)

Performs coordinate descent on the dual problem.

Output: Updated w

SMO(y,x,lambda,scale,iter)

Performs sequential minimal optimization on the dual problem.

Output: Updated w