# This is the k-nearest neighbors workbook for ECE 239AS Assignment #2

Please follow the notebook linearly to implement k-nearest neighbors.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with the data, training and evaluating a simple classifier, k-fold cross validation, and as a Python refresher.

# Import the appropriate libraries

In [148]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.data_utils import load_CIFAR10 # function to load the CIFAR-10 dataset.

# Load matplotlib images inline
%matplotlib inline

# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [149]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = 'cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
('Training data shape: ', (50000, 32, 32, 3))
('Training labels shape: ', (50000,))
('Test data shape: ', (10000, 32, 32, 3))
('Test labels shape: ', (10000,))
```

In [150]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse'
, 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```


In [151]:
```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]

# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
((5000, 3072), (500, 3072))
```

# K-nearest neighbors

In the following cells, you will build a KNN classifier and choose hyperparameters via k-fold cross-validation.

In [152]:
```
# Import the KNN class

from nndl import KNN
```

In [153]:
```
# Declare an instance of the knn class.
knn = KNN()

# Train the classifier.
#   We have implemented the training of the KNN classifier.
#   Look at the train function in the KNN class to see what this does.
knn.train(X=X_train, y=y_train)
```

## Questions

(1) Describe what is going on in the function knn.train().

(2) What are the pros and cons of this training step?

## Answers

(1) We are just assigning each point to X and Y.

(2)Pro-Simple implementation. Cons- Memory usage

## KNN prediction

In the following sections, you will implement the functions to calculate the distances of test points to training points, and from this information, predict the class of the KNN.

In [154]:
```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definitio
n of the norm
#   in the code, which is the 2-norm.
# You should only have to fill out the clearly marked sections.

import time
time_start =time.time()

dists_L2 = knn.compute_distances(X=X_test)

print('Time to run code: {}'.format(time.time()-time_start))
print('Frobenius norm of L2 distances: {}'.format(np.linalg.norm(dists_L
2, 'fro')))
```

```
Time to run code: 38.829501152
Frobenius norm of L2 distances: 7906696.07704
```

#### Really slow code

Note: This probably took a while. This is because we use two for loops. We could increase the speed via vectorization, removing the for loops.

If you implemented this correctly, evaluating np.linalg.norm(dists_L2, 'fro') should return: ~7906696

## KNN vectorization

The above code took far too long to run. If we wanted to optimize hyperparameters, it would be time-expensive. Thus, we will speed up the code by vectorizing it, removing the for loops.

In [155]:
```
# Implement the function compute_L2_distances_vectorized() in the KNN cl
ass.
# In this function, you ought to achieve the same L2 distance but WITHOU
T any for loops.
# Note, this is SPECIFIC for the L2 norm.

time_start =time.time()
dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=X_test)
print('Time to run code: {}'.format(time.time()-time_start))
print('Difference in L2 distances between your KNN implementations (shou
ld be 0): {}'.format(np.linalg.norm(dists_L2 - dists_L2_vectorized, 'fr
o')))
```

```
Time to run code: 0.238061904907
Difference in L2 distances between your KNN implementations (should be
0): 0.0
```

**Speedup**

Depending on your computer speed, you should see a 10-100x speed up from vectorization. On our computer, the vectorized form took 0.36 seconds while the naive implementation took 38.3 seconds.

### Implementing the prediction

Now that we have functions to calculate the distances from a test point to given training points, we now implement the function that will predict the test point labels.

In [156]:
```
# Implement the function predict_labels in the KNN class.
# Calculate the training error (num_incorrect / total_samples)
#   from running knn.predict_labels with k=1
yPredicted = knn.predict_labels(dists=dists_L2_vectorized)

error = 1

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the error rate by calling predict_labels on the test
#   data with k = 1.  Store the error rate in the variable error.
# ================================================================ #
error = np.count_nonzero(y_test-yPredicted)/float(len(y_test))
pass
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print(error)
```

```
0.726
```

If you implemented this correctly, the error should be: 0.726.

This means that the k-nearest neighbors classifier is right 27.4% of the time, which is not great, considering that chance levels are 10%.

# Optimizing KNN hyperparameters

In this section, we'll take the KNN classifier that you have constructed and perform cross-validation to choose a best value of $k$, as well as a best choice of norm.

### Create training and validation folds

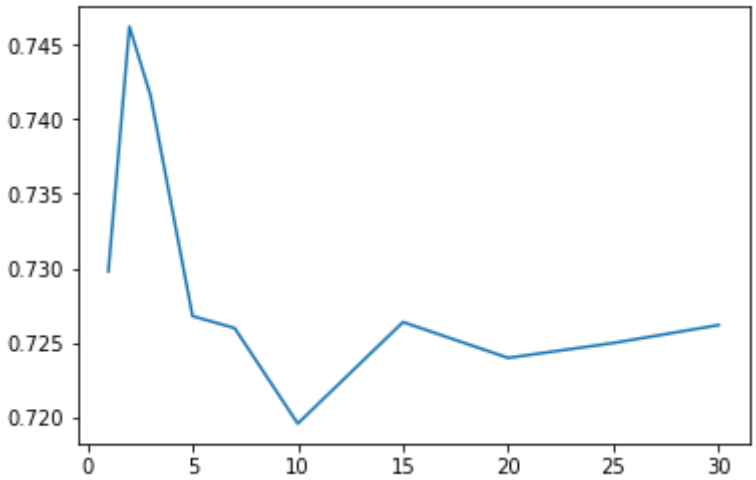
First, we will create the training and validation folds for use in k-fold cross validation.

In [166]:
```
# Create the dataset folds for cross-valdiation.
num_folds = 5
foldSize = y_train.shape[0]/num_folds
X_train_folds = []
y_train_folds =  []
indices = np.arange(X_train.shape[0])
np.random.shuffle(indices)
# ================================================================ #
# YOUR CODE HERE:
#   Split the training data into num_folds (i.e., 5) folds.
#   X_train_folds is a list, where X_train_folds[i] contains the
#      data points in fold i.
#   y_train_folds is also a list, where y_train_folds[i] contains
#      the corresponding labels for the data in X_train_folds[i]
# ================================================================ #
for i in np.arange(num_folds):
    X_train_folds.append(X_train[indices[foldSize*i: min((foldSize*(i+1
)),len(X_train))]])
    y_train_folds.append(y_train[indices[foldSize*i: min((foldSize*(i+1
)),len(y_train))]])
pass

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

## Optimizing the number of nearest neighbors hyperparameter.

In this section, we select different numbers of nearest neighbors and assess which one has the lowest k-fold cross validation error.

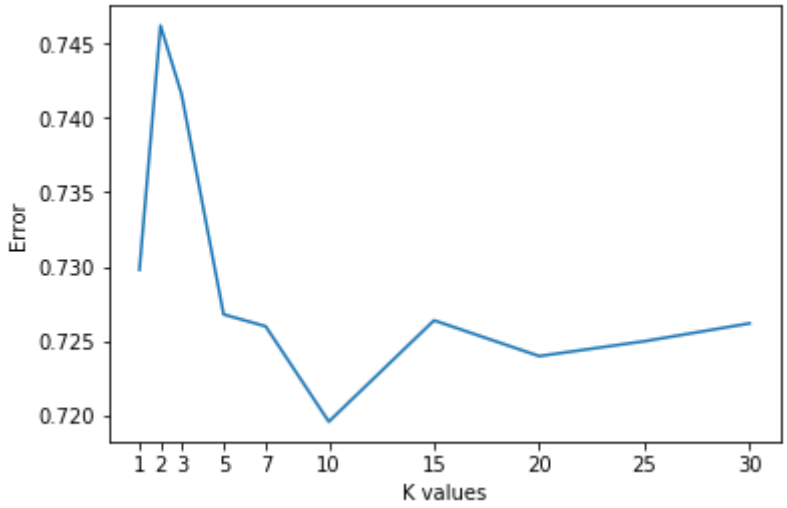
In [173]:
```
time_start =time.time()

ks = [1, 2, 3, 5, 7, 10, 15, 20, 25, 30]
errorAvg = []
# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each k in ks, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of k vs. cross-validation error. Since
#   we are assuming L2 distance here, please use the vectorized code!
#   Otherwise, you might be waiting a long time.
# ================================================================ #
for kval in ks:
    error = []
    for i in np.arange(num_folds):
        yTraining = np.concatenate(map(lambda x: y_train_folds[x], np.se
tdiff1d(np.arange(num_folds), i)))
        XTraining = np.concatenate(map(lambda x: X_train_folds[x], np.se
tdiff1d(np.arange(num_folds), i)))
        Xtest = X_train_folds[i]
        Ytest = y_train_folds[i]
        knn.train(X=XTraining, y=yTraining)
        dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=Xtes
t)
        yPredicted = knn.predict_labels(dists=dists_L2_vectorized, k=kva
l)
        error.append((np.count_nonzero(Ytest-yPredicted))/float(len(Ytes
t)))

    errorAvg.append(np.mean(error, axis = 0))
plt.plot(ks, errorAvg)
plt.show()
pass
kBest = ks[np.argsort(errorAvg)[0]]
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print('Computation time: %.2f'%(time.time()-time_start))
```




```
Computation time: 29.37
```

In [190]:
```
plt.plot(ks, errorAvg)
plt.xticks(ks)
plt.xlabel("K values")
plt.ylabel("Error")
print"Best value of K is", ks[np.argsort(errorAvg)[0]]
print"Error for best K is", errorAvg[np.argsort(errorAvg)[0]]
kBest = ks[np.argsort(errorAvg)[0]]
```

```
Best value of K is 10
Error for best K is 0.7196
```




# Questions:

(1) What value of $k$ is best amongst the tested $k$'s?

(2) What is the cross-validation error for this value of $k$?

# Answers:

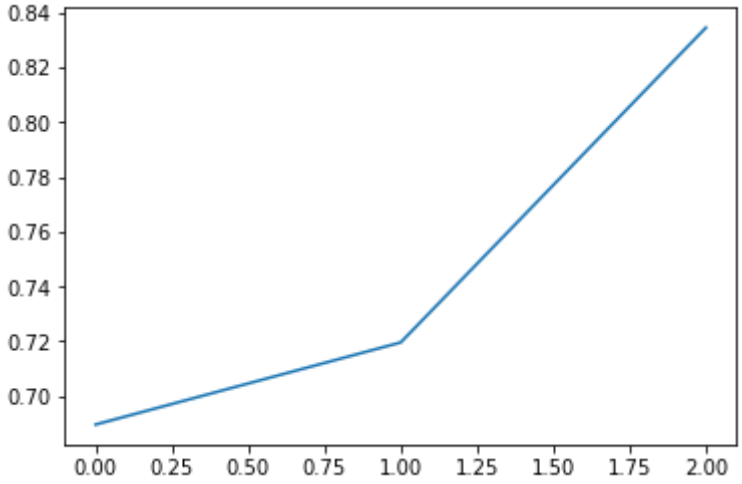
(1) The best k = 10

(2) error for k = 10 is 0.7196

### Optimizing the norm

Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

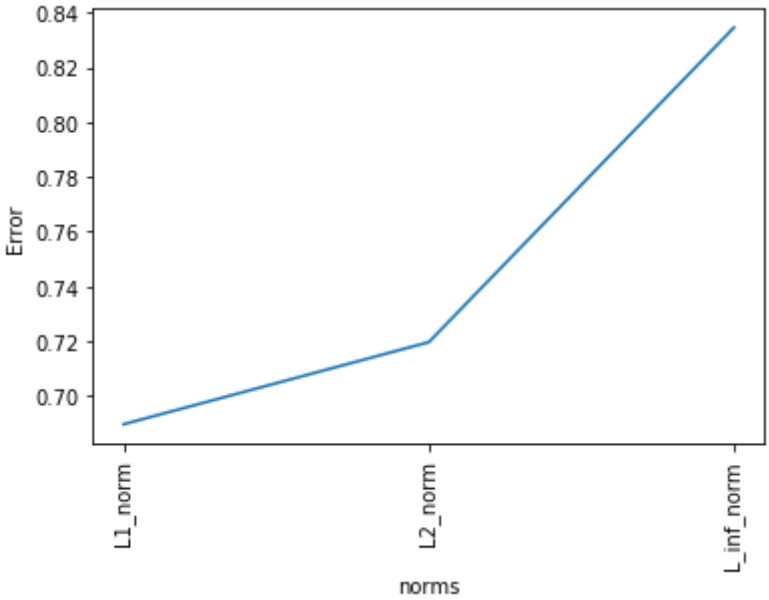
In [176]:
```
time_start =time.time()

L1_norm = lambda x: np.linalg.norm(x, ord=1)
L2_norm = lambda x: np.linalg.norm(x, ord=2)
Linf_norm = lambda x: np.linalg.norm(x, ord= np.inf)
norms = [L1_norm, L2_norm, Linf_norm]

errorNormAvg = []
# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each norm in norms, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of the norm used vs the cross-validation er
ror
#   Use the best cross-validation k from the previous part.
#
#   Feel free to use the compute_distances function.  We're testing just
#   three norms, but be advised that this could still take some time.
#   You're welcome to write a vectorized form of the L1- and Linf- norms
#   to speed this up, but it is not necessary.
# ================================================================ #
for normVal in norms:
    errorNorm = []
    for i in np.arange(num_folds):
        yTraining = np.concatenate(map(lambda x: y_train_folds[x], np.se
tdiff1d(np.arange(num_folds), i)))
        XTraining = np.concatenate(map(lambda x: X_train_folds[x], np.se
tdiff1d(np.arange(num_folds), i)))
        Xtest = X_train_folds[i]
        Ytest = y_train_folds[i]
        knn.train(X=XTraining, y=yTraining)
        distance = knn.compute_distances(X=Xtest, norm=normVal)
        yPredicted = knn.predict_labels(dists=distance, k=kBest)
        errorNorm.append(np.count_nonzero(Ytest-yPredicted)/float(len(Yt
est)))


    errorNormAvg.append(np.mean(errorNorm))


plt.plot(range(3), errorNormAvg)
plt.show()

pass

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
print('Computation time: %.2f'%(time.time()-time_start))
```




```
Computation time: 853.32
```

In [193]:
```
normStr = ["L1_norm","L2_norm", "L_infnorm"]
plt.plot(range(3), errorNormAvg)
plt.xticks(range(3), ("L1_norm","L2_norm", "L_inf_norm"), rotation=90)
plt.xlabel('norms')
plt.ylabel('Error')
#plt.show()
#pass
normBest = normStr[np.argsort(errorNormAvg)[0]]
print"Best norm is", normBest
print "Error for", normBest, "is: ", errorNormAvg[np.argsort(errorNormAv
g)[0]]
```

```
Best norm is L1_norm
Error for L1_norm is:  0.6895999999999999
```



## Questions:

(1) What norm has the best cross-validation error?

(2) What is the cross-validation error for your given norm and k?

## Answers:

(1) L1 norm

(2) 0.6895999999999999

# Evaluating the model on the testing dataset.

Now, given the optimal $k$ and norm you found in earlier parts, evaluate the testing error of the k-nearest neighbors model.

In [187]:
```
errorFinal = 1

# ================================================================ #
# YOUR CODE HERE:
#   Evaluate the testing error of the k-nearest neighbors classifier
#   for your optimal hyperparameters found by 5-fold cross-validation.
# ================================================================ #
knn.train(X=X_train, y=y_train)
distance = knn.compute_distances(X=X_test, norm= norms[0])
yPredicted = knn.predict_labels(dists=distance, k=kBest)
errorFinal = np.count_nonzero(y_test-yPredicted)/float(len(y_test))

pass

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print('Error rate achieved: {}'.format(errorFinal))
```

```
Error rate achieved: 0.712
```

## Question:

How much did your error improve by cross-validation over naively choosing $k = 1$ and using the L2-norm?

## Answer:

The new error is 0.712. The error when we naively chose k=1 and L2 norm was 0.726. So the error rate improved by 1.9%.

# This is the svm workbook for ECE 239AS Assignment #2

Please follow the notebook linearly to implement a linear support vector machine.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and includes code to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training an SVM classifier via gradient descent.

# Importing libraries and data setup

In [1]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.data_utils import load_CIFAR10 # function to load the CIFAR-10 dataset.
import pdb

# Load matplotlib images inline
%matplotlib inline

# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [2]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = 'cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
('Training data shape: ', (50000, 32, 32, 3))
('Training labels shape: ', (50000,))
('Test data shape: ', (10000, 32, 32, 3))
('Test labels shape: ', (10000,))
```

In [3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse'
, 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

In [4]:
```
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# We use the first num_test points of the original test set as our
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('Dev data shape: ', X_dev.shape)
print('Dev labels shape: ', y_dev.shape)
```

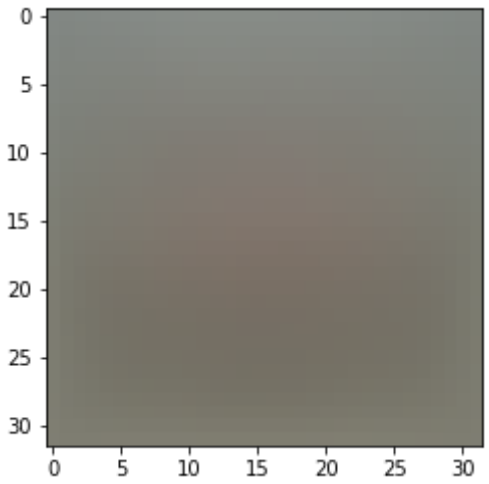
```
('Train data shape: ', (49000, 32, 32, 3))
('Train labels shape: ', (49000,))
('Validation data shape: ', (1000, 32, 32, 3))
('Validation labels shape: ', (1000,))
('Test data shape: ', (1000, 32, 32, 3))
('Test labels shape: ', (1000,))
('Dev data shape: ', (500, 32, 32, 3))
('Dev labels shape: ', (500,))
```

In [5]:
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# As a sanity check, print out the shapes of the data
print('Training data shape: ', X_train.shape)
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)
```

```
('Training data shape: ', (49000, 3072))
('Validation data shape: ', (1000, 3072))
('Test data shape: ', (1000, 3072))
('dev data shape: ', (500, 3072))
```

In [6]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize th
e mean image
plt.show()
```

```
[130.64189796 135.98173469 132.47391837 130.05569388 135.34804082
 131.75402041 130.96055102 136.14328571 132.47636735 131.48467347]
```




In [7]:
```
# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image
```

In [8]:
```
# third: append the bias dimension of ones (i.e. bias trick) so that our
 SVM
# only has to worry about optimizing a single weight matrix W.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

```
((49000, 3073), (1000, 3073), (1000, 3073), (500, 3073))
```

# Question:

(1) For the SVM, we perform mean-subtraction on the data. However, for the KNN notebook, we did not. Why?

# Answer:

(1) If we do a mean subtraction in K nearest neighbours, all data points move by fixed distance. This does not make any difference because the relative distance between two points remains same. Therefore doing mean subtraction in k-nearest neighbours is not required.

# Training an SVM

The following cells will take you through building an SVM. You will implement its loss function, then subsequently train it with gradient descent. Finally, you will choose the learning rate of gradient descent to optimize its classification performance.

In [10]:
```
from nndl.svm import SVM
```

In [13]:
```
# Declare an instance of the SVM class.
# Weights are initialized to a random value.
# Note, to keep people's initial solutions consistent, we are going to u
se a random seed.

np.random.seed(1)

num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]

svm = SVM(dims=[num_classes, num_features])
```

#### SVM loss

In [19]:
```
## Implement the loss function for in the SVM class(nndl/svm.py), svm.lo
ss()

loss = svm.loss(X_train, y_train)
print('The training set loss is {}.'.format(loss))

# If you implemented the loss correctly, it should be 15569.98
```

```
The training set loss is 15569.9779154.
```

#### SVM gradient

In [30]:
```
## Calculate the gradient of the SVM class.
# For convenience, we'll write one function that computes the loss
#   and gradient together. Please modify svm.loss_and_grad(X, y).
# You may copy and paste your loss code from svm.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = svm.loss_and_grad(X_dev,y_dev)

# Compare your gradient to a numerical gradient check.
# You should see relative gradient errors on the order of 1e-07 or less
 if you implemented the gradient correctly.
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 23.825277 analytic: 23.825278, relative error: 5.379596e-09
numerical: -16.837826 analytic: -16.837826, relative error: 3.472920e-0
9
numerical: 0.249854 analytic: 0.249854, relative error: 1.157371e-06
numerical: -24.505644 analytic: -24.505643, relative error: 2.344058e-0
8
numerical: 33.166927 analytic: 33.166927, relative error: 7.718880e-09
numerical: 7.296520 analytic: 7.296520, relative error: 7.335509e-09
numerical: -4.427121 analytic: -4.427121, relative error: 5.476629e-08
numerical: -4.994335 analytic: -4.994336, relative error: 2.426407e-08
numerical: 15.935519 analytic: 15.935520, relative error: 6.877312e-09
numerical: -2.728510 analytic: -2.728510, relative error: 1.598205e-08
```

# A vectorized version of SVM

To speed things up, we will vectorize the loss and gradient calculations. This will be helpful for stochastic gradient descent.

In [31]:
```
import time
```

In [44]:
```
## Implement svm.fast_loss_and_grad which calculates the loss and gradie
nt
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = svm.loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np
.linalg.norm(grad, 'fro'), toc - tic))

tic = time.time()
loss_vectorized, grad_vectorized = svm.fast_loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vect
orized, np.linalg.norm(grad_vectorized, 'fro'), toc - tic))

# The losses should match but your vectorized implementation should be m
uch faster.
print('difference in loss / grad: {} / {}'.format(loss - loss_vectorized
, np.linalg.norm(grad - grad_vectorized)))

# You should notice a speedup with the same output, i.e., differences on
 the order of 1e-12
```

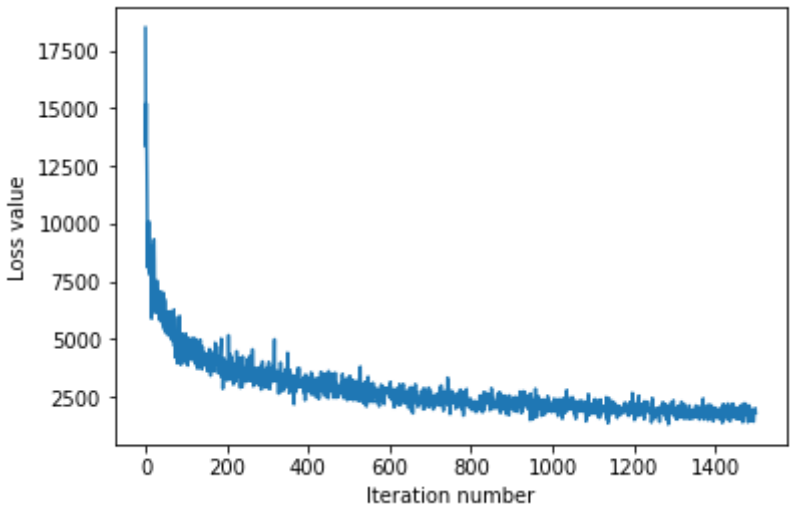
```
Normal loss / grad_norm: 14633.277372 / 2162.85338633 computed in 0.092
8139686584s
(500, 10)
(500, 10)
Vectorized loss / grad: 14633.277372 / 2162.85338633 computed in 0.0066
2708282471s
difference in loss / grad: -1.45519152284e-11 / 3.39052620374e-12
```

# Stochastic gradient descent

We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

In [48]:
```
# Implement svm.train() by filling in the code to extract a batch of dat
a
# and perform the gradient step.

tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=5e-4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took {}s'.format(toc - tic))

plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 18496.8278756
iteration 100 / 1500: loss 4052.62706482
iteration 200 / 1500: loss 3211.4313189
iteration 300 / 1500: loss 3046.11768077
iteration 400 / 1500: loss 2722.6683829
iteration 500 / 1500: loss 3326.00280768
iteration 600 / 1500: loss 2863.99542696
iteration 700 / 1500: loss 2704.54070908
iteration 800 / 1500: loss 2201.04568549
iteration 900 / 1500: loss 2161.61630076
iteration 1000 / 1500: loss 1904.52478287
iteration 1100 / 1500: loss 1905.85202046
iteration 1200 / 1500: loss 2240.88969846
iteration 1300 / 1500: loss 1647.14975863
iteration 1400 / 1500: loss 1936.11806722
That took 3.72931909561s
```




## Evaluate the performance of the trained SVM on the validation data.

In [49]:
```
## Implement svm.predict() and use it to compute the training and testin
g error.

y_train_pred = svm.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pr
ed), )))
y_val_pred = svm.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pre
d)), ))
```

```
training accuracy: 0.303
validation accuracy: 0.306
```

## Optimize the SVM

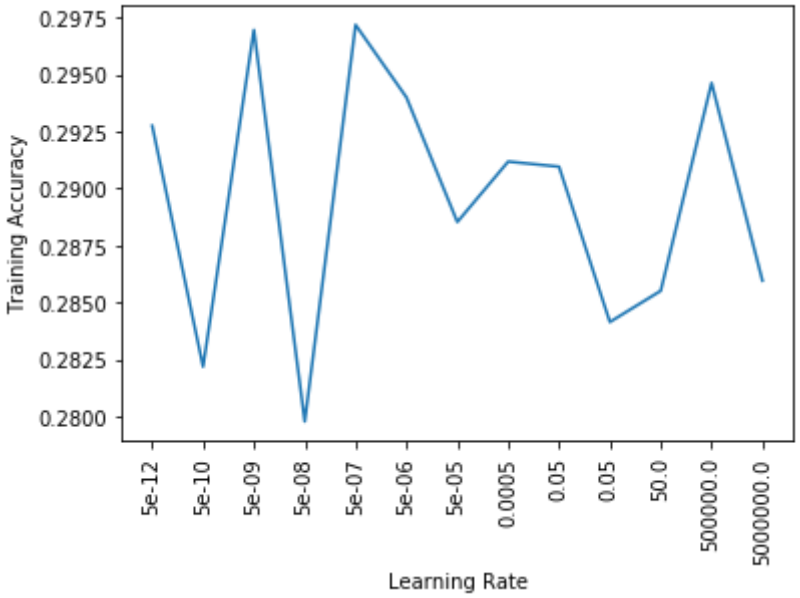
Note, to make things faster and simpler, we won't do k-fold cross-validation, but will only optimize the hyperparameters on the validation dataset (X_val, y_val).

In [80]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Train the SVM with different learning rates and evaluate on the
#     validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best VALIDATION accuracy corresponding to the best VALIDATIO
N error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
#   Note: You do not need to modify SVM class for this section
# ================================================================ #

# ================================================================ #
# END YOUR CODE HERE

learningRates = [5e-12, 5e-10, 5e-9, 5e-8, 5e-7, 5e-6, 5e-5, 5e-4, 5e-2,
 5e-2, 5e1, 5e5, 5e6]
trainingAccuracy = []
validationAccuracy = []
for learningRate in learningRates:
    #print "-"*40, "\n"
    #print " learning rate is:", learningRate, "\n"

    loss_hist = svm.train(X_train, y_train, learning_rate=5e-4,
                      num_iters=1500, verbose=False)
    y_train_pred = svm.predict(X_train)
    trainingAccuracy.append(np.mean(np.equal(y_train,y_train_pred)))
    #print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_tra
in_pred), )))
    y_val_pred = svm.predict(X_val)
    validationAccuracy.append(np.mean(np.equal(y_val, y_val_pred)))
    #print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_va
l_pred)), ))
    #print "-"*40, "\n"
# ================================================================ #
bestvalidation = validationAccuracy[np.argsort(validationAccuracy)[-1]]
bestLearningRate = learningRates[np.argsort(validationAccuracy)[-1]]
print "Best Validation accuracy: ", bestvalidation
print "Corresponding learning rate: ", bestLearningRate
```

```
Best Validation accuracy:  0.309
Corresponding learning rate:  0.0005
```

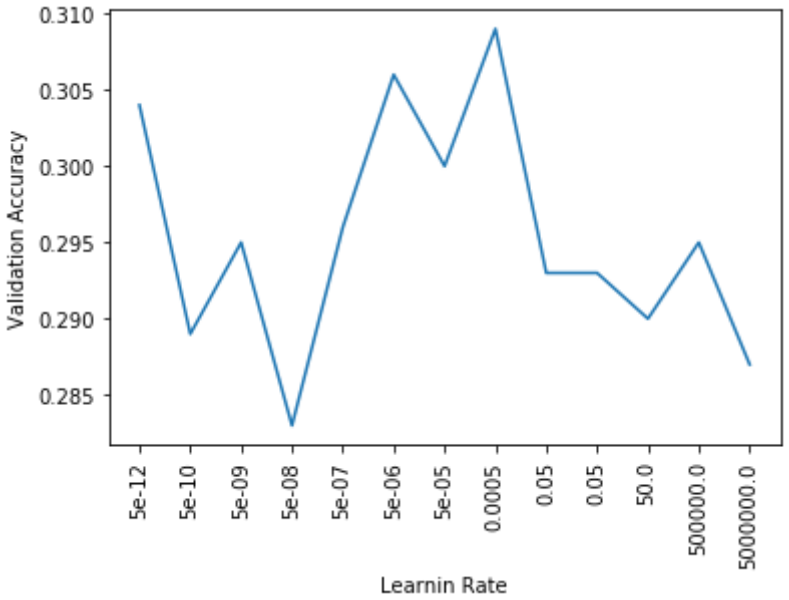
In [81]:
```
plt.plot(range(len(learningRates)), trainingAccuracy)
plt.xticks(range(len(learningRates)), learningRates, rotation =90)
plt.xlabel('Learning Rate')
plt.ylabel('Training Accuracy')
```

Out[81]: <matplotlib.text.Text at 0x118a23610>




In [83]:
```
plt.plot(range(len(learningRates)), validationAccuracy)
plt.xticks(range(len(learningRates)), learningRates, rotation =90)
plt.xlabel('Learnin Rate')
plt.ylabel('Validation Accuracy')
```

Out[83]: <matplotlib.text.Text at 0x117465050>



In [84]:
```
loss_hist = svm.train(X_train, y_train, learning_rate=bestLearningRate,
                      num_iters=1500, verbose=False)

y_train_pred = svm.predict(X_train)
y_test_pred = svm.predict(X_test)
testingAccuracy = np.mean(np.equal(y_test,y_test_pred))
print "Testing accuracy is: ", testingAccuracy
print "Testing Error is: ", 1-testingAccuracy
```

```
Testing accuracy is:  0.273
Testing Error is:  0.727
```

# This is the softmax workbook for ECE 239AS Assignment #2

Please follow the notebook linearly to implement a softmax classifier.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training a softmax classifier.

In [175]:
```
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [176]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1
000, num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to pre
pare
    it for the linear classifier. These are the same steps as we used fo
r the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # Preprocessing: reshape the image data into rows
```

```
    X_train = np.reshape(X_train, (X_train.shape[0], -1))
    X_val = np.reshape(X_val, (X_val.shape[0], -1))
    X_test = np.reshape(X_test, (X_test.shape[0], -1))
    X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_CIFAR
10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
('Train data shape: ', (49000, 3073))
('Train labels shape: ', (49000,))
('Validation data shape: ', (1000, 3073))
('Validation labels shape: ', (1000,))
('Test data shape: ', (1000, 3073))
('Test labels shape: ', (1000,))
('dev data shape: ', (500, 3073))
('dev labels shape: ', (500,))
```

In [177]:
```
aY = [2.1, 0.2, 2.3]
y = [0, 1, 2]
logSum = np.log(np.sum(np.exp(aY)))
classProb = aY[y[0]]
print (logSum)
```

```
2.963299735050387
```

# Training a softmax classifier.

The following cells will take you through building a softmax classifier. You will implement its loss function, then subsequently train it with gradient descent. Finally, you will choose the learning rate of gradient descent to optimize its classification performance.

In [178]:
```
from nndl import Softmax
```

In [179]:
```
# Declare an instance of the Softmax class.
# Weights are initialized to a random value.
# Note, to keep people's first solutions consistent, we are going to use
 a random seed.

np.random.seed(1)

num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]

softmax = Softmax(dims=[num_classes, num_features])
```

#### Softmax loss

In [180]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

loss = softmax.loss(X_train, y_train)
```

In [181]:
```
print(loss)
```

```
2.3277607028048966
```

# Question:

You'll notice the loss returned by the softmax is about 2.3 (if implemented correctly). Why does this value make sense?

# Answer:

The loss is high and it makes sense because the weight matrix is randomly chosen. This is equal to log(num_classes). num of classes = 10. log10 = 2.3. This is expected as we do sum(log(y=j)).

#### Softmax gradient

In [182]:
```
## Calculate the gradient of the softmax loss in the Softmax class.
# For convenience, we'll write one function that computes the loss
#   and gradient together, softmax.loss_and_grad(X, y)
# You may copy and paste your loss code from softmax.loss() here, and th
en
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = softmax.loss_and_grad(X_dev,y_dev)

# Compare your gradient to a gradient check we wrote.
# You should see relative gradient errors on the order of 1e-07 or less
 if you implemented the gradient correctly.
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -0.877588 analytic: -0.877588, relative error: 7.639372e-10
numerical: 1.053863 analytic: 1.053863, relative error: 6.004968e-08
numerical: 0.190154 analytic: 0.190154, relative error: 8.059006e-08
numerical: 2.096415 analytic: 2.096415, relative error: 1.085186e-08
numerical: 0.233771 analytic: 0.233771, relative error: 1.745935e-07
numerical: 1.220637 analytic: 1.220636, relative error: 2.151283e-08
numerical: -1.785057 analytic: -1.785057, relative error: 3.253666e-08
numerical: -1.569662 analytic: -1.569662, relative error: 2.301984e-08
numerical: 0.516896 analytic: 0.516896, relative error: 2.821143e-08
numerical: -1.477842 analytic: -1.477842, relative error: 5.513787e-08
```

# A vectorized version of Softmax

To speed things up, we will vectorize the loss and gradient calculations. This will be helpful for stochastic gradient descent.

In [183]:
```
import time
```

In [184]:
```
sum = 0
loss = 0.0
dims=[10, 3073]
W = np.random.normal(size=dims) * 0.0001
aY = np.dot(X_train, np.transpose(W))


logSum = np.log(np.sum(np.exp(aY), axis = 1))
sum1 = -aY[np.arange(X_train.shape[0]), y_train]
loss = np.sum(sum1+logSum)/X_train.shape[0]
print sum1.shape
```

```
(49000,)
```

In [185]:
```
## Implement softmax.fast_loss_and_grad which calculates the loss and gr
adient
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = softmax.loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np
.linalg.norm(grad, 'fro'), toc - tic))

tic = time.time()
loss_vectorized, grad_vectorized = softmax.fast_loss_and_grad(X_dev, y_d
ev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vect
orized, np.linalg.norm(grad_vectorized, 'fro'), toc - tic))

# The losses should match but your vectorized implementation should be m
uch faster.
print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized
, np.linalg.norm(grad - grad_vectorized)))

# You should notice a speedup with the same output.
```

```
Normal loss / grad_norm: 2.32036584895 / 329.871439636 computed in 0.23
1281042099s
Vectorized loss / grad: 2.32036584895 / 329.871439636 computed in 0.006
36005401611s
difference in loss / grad: 0.0 /2.11434026275e-13
```

## Stochastic gradient descent

We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

## Question:

How should the softmax gradient descent training step differ from the svm training step, if at all?

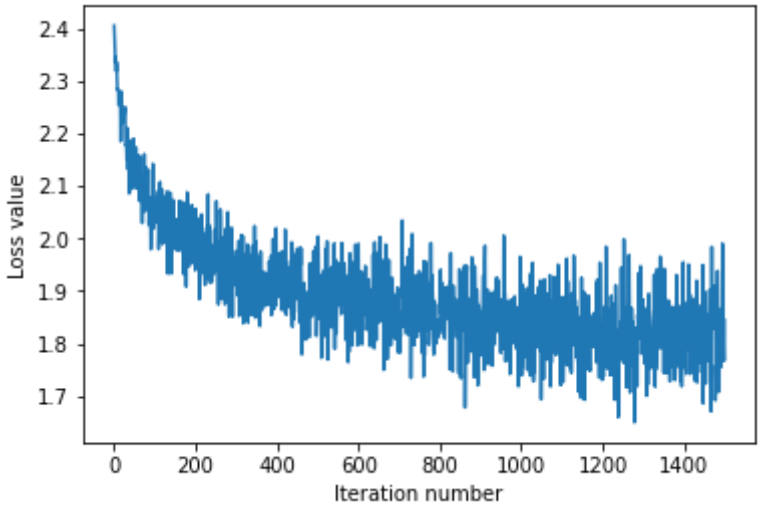
## Answer:

The SVM interprets computed scores as class scores and its loss function encourages the correct class to have a score higher by a margin than the other class scores. The Softmax classifier instead interprets the scores as (unnormalized) log probabilities for each class and then encourages the (normalized) log probability of the correct class to be high.

In SVM cost function is loss scores and in Softmax it is log Probabilities. So in SVM gradient, we reduce the loss and in softmax we increase the log probability.

In [186]:
```
# Implement softmax.train() by filling in the code to extract a batch of
 data
# and perform the gradient step.
import time


tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took {}s'.format(toc - tic))

plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 2.40500420842
iteration 100 / 1500: loss 2.07570101
iteration 200 / 1500: loss 2.05672734652
iteration 300 / 1500: loss 1.97651770031
iteration 400 / 1500: loss 1.9063129596
iteration 500 / 1500: loss 1.91017064455
iteration 600 / 1500: loss 1.98994833994
iteration 700 / 1500: loss 1.82911919946
iteration 800 / 1500: loss 1.83537656104
iteration 900 / 1500: loss 1.83746196393
iteration 1000 / 1500: loss 1.78263530483
iteration 1100 / 1500: loss 1.8246265651
iteration 1200 / 1500: loss 1.83692948919
iteration 1300 / 1500: loss 1.82663301352
iteration 1400 / 1500: loss 1.79349164587
That took 3.10122203827s
```




## Evaluate the performance of the trained softmax classifier on the validation data.

In [203]:
```
## Implement softmax.predict() and use it to compute the training and te
sting error.

y_train_pred = softmax.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pr
ed), )))
y_val_pred = softmax.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pre
d)), ))
```

```
training accuracy: 0.370326530612
validation accuracy: 0.372
```

## Optimize the softmax classifier

You may copy and paste your optimization code from the SVM here.

In [215]:
```
np.finfo(float).eps
```

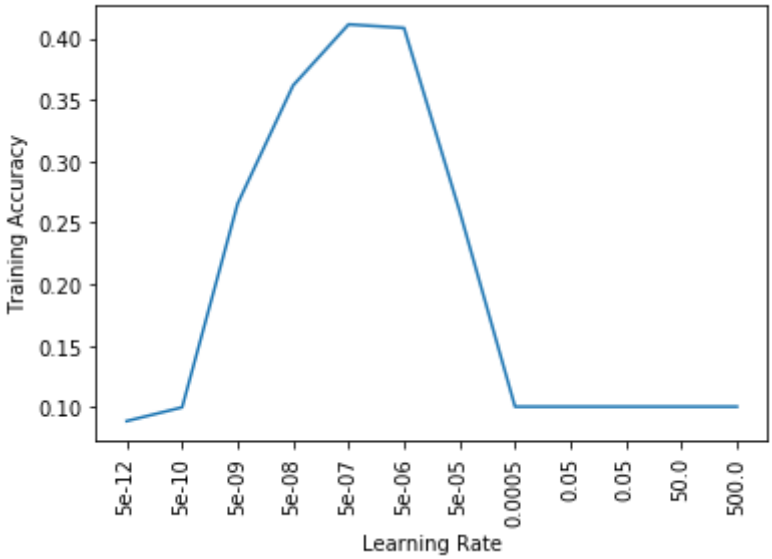
Out[215]:
```
2.220446049250313e-16
```

In [216]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Train the Softmax classifier with different learning rates and
#     evaluate on the validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best validation accuracy corresponding to the best validatio
n error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
learningRates = [5e-12, 5e-10, 5e-9, 5e-8, 5e-7, 5e-6, 5e-5, 5e-4, 5e-2,
 5e-2, 5e1, 5e2]
trainingAccuracy = []
validationAccuracy = []
for learningRate in learningRates:
    #print "-"*40, "\n"
    #print " learning rate is:", learningRate, "\n"

    loss_hist = softmax.train(X_train, y_train, learning_rate=learningRa
te,
                      num_iters=1500, verbose=False)
    y_train_pred = softmax.predict(X_train)
    trainingAccuracy.append(np.mean(np.equal(y_train,y_train_pred)))
    #print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_tra
in_pred), )))
    y_val_pred = softmax.predict(X_val)
    validationAccuracy.append(np.mean(np.equal(y_val, y_val_pred)))
    #print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_va
l_pred)), ))

# ================================================================ #
bestvalidation = validationAccuracy[np.argsort(validationAccuracy)[-1]]
bestLearningRate = learningRates[np.argsort(validationAccuracy)[-1]]
print "Best Validation accuracy: ", bestvalidation
print "Corresponding learning rate: ", bestLearningRate
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Best Validation accuracy:  0.408
Corresponding learning rate:  5e-07
```

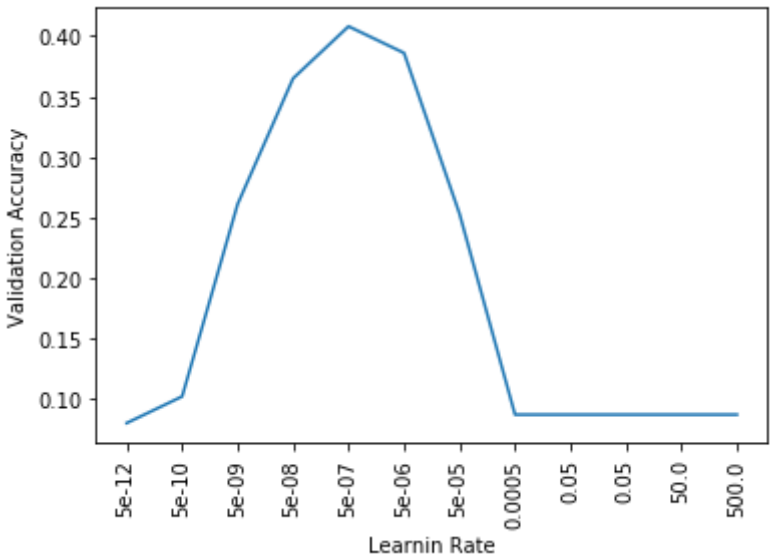
In [217]:
```
plt.plot(range(len(learningRates)), trainingAccuracy)
plt.xticks(range(len(learningRates)), learningRates, rotation =90)
plt.xlabel('Learning Rate')
plt.ylabel('Training Accuracy')
```

Out[217]: <matplotlib.text.Text at 0x110de5410>




In [218]:
```
plt.plot(range(len(learningRates)), validationAccuracy)
plt.xticks(range(len(learningRates)), learningRates, rotation =90)
plt.xlabel('Learnin Rate')
plt.ylabel('Validation Accuracy')
```

Out[218]: <matplotlib.text.Text at 0x110df0910>



```python
import numpy as np
import pdb

"""
This code was based off of code from cs231n at Stanford University, and modified for ece239as at UCLA.
"""

class KNN(object):

  def __init__(self):
    pass

  def train(self, X, y):

    self.X_train = X
    self.y_train = y

  def compute_distances(self, X, norm=None):

    if norm is None:
      norm = lambda x: np.sqrt(np.sum(x**2))
      #norm = 2

    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in np.arange(num_test):

      for j in np.arange(num_train):
        # ================================================================ #
        # YOUR CODE HERE:
        #   Compute the distance between the ith test point and the jth
        #   training point using norm(), and store the result in dists[i, j].
        # ================================================================ #
        dists[i,j] = norm(X[i] - self.X_train[j])
        pass

        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

    return dists

  def compute_L2_distances_vectorized(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train WITHOUT using any for loops.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))

    # ================================================================ #
    # YOUR CODE HERE:
    #   Compute the L2 distance between the ith test point and the jth
    #   training point and store the result in dists[i, j].  You may
    #    NOT use a for loop (or list comprehension).  You may only use
    #     numpy operations.
    #
    #     HINT: use broadcasting.  If you have a shape (N,1) array and
    #   a shape (M,) array, adding them together produces a shape (N, M)
    #   array.
    # ================================================================ #
    dists = np.sqrt(((X**2).sum(axis=1,keepdims= True ))+ (self.X_train**2).sum(axis=1) - 2* X.dot(self.X_train.T))

    #sum(||X-X_train||^2) can be written as above keeping in mind matrix multiplication dimensionality and broadcasting rules.
    pass

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return dists


  def predict_labels(self, dists, k=1):
    """
    Given a matrix of distances between test points and training points,
```

```
    predict a label for each test point.

    Inputs:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      gives the distance betwen the ith test point and the jth training point.

    Returns:
    - y: A numpy array of shape (num_test,) containing predicted labels for the
      test data, where y[i] is the predicted label for the test point X[i].
    """
    num_test = dists.shape[0]
    y_pred = np.zeros(num_test)
    for i in np.arange(num_test):
      # A list of length k storing the labels of the k nearest neighbors to
      # the ith test point.
      closest_y = []
      # ========================================================================= #
      # YOUR CODE HERE:
      #   Use the distances to calculate and then store the labels of
      #   the k-nearest neighbors to the ith test point.  The function
      #   numpy.argsort may be useful.
      #
      #   After doing this, find the most common label of the k-nearest
      #   neighbors.  Store the predicted label of the ith training example
      #   as y_pred[i].  Break ties by choosing the smaller label.
      # ========================================================================= #
      closest_y = list(self.y_train[np.argsort(dists[i])[:k]])
      y_pred[i] = max(set(closest_y), key = closest_y.count)

      # ========================================================================= #
      # END YOUR CODE HERE
      # ========================================================================= #

    return y_pred
```

```python
import numpy as np
import pdb

"""
This code was based off of code from cs231n at Stanford University, and modified for ece239as at UCLA.
"""
class SVM(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

  def init_weights(self, dims):
    """
    Initializes the weight matrix of the SVM.  Note that it has shape (C, D)
    where C is the number of classes and D is the feature size.
    """
    self.W = np.random.normal(size=dims)

  def loss(self, X, y):
    """
    Calculates the SVM loss.

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.

    Returns a tuple of:
    - loss as single float
    """

    # compute the loss and the gradient
    num_classes = self.W.shape[0]
    num_train = X.shape[0]
    loss = 0.0
    a_mat = np.dot(X, np.transpose(self.W))
    for i in np.arange(num_train):
    # ============================================================= #
    # YOUR CODE HERE:
    #   Calculate the normalized SVM loss, and store it as 'loss'.
    #   (That is, calculate the sum of the losses of all the training
    #   set margins, and then normalize the loss by the number of
    #   training examples.)
    # ============================================================= #

      for j in range(num_classes):
        if(j != y[i]):
          ajx = a_mat[i,j]
          ayx = a_mat[i, y[i]]
          loss +=np.maximum(0, 1+ajx-ayx)
    # ============================================================= #
    # END YOUR CODE HERE
    # ============================================================= #
    loss = loss/num_train
    return loss

  def loss_and_grad(self, X, y):
    """
  Same as self.loss(X, y), except that it also returns the gradient.

  Output: grad -- a matrix of the same dimensions as W containing
    the gradient of the loss with respect to W.
  """

    # compute the loss and the gradient
    num_classes = self.W.shape[0]
    num_train = X.shape[0]
    loss = 0.0
    grad = np.zeros_like(self.W)
```

```
    a_mat = np.dot(X, np.transpose(self.W))
    for i in np.arange(num_train):
    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the SVM loss and the gradient.  Store the gradient in
    #   the variable grad.
    # ================================================================ #
      for j in range(num_classes):
        if(j != y[i]):
          ajx = a_mat[i,j]
          ayx = a_mat[i, y[i]]
          zj = 1+ajx-ayx
          loss += np.maximum(0, zj)
          grad[j] +=  (zj > 0) * X[i]
          grad[y[i]] -=  (zj > 0) * X[i]

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    loss /= num_train
    grad /= num_train

    return loss, grad

  def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
    """
    sample a few random elements and only return numerical
    in these dimensions.
    """

    for i in np.arange(num_checks):
      ix = tuple([np.random.randint(m) for m in self.W.shape])

      oldval = self.W[ix]
      self.W[ix] = oldval + h # increment by h
      fxph = self.loss(X, y)
      self.W[ix] = oldval - h # decrement by h
      fxmh = self.loss(X,y) # evaluate f(x - h)
      self.W[ix] = oldval # reset

      grad_numerical = (fxph - fxmh) / (2 * h)
      grad_analytic = your_grad[ix]
      rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) + abs(grad_analytic))
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic, rel_error))

  def fast_loss_and_grad(self, X, y):
    """
    A vectorized implementation of loss_and_grad. It shares the same
  inputs and ouptuts as loss_and_grad.
    """
    loss = 0.0
    grad = np.zeros(self.W.shape) # initialize the gradient as zero

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the SVM loss WITHOUT any for loops.
    # ================================================================ #
    a_mat = np.dot(X, np.transpose(self.W))
    ayx = a_mat[np.arange(X.shape[0]), y]

    zj = 1+ a_mat - np.matrix(ayx).T
    zj[np.arange(X.shape[0]), y] -= 1

    loss += np.maximum(0, zj)

    loss = np.sum(loss)/X.shape[0]
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
```

```
    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the SVM grad WITHOUT any for loops.
    # ================================================================ #
    ind_mat = np.zeros_like(zj)
    ind_mat[zj>0] = 1
    sum1 = np.sum(ind_mat, axis = 1)
    ind_mat[np.arange(X.shape[0]), y] = -sum1.T
    grad = np.matmul(ind_mat.T, X)
    grad = grad/X.shape[0]
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return loss, grad

  def train(self, X, y, learning_rate=1e-3, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this linear classifier using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) containing training data; there are N
      training samples each of dimension D.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c
      means that X[i] has label 0 <= c < C for C classes.
    - learning_rate: (float) learning rate for optimization.
    - num_iters: (integer) number of steps to take when optimizing
    - batch_size: (integer) number of training examples to use at each step.
    - verbose: (boolean) If true, print progress during optimization.

    Outputs:
    A list containing the value of the loss function at each training iteration.
    """
    num_train, dim = X.shape
    num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of classes

    self.init_weights(dims=[np.max(y) + 1, X.shape[1]])  # initializes the weights of self.W

    # Run stochastic gradient descent to optimize W
    loss_history = []

    for it in np.arange(num_iters):
      X_batch = None
      y_batch = None

      # ================================================================ #
      # YOUR CODE HERE:
      #   Sample batch_size elements from the training data for use in
      #   gradient descent.  After sampling,
      #     - X_batch should have shape: (dim, batch_size)
      #     - y_batch should have shape: (batch_size,)
      #   The indices should be randomly generated to reduce correlations
      #   in the dataset.  Use np.random.choice.  It's okay to sample with
      #   replacement.
      # ================================================================ #
      indic = np.random.choice(num_train, batch_size)
      X_batch = X[indic,:]
      y_batch = y[indic]
      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #

      # evaluate loss and gradient
      loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
      loss_history.append(loss)

      # ================================================================ #
      # YOUR CODE HERE:
      #   Update the parameters, self.W, with a gradient step
      # ================================================================ #
      self.W -= learning_rate*grad
      # ================================================================ #
```

```
219       # END YOUR CODE HERE
220       # ================================================================ #
221
222       if verbose and it % 100 == 0:
223         print('iteration {} / {}: loss {}'.format(it, num_iters, loss))
224
225     return loss_history
226
227   def predict(self, X):
228     """
229     Inputs:
230     - X: N x D array of training data. Each row is a D-dimensional point.
231
232     Returns:
233     - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
234       array of length N, and each element is an integer giving the predicted
235       class.
236     """
237     y_pred = np.zeros(X.shape[0])
238
239
240     # ================================================================ #
241     # YOUR CODE HERE:
242     #   Predict the labels given the training data with the parameter self.W.
243     # ================================================================ #
244     prod = np.dot(X, np.transpose(self.W))
245
246     y_pred = np.argmax(prod, axis = 1)
247     # ================================================================ #
248     # END YOUR CODE HERE
249     # ================================================================ #
250
251     return y_pred
```

```python
import numpy as np

class Softmax(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

  def init_weights(self, dims):
    """
  Initializes the weight matrix of the Softmax classifier.
  Note that it has shape (C, D) where C is the number of
  classes and D is the feature size.
  """
    self.W = np.random.normal(size=dims) * 0.0001

  def loss(self, X, y):
    """
    Calculates the softmax loss.

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.

    Returns a tuple of:
    - loss as single float
    """

    # Initialize the loss to zero.
    loss = 0.0
    num_samples = X.shape[0]
    # ============================================================= #
    # YOUR CODE HERE:
    #   Calculate the normalized softmax loss.  Store it as the variable loss.
    #   (That is, calculate the sum of the losses of all the training
    #   set margins, and then normalize the loss by the number of
    #   training examples.)
    # ============================================================= #
    aY = X.dot(self.W.T)
    for i in range(num_samples):


      logSum = np.log(np.sum(np.exp(aY[i])))
      classProb = aY[i,y[i]]
      loss += -classProb+logSum

    # ============================================================= #
    # END YOUR CODE HERE
    # ============================================================= #
    loss = loss/num_samples
    return loss

  def loss_and_grad(self, X, y):
    """
  Same as self.loss(X, y), except that it also returns the gradient.

  Output: grad -- a matrix of the same dimensions as W containing
    the gradient of the loss with respect to W.
  """

    # Initialize the loss and gradient to zero.
    loss = 0.0
    grad = np.zeros_like(self.W)
    num_samples = X.shape[0]
    numClasses = self.W.shape[0]
    # ============================================================= #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the gradient
    #   as the variable grad.
    # ============================================================= #
    aY = X.dot(self.W.T)
    for i in range(num_samples):
      logSum = np.log(np.sum(np.exp(aY[i])))
      classProb = aY[i,y[i]]
      loss += -classProb+logSum

      for j in range(numClasses):
```

```
81          grad[j] += ((np.exp(X[i].dot(self.W.T)[j])/float(np.sum(np.exp(X[i].dot(self.W.T)))) ) - (y[i] == j)) * X[i]
82
83      loss = loss/num_samples
84      grad = grad/num_samples
85      # ============================================================= #
86      # END YOUR CODE HERE
87      # ============================================================= #
88
89      return loss, grad
90
91    def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
92      """
93      sample a few random elements and only return numerical
94      in these dimensions.
95      """
96
97      for i in np.arange(num_checks):
98        ix = tuple([np.random.randint(m) for m in self.W.shape])
99
100       oldval = self.W[ix]
101       self.W[ix] = oldval + h # increment by h
102       fxph = self.loss(X, y)
103       self.W[ix] = oldval - h # decrement by h
104       fxmh = self.loss(X,y) # evaluate f(x - h)
105       self.W[ix] = oldval # reset
106
107       grad_numerical = (fxph - fxmh) / (2 * h)
108       grad_analytic = your_grad[ix]
109       rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) + abs(grad_analytic))
110       print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic, rel_error))
111
112   def fast_loss_and_grad(self, X, y):
113     """
114     A vectorized implementation of loss_and_grad. It shares the same
115   inputs and ouptuts as loss_and_grad.
116     """
117     loss = 0.0
118     grad = np.zeros(self.W.shape) # initialize the gradient as zero
119     # ============================================================= #
120     # YOUR CODE HERE:
121     #   Calculate the softmax loss and gradient WITHOUT any for loops.
122     # ============================================================= #
123     aY = X.dot(np.transpose(self.W))
124     aExp = np.exp(aY)
125
126     aCol = np.sum(aExp, axis = 1)
127
128     logSum = np.log(aCol)
129     sum1 = -aY[np.arange(X.shape[0]), y]
130     loss = np.sum(sum1+logSum)/X.shape[0]
131
132     aNorm = aExp / np.transpose(np.matrix(aCol))
133
134
135     aNorm[np.arange(X.shape[0]),y] -= 1
136     grad = aNorm.T.dot(X)
137
138     grad = grad/X.shape[0]
139
140
141     # ============================================================= #
142     # END YOUR CODE HERE
143     # ============================================================= #
144
145     return loss, grad
146
147   def train(self, X, y, learning_rate=1e-3, num_iters=100,
148             batch_size=200, verbose=False):
149     """
150     Train this linear classifier using stochastic gradient descent.
151
152     Inputs:
153     - X: A numpy array of shape (N, D) containing training data; there are N
154       training samples each of dimension D.
155     - y: A numpy array of shape (N,) containing training labels; y[i] = c
156       means that X[i] has label 0 <= c < C for C classes.
157     - learning_rate: (float) learning rate for optimization.
158     - num_iters: (integer) number of steps to take when optimizing
159     - batch_size: (integer) number of training examples to use at each step.
160     - verbose: (boolean) If true, print progress during optimization.
```

```
161
162     Outputs:
163     A list containing the value of the loss function at each training iteration.
164     """
165     num_train, dim = X.shape
166     num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of classes
167
168     self.init_weights(dims=[np.max(y) + 1, X.shape[1]])  # initializes the weights of self.W
169
170     # Run stochastic gradient descent to optimize W
171     loss_history = []
172
173     for it in np.arange(num_iters):
174
175       X_batch = None
176       y_batch = None
177       indic = np.random.choice(num_train, batch_size)
178       X_batch = X[indic,:]
179       y_batch = y[indic]
180
181       # ========================================================== #
182       # YOUR CODE HERE:
183       #   Sample batch_size elements from the training data for use in
184       #       gradient descent.  After sampling,
185       #     - X_batch should have shape: (dim, batch_size)
186       #     - y_batch should have shape: (batch_size,)
187       #   The indices should be randomly generated to reduce correlations
188       #   in the dataset.  Use np.random.choice.  It's okay to sample with
189       #   replacement.
190       # ========================================================== #
191
192       # ========================================================== #
193       # END YOUR CODE HERE
194       # ========================================================== #
195
196       # evaluate loss and gradient
197
198       loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
199       loss_history.append(loss)
200
201       # ========================================================== #
202       # YOUR CODE HERE:
203       #   Update the parameters, self.W, with a gradient step
204       # ========================================================== #
205
206       self.W -= learning_rate*grad
207
208       # ========================================================== #
209       # END YOUR CODE HERE
210       # ========================================================== #
211
212       if verbose and it % 100 == 0:
213         print('iteration {} / {}: loss {}'.format(it, num_iters, loss))
214
215     return loss_history
216
217   def predict(self, X):
218     """
219     Inputs:
220     - X: N x D array of training data. Each row is a D-dimensional point.
221
222     Returns:
223     - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
224       array of length N, and each element is an integer giving the predicted
225       class.
226     """
227     y_pred = np.zeros(X.shape[0])
228     # ========================================================== #
229     # YOUR CODE HERE:
230     #   Predict the labels given the training data.
231     # ========================================================== #
232     prod = np.dot(X, np.transpose(self.W))
233
234     y_pred = np.argmax(prod, axis = 1)
235     # ========================================================== #
236     # END YOUR CODE HERE
237     # ========================================================== #
238     return y_pred
```