This is the k-nearest neighbors workbook for ECE C147/C247 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 [2]:

```
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 datas
et.

# 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-ipytho
n
%load_ext autoreload
%autoreload 2
```

In [3]:

```
# Set the path to the CIFAR-10 data
cifar10_dir = 'cifar-10-batches-py' # You need to update this line
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 [6]:

```
# 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 [7]:

```
# 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 [8]:

```
# Import the KNN class

from nndl import KNN
```

In [9]:

```
# 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) Store the training data and test data into knn's variables.

(2) Pros: Make the following implementation simple(knn object can use its own data). Cons: Memory we need will increase.

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 [10]:

```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition 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_L2, 'fro'
)))
```

```
Time to run code: 33.93133330345154
Frobenius norm of L2 distances: 7906696.077040902
```

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 [11]:

```
# Implement the function compute_L2_distances_vectorized() in the KNN class.
# In this function, you ought to achieve the same L2 distance but WITHOUT any fo
r 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('Frobenius norm of L2 distances: {}'.format(np.linalg.norm(dists_L2_vector
ized, 'fro')))
print('Difference in L2 distances between your KNN implementations (should be
 0): {}'.format(np.linalg.norm(dists_L2 - dists_L2_vectorized, 'fro')))
```

```
Time to run code: 0.24333500862121582
Frobenius norm of L2 distances: 7906696.077040902
Difference in L2 distances between your KNN implementations (should
be 0): 1.4651847440245846e-10
```

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 [14]:

```
# 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

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.
# ================================================================ #
pred = knn.predict_labels(dists_L2_vectorized)
count = 0
for i in range(len(y_test)):
    if pred[i] != y_test[i]:
        count += 1
error = count / y_test.shape[0]
# ================================================================ #
# 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 [25]:

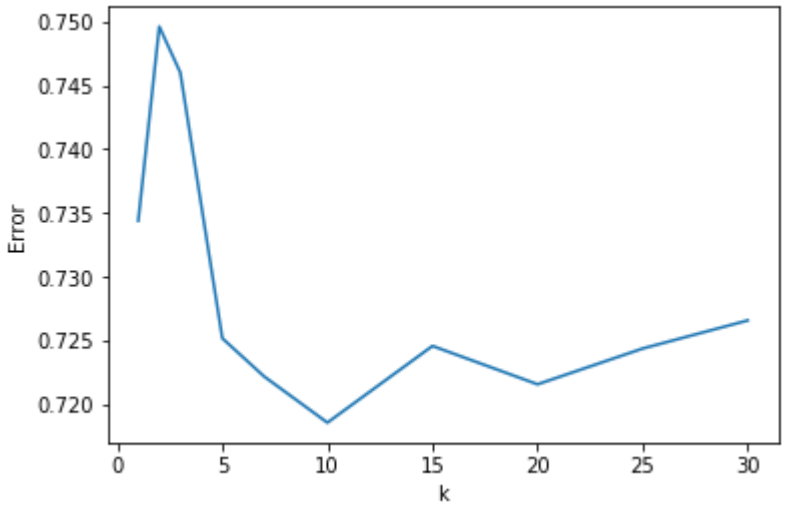
```
# Create the dataset folds for cross-valdiation.
num_folds = 5

X_train_folds = []
y_train_folds =  []

# ================================================================ #
# 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]
# ================================================================ #

length = int(len(X_train)/num_folds)
for i in range(1,num_folds+1):
    X_train_folds.append(X_train[(i-1)*length:i*length])
    y_train_folds.append(y_train[(i-1)*length:i*length])

# ================================================================ #
# 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 [30]:

```
time_start =time.time()

ks = [1, 2, 3, 5, 7, 10, 15, 20, 25, 30]

# ================================================================ #
# 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.
# ================================================================ #
errors = []
for k in ks:
    error = 0
    for i in range(num_folds):
        X_train = np.concatenate(X_train_folds[:i] + X_train_folds[i + 1:])
        y_train = np.concatenate(y_train_folds[:i] + y_train_folds[i + 1:])
        X_val = X_train_folds[i]
        y_val = y_train_folds[i]
        knn.train(X_train, y_train)
        dists = knn.compute_L2_distances_vectorized(X_val)
        pred = knn.predict_labels(dists, k = k)

        count = 0
        for i in range(len(pred)):
            if pred[i] != y_val[i]:
                count += 1
        error += count / len(pred)
    error = error/num_folds
    errors.append(error)

plt.figure()
plt.plot(ks, errors)
plt.ylabel('Error')
plt.xlabel('k')
plt.show()

print('optimal k: %d' %ks[errors.index(min(errors))])
print('cross-validation error: %s'%min(errors))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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




```
optimal k: 10
cross-validation error: 0.7186
Computation time: 147.48
```

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) 10.

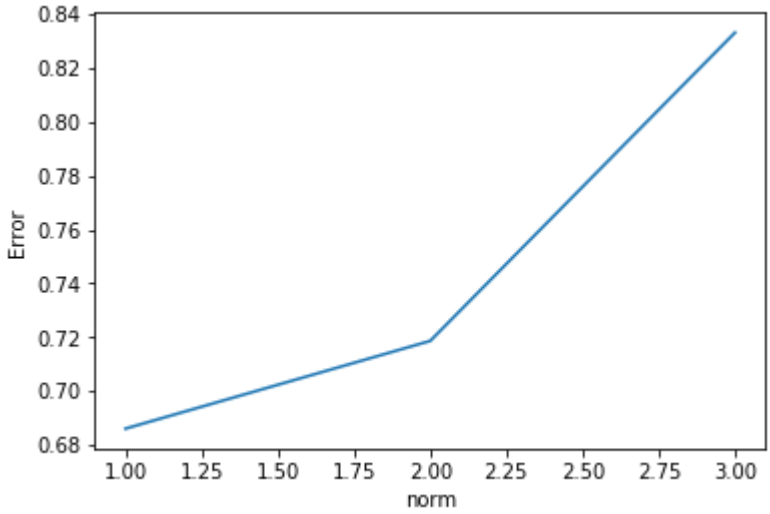
(2) 0.7186.

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 [31]:

```
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]

# ================================================================ #
# 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 error
#   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.
# ================================================================ #
# use optimal k we just got.
k_optimal = ks[errors.index(min(errors))]
errors = []
for n in norms:
    error = 0
    for i in range(num_folds):
        X_train = np.concatenate(X_train_folds[:i] + X_train_folds[i + 1:])
        y_train = np.concatenate(y_train_folds[:i] + y_train_folds[i + 1:])
        X_val = X_train_folds[i]
        y_val = y_train_folds[i]
        knn_norm = KNN()
        knn_norm.train(X_train, y_train)
        dists = knn_norm.compute_distances(X_val,norm=n)
        pred = knn_norm.predict_labels(dists, k = k_optimal)

        count = 0
        for i in range(len(pred)):
            if pred[i] != y_val[i]:
                count += 1
        error += count / len(pred)
    error = error/num_folds
    errors.append(error)

plt.figure()
plt.plot([1,2,3], errors)
plt.ylabel('Error')
plt.xlabel('norm')
plt.show()

norms_name = ['L1', 'L2', 'Infinty']
print('optimal norm: ' +norms_name[errors.index(min(errors))])
print('cross-validation error: %s'%min(errors))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
print('Computation time: %.2f'%(time.time()-time_start))
```




```
optimal norm: L1
cross-validation error: 0.686
Computation time: 1222.27
```

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) 0.686.

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 [35]:

```
error = 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 = KNN()
knn.train(X=X_train, y=y_train)
dists = knn.compute_distances(X=X_test, norm=L1_norm)
pred = knn.predict_labels(dists, k=10)
error = 0
for i in range(len(y_test)):
    if pred[i] != y_test[i]:
        error += 1
error /= len(y_test)

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

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

```
Error rate achieved: 0.714
```

Question:

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

Answer:

0.0165

In [ ]:

```
def compute_distances(self, X, norm=None):
  """
  Compute the distance between each test point in X and each training point
  in self.X_train.

  Inputs:
  - X: A numpy array of shape (num_test, D) containing test data.
  - norm: the function with which the norm is taken.

  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.
  """
  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,:])

      # ================================================================ #
      # 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.
  # ================================================================ #

  test_sum = np.sum(X**2, axis=1).reshape((num_test, 1))
  train_sum = np.sum(self.X_train**2, axis=1).reshape((1, num_train))
  test_train = np.dot(X, self.X_train.T)
  dists = np.sqrt(test_sum + train_sum - 2 * test_train)

  # ================================================================ #
  # 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.
    # ================================================================ #

    idx = sorted(range(dists.shape[1]), key = lambda j : dists[i,:][j])[:k]
    k_closest_y = [self.y_train[i] for i in idx]
    y_pred[i] = max(set(k_closest_y),key = k_closest_y.count)

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

  return y_pred
```

This is the svm workbook for ECE C147/C247 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 datas
et.
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-ipytho
n
%load_ext autoreload
%autoreload 2
```

In [3]:

```
# Set the path to the CIFAR-10 data
cifar10_dir = 'cifar-10-batches-py' # You need to update this line
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 [4]:

```
# 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 [5]:

```
# 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 [6]:

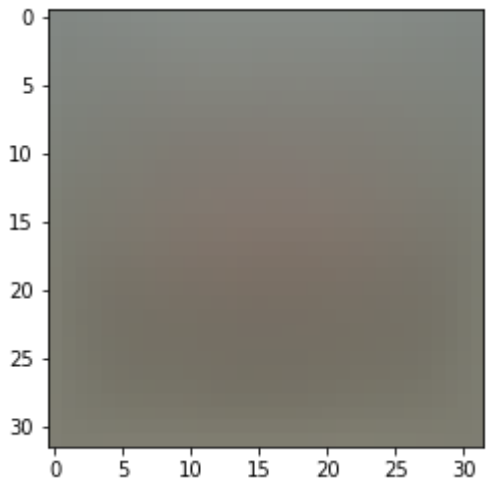
```
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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 [7]:

```
# 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 the mean i
mage
plt.show()
```

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



In [8]:

```
# 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 [9]:

```
# 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) In KNN, we already normalized the data by calculating the 'distance'. But in SVM, we need this subtraction to normalize the data to be zero-centered to get better performance of our model

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 [11]:

```
# 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 use a ran
dom 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 [16]:

```
## Implement the loss function for in the SVM class(nndl/svm.py), svm.loss()

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 15569.977915410238.
```

SVM gradient

In [33]:

```
## 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 i
mplemented the gradient correctly.
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 15.369995 analytic: 15.369995, relative error: 2.739087e-
09
numerical: -26.315199 analytic: -26.315198, relative error: 1.152446
e-08
numerical: 0.536032 analytic: 0.536031, relative error: 5.948596e-07
numerical: -29.212061 analytic: -29.212061, relative error: 4.224807
e-09
numerical: 20.783484 analytic: 20.783484, relative error: 4.585006e-
09
numerical: 5.317689 analytic: 5.317690, relative error: 5.027553e-08
numerical: -1.712038 analytic: -1.712038, relative error: 5.815066e-
08
numerical: -7.338403 analytic: -7.338403, relative error: 5.894616e-
09
numerical: 16.356000 analytic: 16.356000, relative error: 3.227819e-
09
numerical: -0.460831 analytic: -0.460832, relative error: 2.353996e-
07
```

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 [34]:

```
import time
```

In [37]:

```
## Implement svm.fast_loss_and_grad which calculates the loss and gradient
#    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_vectorized,
np.linalg.norm(grad_vectorized, 'fro'), toc - tic))

# The losses should match but your vectorized implementation should be much fast
er.
print('difference in loss / grad: {} / {}'.format(loss - loss_vectorized, np.lin
alg.norm(grad - grad_vectorized)))

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

```
Normal loss / grad_norm: 14817.050152377105 / 2177.4920258386 comput
ed in 0.054823875427246094s
Vectorized loss / grad: 14817.05015237709 / 2177.4920258386 computed
in 0.003526926040649414s
difference in loss / grad: 1.4551915228366852e-11 / 6.11409545260408
4e-12
```

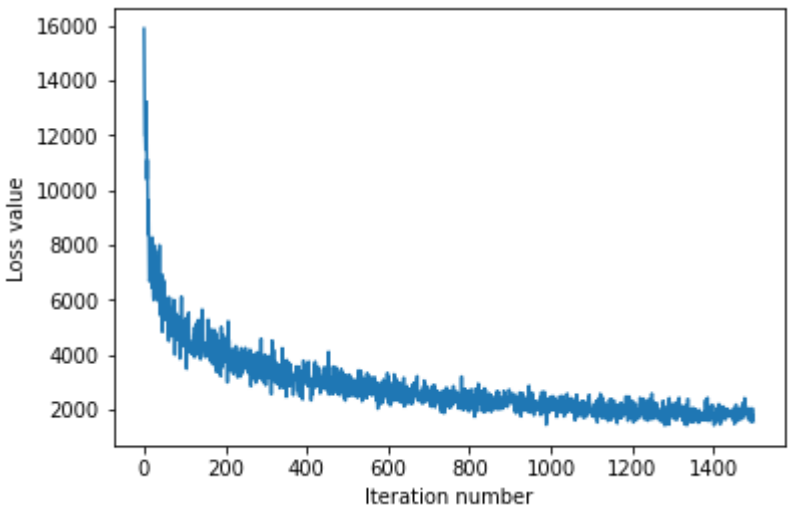
Stochastic gradient descent

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 [63]:

```
# Implement svm.train() by filling in the code to extract a batch of data
# 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 15877.418847411274
iteration 100 / 1500: loss 4438.660911273349
iteration 200 / 1500: loss 3913.9954900402063
iteration 300 / 1500: loss 3715.333666653142
iteration 400 / 1500: loss 3172.805916491734
iteration 500 / 1500: loss 2530.461298297793
iteration 600 / 1500: loss 2230.5901661338344
iteration 700 / 1500: loss 2306.4226560076822
iteration 800 / 1500: loss 2671.1476399755857
iteration 900 / 1500: loss 2306.8400548682857
iteration 1000 / 1500: loss 1959.5245130098583
iteration 1100 / 1500: loss 1893.27240981261
iteration 1200 / 1500: loss 2246.0905408727053
iteration 1300 / 1500: loss 2050.0591690221113
iteration 1400 / 1500: loss 1811.4605328840937
That took 2.4766201972961426s
```



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

In [64]:

```
## Implement svm.predict() and use it to compute the training and testing error.

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

```
training accuracy: 0.28685714285714287
validation accuracy: 0.31
```

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 [67]:

```
# ================================================================ #
# 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 VALIDATION 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
# ================================================================ #
learning_rates = [1e-5, 5e-5, 1e-4, 5e-4, 1e-3, 5e-3, 1e-2, 5e-2, 1e-1, 5e-1]
accuracy = []
for lr in learning_rates:
    svm.train(X_train, y_train, learning_rate=lr,num_iters=1500, verbose=False)
    pred = svm.predict(X_val)
    correct_num = np.sum(y_val == pred)
    accuracy.append(correct_num/len(y_val))

svm.train(X_train, y_train, learning_rate=learning_rates[np.argmax(accuracy)],nu
m_iters=1500, verbose=False)
pred = svm.predict(X_test)
correct_num = np.sum(y_test == pred)
error_rate = 1 - correct_num/ len(y_test)
print("Learning rate selected: ", learning_rates[np.argmax(accuracy)])
print("Validation error: ", 1 - accuracy[np.argmax(accuracy)])
print("Error rate on the test set ", error_rate)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Learning rate selected:  0.01
Validation error:  0.649
Error rate on the test set  0.69
```

```
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

  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.)
  # ============================================================= #
      scores = np.dot(self.W, X[i])
      score_cur_y = scores[y[i]]
      margin = scores + 1 - score_cur_y
      margin[y[i]] = 0
      margin = np.maximum(margin, 0)
      loss += np.sum(margin)
  loss /= float(num_train)
  # ============================================================= #
  # END YOUR CODE HERE
  # ============================================================= #

  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)

  for i in np.arange(num_train):
  # ============================================================= #
  # YOUR CODE HERE:
  #   Calculate the SVM loss and the gradient.  Store the gradient in
  #   the variable grad.
  # ============================================================= #
      scores = np.dot(self.W, X[i])
      score_cur_y = scores[y[i]]
      margin = scores + 1 - score_cur_y
      margin[y[i]] = 0
      margin = np.maximum(margin, 0)
      loss += np.sum(margin)

      count_positive = 0
      for j in range(len(margin)):
          if margin[j] > 0:
              grad[j] += X[i]
              count_positive += 1

      grad[y[i]] -= count_positive * X[i]

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

  loss /= num_train
  grad /= num_train

  return loss, grad
```

```
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.
  # ============================================================ #
  num_train = X.shape[0]
  scores = np.dot(self.W, X.T) # totally this time
  true_y_scores = scores[y, range(len(y)) ]
  margin = scores - true_y_scores + np.ones(scores.shape)
  margin[y, range(len(y))] = 0
  margin = np.maximum(margin, 0)
  loss = np.sum(margin) / num_train
  # ============================================================ #
  # END YOUR CODE HERE
  # ============================================================ #



  # ============================================================ #
  # YOUR CODE HERE:
  #   Calculate the SVM grad WITHOUT any for loops.
  # ============================================================ #
  margin[margin > 0] = 1
  margin[y, range(len(y))] = -1 * np.sum(margin, axis=0)
  grad = np.dot(margin, X) / 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):
  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.
    # ============================================================ #
    indices = np.random.choice(X.shape[0], batch_size)
    X_batch = X[indices]
    y_batch = y[indices]
    # ============================================================ #
    # 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
      # ============================================================ #
    # END YOUR CODE HERE
    # ============================================================ #

    if verbose and it % 100 == 0:
      print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

  return loss_history
```

```
def predict(self, X):
  """
  Inputs:
  - X: N x D array of training data. Each row is a D-dimensional point.

  Returns:
  - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
    array of length N, and each element is an integer giving the predicted
    class.
  """
  y_pred = np.zeros(X.shape[1])


  # ============================================================ #
  # YOUR CODE HERE:
  #   Predict the labels given the training data with the parameter self.W.
  # ============================================================ #
  scores = np.dot(self.W, X.T)
  y_pred = np.argmax(scores, axis=0)
  # ============================================================ #
  # END YOUR CODE HERE
  # ============================================================ #

  return y_pred
```

This is the softmax workbook for ECE C147/C247 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 [1]:

```
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
```

In [3]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num
_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cifar-10-batches-py' # You need to update this line
    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_CIFAR10_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,)
```

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 [4]:

```
from nndl import Softmax
```

In [5]:

```
# 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 rando
m 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 [8]:

```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

In [9]:

```
print(loss)
```

```
2.3277607028048863
```

Question:

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

Answer:

With a initial randomly uniform weight matrix and totally 10 classes, the loss should be around -ln(1/10) ~ 2.3

Softmax gradient

In [12]:

```
## 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 then
#   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 i
mplemented the gradient correctly.
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 2.370950 analytic: 2.370950, relative error: 1.886866e-08
numerical: 0.593285 analytic: 0.593285, relative error: 7.380073e-09
numerical: 2.813118 analytic: 2.813118, relative error: 4.273877e-09
numerical: 1.769662 analytic: 1.769662, relative error: 2.670171e-08
numerical: 1.673273 analytic: 1.673273, relative error: 2.090232e-08
numerical: -0.459957 analytic: -0.459957, relative error: 1.676457e-
08
numerical: -0.522956 analytic: -0.522956, relative error: 1.619343e-
08
numerical: 1.310494 analytic: 1.310494, relative error: 2.831244e-08
numerical: 3.617351 analytic: 3.617351, relative error: 3.287133e-09
numerical: 2.111043 analytic: 2.111043, relative error: 1.342032e-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 [13]:

```
import time
```

In [19]:

```
## Implement softmax.fast_loss_and_grad which calculates the loss and gradient
#    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_dev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vectorized,
np.linalg.norm(grad_vectorized, 'fro'), toc - tic))

# The losses should match but your vectorized implementation should be much fast
er.
print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized, np.lin
alg.norm(grad - grad_vectorized)))

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

```
Normal loss / grad_norm: 2.3567265846170318 / 340.6060881862438 comp
uted in 0.06967616081237793s
Vectorized loss / grad: 2.356726584617034 / 340.6060881862438 comput
ed in 0.00357818603515625s
difference in loss / grad: -2.220446049250313e-15 /3.12498042513341e
-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?

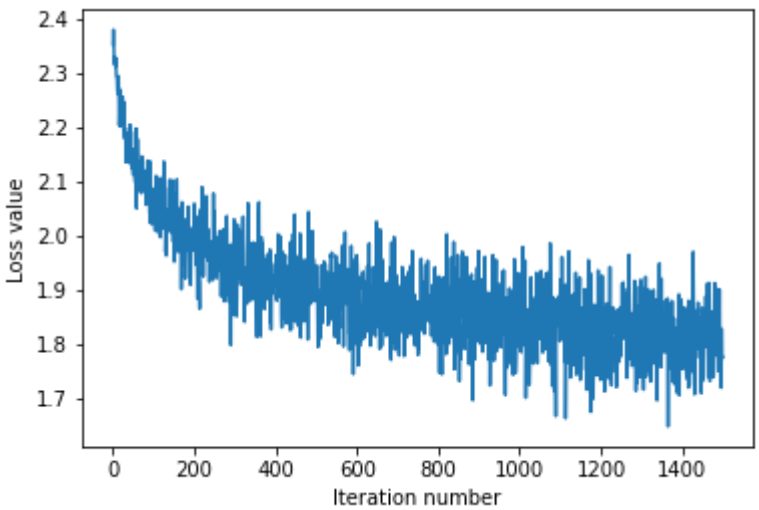
Answer:

The gradients are different because of different loss function.

In [24]:

```
# 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 2.3518126906059265
iteration 100 / 1500: loss 2.0860081811621445
iteration 200 / 1500: loss 2.0589314799843645
iteration 300 / 1500: loss 1.887860412704343
iteration 400 / 1500: loss 1.9087001113627347
iteration 500 / 1500: loss 1.9449831094343881
iteration 600 / 1500: loss 1.959694578138808
iteration 700 / 1500: loss 1.882098015704022
iteration 800 / 1500: loss 1.9121730024146637
iteration 900 / 1500: loss 1.8872861722918102
iteration 1000 / 1500: loss 1.794770761842581
iteration 1100 / 1500: loss 1.918992746737748
iteration 1200 / 1500: loss 1.745805085594816
iteration 1300 / 1500: loss 1.832135029183785
iteration 1400 / 1500: loss 1.8208973043354932
That took 2.550896167755127s
```



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

In [25]:

```
## Implement softmax.predict() and use it to compute the training and testing er
ror.

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

```
training accuracy: 0.38087755102040816
validation accuracy: 0.394
```

Optimize the softmax classifier

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

In [26]:

```
np.finfo(float).eps
```

Out[26]:

```
2.220446049250313e-16
```

In [29]:

```
# ================================================================ #
# 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 validation error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
learning_rates = [1e-6, 5e-6, 1e-5, 5e-5, 1e-4, 5e-4, 1e-3, 5e-3, 1e-2, 5e-2, 1e
-1, 5e-1]
accuracy = []

for lr in learning_rates:
    softmax.train(X_train, y_train, learning_rate=lr,num_iters=1500, verbose=Fal
se)
    pred = softmax.predict(X_val)
    correct_num = np.sum(y_val == pred)
    accuracy.append(correct_num/len(y_val))

softmax.train(X_train, y_train, learning_rate=learning_rates[np.argmax(accuracy
)],num_iters=1500, verbose=False)
pred = softmax.predict(X_test)
correct_num = np.sum(y_test == pred)
error_rate = 1 - correct_num/ len(y_test)
print("Learning rate selected: ", learning_rates[np.argmax(accuracy)])
print("Validation error: ", 1 - accuracy[np.argmax(accuracy)])
print("Error rate on the test set ", error_rate)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Learning rate selected:  1e-06
Validation error:  0.589
Error rate on the test set  0.601
```

In [ ]:

```

```

```
def loss(self, X, y):
  # Initialize the loss to zero.
  loss = 0.0
  # ============================================================ #
  # 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.)
  # ============================================================ #
  num_train = X.shape[0]
  scores = np.dot(self.W, X.T)
  for i in range(num_train):
    score = scores[:, i]
    score -= np.max(score)
    loss -= score[y[i]]
    sum_i = np.sum(np.exp(score))
    loss += np.log(sum_i)
  loss /= num_train
  # ============================================================ #
  # END YOUR CODE HERE
  # ============================================================ #
  return loss
```

```
def loss_and_grad(self, X, y):
  # Initialize the loss and gradient to zero.
  loss = 0.0
  grad = np.zeros_like(self.W)
  # ============================================================ #
  # YOUR CODE HERE:
  #   Calculate the softmax loss and the gradient. Store the gradient
  #   as the variable grad.
  # ============================================================ #
  num_train = X.shape[0]
  scores = np.dot(self.W, X.T)
  for i in range(num_train):
    score = scores[:, i]
    score -= np.max(score)
    loss -= score[y[i]]
    sum_i = np.sum(np.exp(score))
    loss += np.log(sum_i)

    for j in range(10):
      grad[j] += (np.exp(score[j]) / sum_i) * X[i]
    grad[y[i]] -= X[i]
  loss /= num_train
  grad /= num_train
  # ============================================================ #
  # END YOUR CODE HERE
  # ============================================================ #
  return loss, grad
```

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

  # ============================================================ #
  # YOUR CODE HERE:
  #   Calculate the softmax loss and gradient WITHOUT any for loops.
  # ============================================================ #

  num_train = X.shape[0]
  scores = np.dot(self.W, X.T)
  scores -= np.max(scores, axis=0, keepdims=True)
  soft_max = np.exp(scores) / np.sum(np.exp(scores), axis=0, keepdims=True)
  true_y = soft_max[y, range(num_train)]
  loss = np.sum(-np.log(true_y.clip(np.finfo(float).eps,)))
  loss /= num_train

  soft_max[y, range(num_train)] -= 1
  grad = np.dot(soft_max, X)
  grad /= num_train

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

  return loss, grad
```

```
def train(self, X, y, learning_rate=1e-3, num_iters=100,
          batch_size=200, verbose=False):
  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.
    # ========================================================== #
    indices = np.random.choice(X.shape[0], batch_size)
    X_batch = X[indices]
    y_batch = y[indices]
    # ========================================================== #
    # 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 = self.W - grad* learning_rate
    # ========================================================== #
    # END YOUR CODE HERE
    # ========================================================== #
    if verbose and it % 100 == 0:
      print('iteration {} / {}: loss {}'.format(it, num_iters, loss))
  return loss_history
```

```
def predict(self, X):
  """
  Inputs:
  - X: N x D array of training data. Each row is a D-dimensional point.

  Returns:
  - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
    array of length N, and each element is an integer giving the predicted
    class.
  """
  y_pred = np.zeros(X.shape[1])
  # ========================================================== #
  # YOUR CODE HERE:
  #   Predict the labels given the training data.
  # ========================================================== #
  y_pred = np.argmax(np.dot(self.W, X.T),axis=0)
  # ========================================================== #
  # END YOUR CODE HERE
  # ========================================================== #

  return y_pred
```