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 [19]:
```
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 [20]:
```
# 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 [21]:
```
# 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 [22]:
```
# 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 [23]:
```
# Import the KNN class

from nndl import KNN
```

In [24]:
```
# 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) In the training phase , knn.train(), the KNN remembers the training data

(2) Pros: It is simple. Memomirze the data Cons: Takes a lot of memory and computation power.

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 [25]:
```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition of the no
rm
#   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: 34.22079277038574
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 [26]:
```
# 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 for l
oops.
# 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 (should be 0): {
}'.format(np.linalg.norm(dists_L2 - dists_L2_vectorized, 'fro')))
```

```
Time to run code: 0.35863304138183594
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 [27]:
```
# 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.
# ================================================================ #
y_test_pred = knn.predict_labels(dists_L2_vectorized,1)

error_count = np.sum(y_test_pred != y_test)
error = float(error_count) / num_test
# ================================================================ #
# 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

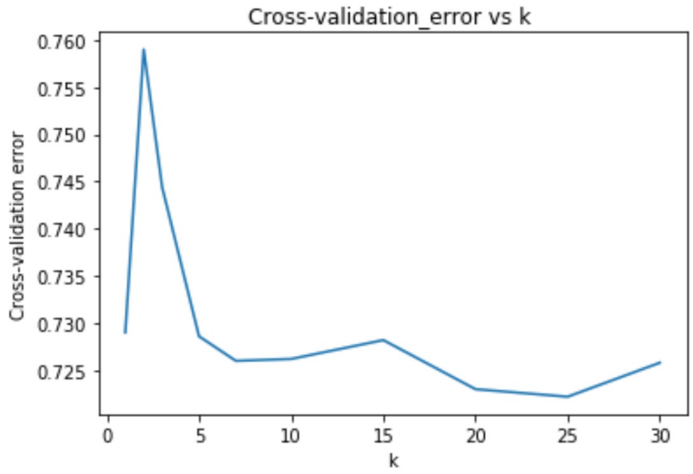
First, we will create the training and validation folds for use in k-fold cross validation.

In [36]:
```
# 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]
# ================================================================ #
cv_idx = np.arange(num_training)
np.random.shuffle(cv_idx)

X_train_folds = np.array(np.array_split(X_train[cv_idx],num_folds))
y_train_folds = np.array(np.array_split(y_train[cv_idx],num_folds))

# ================================================================ #
# 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 [57]:
```
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.
# ================================================================ #

cross_validation_errors = []
for i in ks:
  k_errors = []
  for j in range(num_folds):

    training = [ x for x in range(num_folds) if x != j ]
    X_train_cycle = np.concatenate(X_train_folds[training])
    y_train_cycle = np.concatenate(y_train_folds[training])


    knn.train(X_train_cycle,y_train_cycle)
    dists_L2_vectorK = knn.compute_L2_distances_vectorized(X=X_train_folds[j])

    y_test_pred = knn.predict_labels(dists_L2_vectorK,i)

    num_errors = np.sum(y_train_folds[j] != y_test_pred)

    k_errors.append(float(num_errors) / (len(y_test_pred)))

  cross_validation_errors.append(np.mean(k_errors))

plt.plot(ks,cross_validation_errors)
plt.title('Cross-validation_error vs k ')
plt.xlabel('k')
plt.ylabel('Cross-validation error')
plt.show()


print(np.argmin(cross_validation_errors))
print(cross_validation_errors)
print(ks[np.argmin(cross_validation_errors)])
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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




```
8
[0.7289999999999999, 0.759, 0.7444, 0.7286, 0.726, 0.7262000000000001, 0.7282, 0
.723, 0.7222000000000001, 0.7258]
25
Computation time: 43.82
```

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:

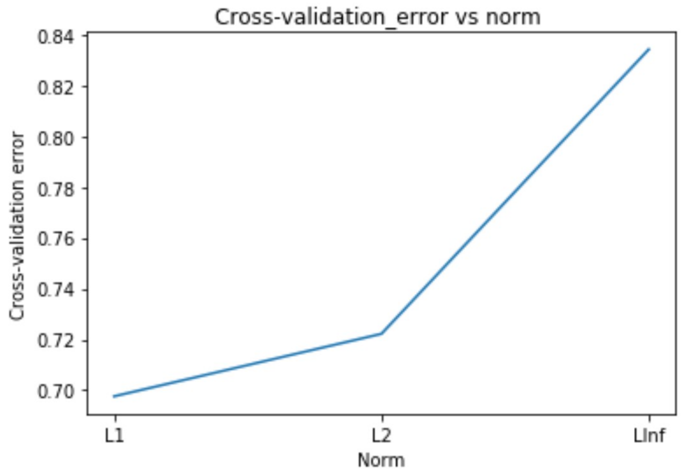
(1) The best value of k is 25

(2) The cross-validation error for k=25 is 0.7222

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 [62]:
```
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.
# ================================================================ #
cross_validation_errors = []
for i in norms:
  k_errors = []
  for j in range(num_folds):

    training = [ x for x in range(num_folds) if x != j ]
    X_train_cycle = np.concatenate(X_train_folds[training])
    y_train_cycle = np.concatenate(y_train_folds[training])


    knn.train(X_train_cycle,y_train_cycle)

    if(i == L2_norm):
        dists = knn.compute_L2_distances_vectorized(X=X_train_folds[j])
    else:
        dists = knn.compute_distances(X=X_train_folds[j],norm=i)


    y_test_pred = knn.predict_labels(dists,25) #25 is best k

    num_errors = np.sum(y_train_folds[j] != y_test_pred)

    k_errors.append(float(num_errors) / (len(y_test_pred)))

  cross_validation_errors.append(np.mean(k_errors))

plt.xticks(range(3), ["L1","L2","LInf"])
plt.plot(range(3), cross_validation_errors)
plt.title('Cross-validation_error vs norm ')
plt.xlabel('Norm')
plt.ylabel('Cross-validation error')
plt.show()


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




```
Computation time: 538.08
```

In [63]:
```
print(cross_validation_errors)
```

```
[0.6976000000000001, 0.7222000000000001, 0.8342]
```

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 has the best cross-validation error

(2) The cross validation for L1 norm and k=25 is 0.6976

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 [69]:
```
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.train(X_train,y_train)
dists = np.zeros((X_test.shape[0], X_train.shape[0]))

for i in range(X_test.shape[0]):
    for j in range(X_train.shape[0]):
        dists[i,j] = L1_norm(knn.X_train[j] - X_test[i])

y_pred = knn.predict_labels(dists,25)

error = np.sum(y_pred != y_test ) / len(y_pred)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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

```
Error rate achieved: 0.728
```

Question:

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

Answer:

The cross validation error slightly increase for L2 norm and k=25 (From 0.726 to 0.728)

```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):
    """
    Inputs:
    - X is a numpy array of size (num_examples, D)
    - y is a numpy array of size (num_examples, )
    """
    self.X_train = X
    self.y_train = y

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

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

      y_indicies = np.argsort(dists[i, :], axis = 0)
      closest_y = self.y_train[y_indicies[:k]]
```

```
            y_pred[i] = np.argmax(np.bincount(closest_y))

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

        return y_pred
```

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 [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 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
```

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

In [3]:

```
# 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 [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)
```

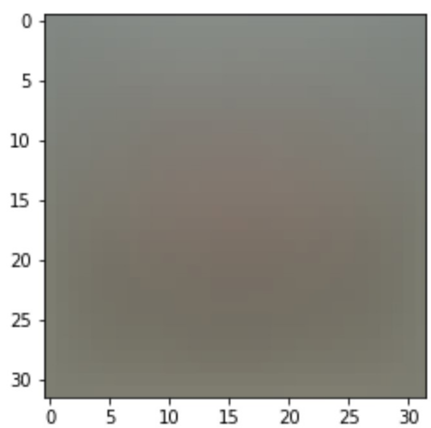
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

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 imag
e
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)Subtracting same vector from every single data point, they will move in the same direction and distance and the relationships between individual data points would not change. The nearest neighbours will remain the same as before. Therefore we don't perform mean-subtraction on the data for KNN.

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 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 [12]:
```
## 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 15569.977915410187.
```

SVM gradient

In [13]:
```
## 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 impl
emented the gradient correctly.
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -8.564456 analytic: -8.564455, relative error: 3.874708e-08
numerical: 5.957144 analytic: 5.957143, relative error: 8.856930e-08
numerical: -2.471037 analytic: -2.471037, relative error: 3.267331e-08
numerical: 10.832455 analytic: 10.832456, relative error: 3.319934e-08
numerical: -5.622402 analytic: -5.622402, relative error: 2.587467e-08
numerical: -2.404375 analytic: -2.404376, relative error: 1.760848e-07
numerical: 10.299545 analytic: 10.299546, relative error: 5.181998e-08
numerical: -11.770112 analytic: -11.770112, relative error: 2.908743e-09
numerical: -5.572953 analytic: -5.572952, relative error: 2.745204e-08
numerical: -22.985674 analytic: -22.985674, relative error: 9.956625e-09
```

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 [14]:
```
import time
```

In [15]:
```
## 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.nor
m(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 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
```

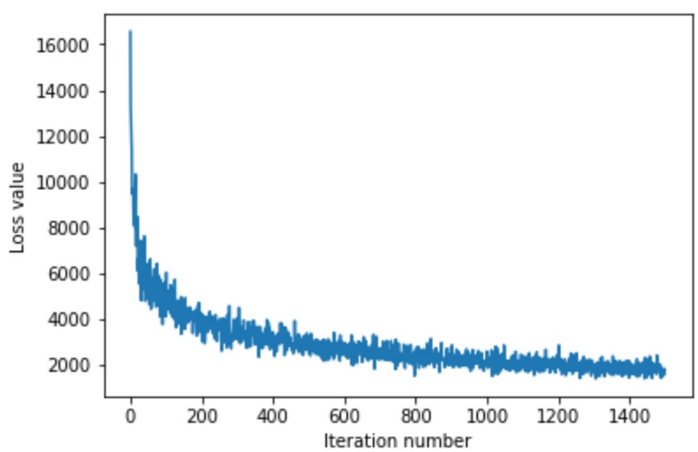
```
Normal loss / grad_norm: 15449.447848393516 / 2172.948189419817 computed in 0.04
158282279968262s
Vectorized loss / grad: 15449.447848393529 / 2172.948189419817 computed in 0.007
0226192474365234s
difference in loss / grad: -1.2732925824820995e-11 / 7.565170600556581e-12
```

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 [16]:
```
# 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 16557.38000190916
iteration 100 / 1500: loss 4701.089451272713
iteration 200 / 1500: loss 4017.3331379427877
iteration 300 / 1500: loss 3681.9226471953616
iteration 400 / 1500: loss 2732.6164373988995
iteration 500 / 1500: loss 2786.6378424645054
iteration 600 / 1500: loss 2837.035784278267
iteration 700 / 1500: loss 2206.2348687399326
iteration 800 / 1500: loss 2269.0388241169803
iteration 900 / 1500: loss 2543.23781538592
iteration 1000 / 1500: loss 2566.692135726826
iteration 1100 / 1500: loss 2182.068905905164
iteration 1200 / 1500: loss 1861.1182244250451
iteration 1300 / 1500: loss 1982.9013858528256
iteration 1400 / 1500: loss 1927.5204158582117
That took 8.2634859085083s
```



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

In [17]:
```
## 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: 0.28530612244897957
validation accuracy: 0.3
```

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 [21]:
```
# ================================================================ #
# 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
# ================================================================ #
valid_accuracy = []
learn_rates = [1e-3, 1e-4, 2e-4, 3e-3]
for i in learn_rates:
    loss_hist = svm.train(X_train, y_train, learning_rate=i,num_iters=1500, verbose
=True)
    y_val_pred = svm.predict(X_val)
    accuracy = np.mean(np.equal(y_val, y_val_pred))
    valid_accuracy.append(accuracy)

print(np.max(valid_accuracy))
print(valid_accuracy)
print(learn_rates[np.argmax(valid_accuracy)])

# We got best results for learning rate = 3e-3
svm.train(X_train, y_train, learning_rate=3e-3, num_iters=1500, verbose=True)

y_test_pred = svm.predict(X_test)
test_accuracy = np.mean(np.equal(y_test, y_test_pred))

print(1- test_accuracy)

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

```
iteration 0 / 1500: loss 16560.893685621082
iteration 100 / 1500: loss 3623.7705569357518
iteration 200 / 1500: loss 3805.4093016704574
iteration 300 / 1500: loss 3076.966650071273
iteration 400 / 1500: loss 3291.0420523193457
iteration 500 / 1500: loss 2510.259068015963
iteration 600 / 1500: loss 2011.9538581084523
iteration 700 / 1500: loss 2495.61248363399
iteration 800 / 1500: loss 1893.4811909971002
iteration 900 / 1500: loss 2246.277133002418
iteration 1000 / 1500: loss 1911.9791615698737
iteration 1100 / 1500: loss 1676.6053395089618
iteration 1200 / 1500: loss 1862.8088347057462
iteration 1300 / 1500: loss 1996.7656264566133
iteration 1400 / 1500: loss 1278.3151396785659
iteration 0 / 1500: loss 18554.048823606085
iteration 100 / 1500: loss 6033.438339949977
iteration 200 / 1500: loss 5981.636611349664
iteration 300 / 1500: loss 5058.831887159264
iteration 400 / 1500: loss 4738.621134558973
iteration 500 / 1500: loss 4836.291879775211
iteration 600 / 1500: loss 4591.249546646618
iteration 700 / 1500: loss 4714.200211267691
iteration 800 / 1500: loss 2953.409141807118
iteration 900 / 1500: loss 3977.589800971931
iteration 1000 / 1500: loss 2889.112012181447
iteration 1100 / 1500: loss 3808.4095429493323
iteration 1200 / 1500: loss 3782.7123487881654
iteration 1300 / 1500: loss 4658.69510652719
iteration 1400 / 1500: loss 3199.1302244032604
iteration 0 / 1500: loss 16103.629956431561
iteration 100 / 1500: loss 5826.152302268519
iteration 200 / 1500: loss 4147.24666178521
iteration 300 / 1500: loss 4400.640148040285
iteration 400 / 1500: loss 3762.5521316974914
iteration 500 / 1500: loss 4563.125572962141
iteration 600 / 1500: loss 3781.8651770599427
iteration 700 / 1500: loss 3793.2520111974904
iteration 800 / 1500: loss 2926.9624736546657
iteration 900 / 1500: loss 2645.629092113711
iteration 1000 / 1500: loss 3300.1505381906604
iteration 1100 / 1500: loss 3132.8954965155026
iteration 1200 / 1500: loss 2217.6074096181746
iteration 1300 / 1500: loss 2698.5977784662305
iteration 1400 / 1500: loss 2522.2625720666656
iteration 0 / 1500: loss 20128.17245105615
iteration 100 / 1500: loss 7751.511097239059
iteration 200 / 1500: loss 4916.053050814353
iteration 300 / 1500: loss 5157.881132939107
iteration 400 / 1500: loss 5193.519610053537
iteration 500 / 1500: loss 4889.052637618139
iteration 600 / 1500: loss 3946.715386139158
iteration 700 / 1500: loss 3509.0897477479766
iteration 800 / 1500: loss 4386.381366955726
iteration 900 / 1500: loss 4723.461029332257
iteration 1000 / 1500: loss 5672.401494890504
iteration 1100 / 1500: loss 5055.655601731257
iteration 1200 / 1500: loss 4181.470015002171
iteration 1300 / 1500: loss 6165.799309390361
iteration 1400 / 1500: loss 7369.856058857844
0.318
[0.301, 0.264, 0.286, 0.318]
0.003
iteration 0 / 1500: loss 14827.119740294054
```

```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

    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 = X[i].dot(self.W.T)
        class_score = scores[y[i]]
        for j in np.arange(num_classes):
          if j == y[i]:
            continue
          hinge = scores[j] - class_score + 1
          if hinge > 0:
            loss += hinge

    loss /= num_train

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

    return loss

  def loss_and_grad(self, X, y):
    """
```

```python
    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.
    # ================================================================ #
        count = 0;
        scores = X[i].dot(self.W.T)
        class_score = scores[y[i]]
        for j in np.arange(num_classes):
            if j == y[i]:
                continue
            hinge = scores[j] - class_score + 1
            if hinge > 0:
                count = count + 1
                loss += hinge
                grad[j,:] += X[i]

        grad[y[i],:] -= count* 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
```

```python
    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.
    # ================================================================ #
    scores = X.dot(self.W.T)
    score_class = scores[np.arange(X.shape[0]),y][:,np.newaxis]
    hinge = np.maximum(0,scores - score_class + 1)
    hinge[np.arange(X.shape[0]),y] = 0

    loss = (np.sum(hinge) / X.shape[0])


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



    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the SVM grad WITHOUT any for loops.
    # ================================================================ #

    arr = np.zeros(hinge.shape)
    arr[hinge > 0] = 1
    arr[np.arange(X.shape[0]),y] = -np.sum(hinge > 0 , axis=1)
    grad = (X.T.dot(arr)/X.shape[0]).T

    # ================================================================ #
    # 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.
      # ================================================================ #
      indexes = np.random.choice(num_train, batch_size)
      X_batch = X[indexes]
      y_batch = y[indexes]
      # ================================================================ #
      # 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.
    # ================================================================ #
    y_pred = np.argmax(X.dot(self.W.T), axis=1)
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

    return y_pred
```

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 [2]:

```
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 [3]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_de
v=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'
    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 random s
eed.

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 [6]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

In [7]:
```
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:

Since we are using random weights , the softmax assigns 1/c probability to each class. In our case c=10 and the loss is almost equal to -log(1/10). Predictions will not be correct due to the random weights and there will always be a loss .

Softmax gradient

In [8]:
```
## 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 impl
emented the gradient correctly.
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 0.885368 analytic: 0.885368, relative error: 1.936488e-09
numerical: 1.273703 analytic: 1.273703, relative error: 4.648494e-09
numerical: 0.276652 analytic: 0.276652, relative error: 8.968999e-09
numerical: 2.746925 analytic: 2.746925, relative error: 1.352587e-08
numerical: 1.513210 analytic: 1.513210, relative error: 2.006139e-08
numerical: 1.693581 analytic: 1.693581, relative error: 9.785640e-09
numerical: -1.057307 analytic: -1.057307, relative error: 2.471551e-08
numerical: -1.241624 analytic: -1.241624, relative error: 2.608807e-10
numerical: 1.283044 analytic: 1.283044, relative error: 9.129776e-09
numerical: -4.001285 analytic: -4.001285, relative error: 6.340778e-09
```

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 [9]:
```
import time
```

In [10]:
```
## 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.nor
m(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 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: 2.334914845802273 / 331.19793844281503 computed in 0.06
498885154724121s
Vectorized loss / grad: 2.33491484580227 / 331.19793844281503 computed in 0.0070
21665573120117s
difference in loss / grad: 3.1086244689504383e-15 /2.2893766446743707e-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:

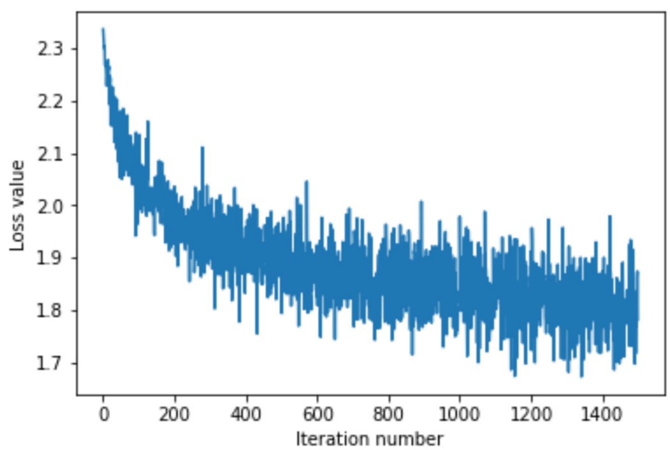
How should the softmax gradient descent training step differ from the svm training step, if at all?

Answer:

Softmax gradient descent training doesn't differ from svm training step.

In [11]:
```
# 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 2.3365926606637544
iteration 100 / 1500: loss 2.0557222613850827
iteration 200 / 1500: loss 2.0357745120662813
iteration 300 / 1500: loss 1.9813348165609888
iteration 400 / 1500: loss 1.9583142443981614
iteration 500 / 1500: loss 1.8622653073541355
iteration 600 / 1500: loss 1.8532611454359382
iteration 700 / 1500: loss 1.8353062223725827
iteration 800 / 1500: loss 1.8293892468827642
iteration 900 / 1500: loss 1.8992158530357484
iteration 1000 / 1500: loss 1.97835035402523
iteration 1100 / 1500: loss 1.8470797913532633
iteration 1200 / 1500: loss 1.8411450268664082
iteration 1300 / 1500: loss 1.7910402495792102
iteration 1400 / 1500: loss 1.8705803029382257
That took 8.079500436782837s
```



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

In [12]:
```
## Implement softmax.predict() and use it to compute the training and testing error
.

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: 0.3811428571428571
validation accuracy: 0.398
```

Optimize the softmax classifier

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

In [13]:
```
np.finfo(float).eps
```

Out[13]:
```
2.220446049250313e-16
```

In [15]:

```
# ================================================================ #
# 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.
# ================================================================ #
valid_accuracy = []
learn_rates = [1e-6, 1e-7, 2e-6, 5e-5]
for i in learn_rates:
    loss_hist = softmax.train(X_train, y_train, learning_rate=i,num_iters=1500, ver
bose=True)
    y_val_pred = softmax.predict(X_val)
    accuracy = np.mean(np.equal(y_val, y_val_pred))
    valid_accuracy.append(accuracy)

print(np.max(valid_accuracy))
print(valid_accuracy)
print(learn_rates[np.argmax(valid_accuracy)])

# We got the best learning rate for 2e-6
softmax.train(X_train, y_train, learning_rate=2e-6, num_iters=1500, verbose=True)

y_test_pred = softmax.predict(X_test)
test_accuracy = np.mean(np.equal(y_test, y_test_pred))

print(1- test_accuracy)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
iteration 0 / 1500: loss 2.352610559430411
iteration 100 / 1500: loss 1.8492610246307748
iteration 200 / 1500: loss 1.8400877106114952
iteration 300 / 1500: loss 1.6573255619166174
iteration 400 / 1500: loss 1.5864093170890907
iteration 500 / 1500: loss 1.852291158311496
iteration 600 / 1500: loss 1.7296631523170733
iteration 700 / 1500: loss 1.7168922194418963
iteration 800 / 1500: loss 1.7911795317419597
iteration 900 / 1500: loss 1.7028199354094893
iteration 1000 / 1500: loss 1.6832881534615014
iteration 1100 / 1500: loss 1.6111143426531287
iteration 1200 / 1500: loss 1.7674338123647495
iteration 1300 / 1500: loss 1.7333371817481167
iteration 1400 / 1500: loss 1.7177939445171801
iteration 0 / 1500: loss 2.3704882876562907
iteration 100 / 1500: loss 1.9615720278025612
iteration 200 / 1500: loss 1.966157039029589
iteration 300 / 1500: loss 1.8930610289041747
iteration 400 / 1500: loss 1.9193743432735466
iteration 500 / 1500: loss 1.8489336263191043
iteration 600 / 1500: loss 1.7616429389971195
iteration 700 / 1500: loss 1.9389588615682165
iteration 800 / 1500: loss 1.8212064159430719
iteration 900 / 1500: loss 1.8434593580954894
iteration 1000 / 1500: loss 1.8234126964890527
iteration 1100 / 1500: loss 1.6828502030460526
iteration 1200 / 1500: loss 1.7951816929127005
iteration 1300 / 1500: loss 1.7686206743770956
iteration 1400 / 1500: loss 1.8648584004789652
iteration 0 / 1500: loss 2.327719807960521
iteration 100 / 1500: loss 1.7122419459455858
iteration 200 / 1500: loss 1.7581548614781355
iteration 300 / 1500: loss 1.6445297395939358
iteration 400 / 1500: loss 1.8122180343327898
iteration 500 / 1500: loss 1.7305869967989957
iteration 600 / 1500: loss 1.7975982134705781
iteration 700 / 1500: loss 1.6141561120277685
iteration 800 / 1500: loss 1.7530084370200194
iteration 900 / 1500: loss 1.6855643585628957
iteration 1000 / 1500: loss 1.643579441811549
iteration 1100 / 1500: loss 1.6230898630888781
iteration 1200 / 1500: loss 1.778178406266838
iteration 1300 / 1500: loss 1.6960540237703572
iteration 1400 / 1500: loss 1.8513025150397333
iteration 0 / 1500: loss 2.3700514725434676
iteration 100 / 1500: loss 12.771400486116494
iteration 200 / 1500: loss 15.418775555700167
iteration 300 / 1500: loss 10.837025397261295
iteration 400 / 1500: loss 11.642940790956752
iteration 500 / 1500: loss 16.36287340512908
iteration 600 / 1500: loss 9.2561446634227
iteration 700 / 1500: loss 18.30656785146666
iteration 800 / 1500: loss 15.41744879014203
iteration 900 / 1500: loss 15.152750329433918
iteration 1000 / 1500: loss 13.282646744133405
iteration 1100 / 1500: loss 16.564419916077306
iteration 1200 / 1500: loss 11.838345013868754
iteration 1300 / 1500: loss 16.664825218925454
iteration 1400 / 1500: loss 15.509610011309519
0.412
[0.392, 0.382, 0.412, 0.308]
2e-06
iteration 0 / 1500: loss 2.34628154698811 3
```

```
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

    # ================================================================ #
    # 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.)
    # ================================================================ #
    for i in range(X.shape[0]):
        scores = X[i].dot(self.W.T)
        scores -= np.max(scores)
        exp_of_scores_sum = np.sum(np.exp(scores))
        softmax = np.exp(scores[y[i]]) / exp_of_scores_sum
        loss -= np.log(softmax)
    loss /= X.shape[0]

    # ================================================================ #
    # 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.
    """

    # Initialize the loss and gradient to zero.
    loss = 0.0
    grad = np.zeros_like(self.W)

```

```python
    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the gradient
    #   as the variable grad.
    # ================================================================ #
    for i in range(X.shape[0]):
        scores = X[i].dot(self.W.T)
        scores -= np.max(scores)
        exp_of_scores_sum = np.sum(np.exp(scores))
        softmax = np.exp(scores[y[i]]) / exp_of_scores_sum
        loss -= np.log(softmax)
        exp_scores = np.exp(scores) / exp_of_scores_sum
        for j in range(self.W.shape[0]):
            if y[i] == j:
                score_change = exp_scores[j] - 1
            else:
                score_change = exp_scores[j]

            grad[j,:] += score_change * X[i]


    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    loss /= X.shape[0]
    grad /= X.shape[0]

    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 softmax loss and gradient WITHOUT any for loops.
    # ================================================================ #
    scores = X.dot(self.W.T)
    scores -= np.max(scores, axis=1, keepdims=True)
    exp_of_scores_sum = np.sum(np.exp(scores), axis=1, keepdims=True)
```

```
    softmaxes = np.exp(scores) / exp_of_scores_sum
    loss = np.sum(-np.log(softmaxes[np.arange(X.shape[0]),y])) / X.shape[0]
    indexes = np.zeros_like(softmaxes)
    indexes[np.arange(X.shape[0]),y] = 1
    grad=X.T.dot(softmaxes - indexes).T
    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.
      # ================================================================ #
      indexes = np.random.choice(num_train, batch_size)
      X_batch = X[indexes]
      y_batch = y[indexes]
      # ================================================================ #
      # 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.
        # ================================================================ #
        y_pred = np.argmax(X.dot(self.W.T), axis=1)
        # ================================================================ #
        # END YOUR CODE HERE
        # ================================================================ #

        return y_pred
```