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 [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-1

# Load matplotlib images inline
%matplotlib inline

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

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

# As a sanity check, we print out the size of the training and test data.
print('Training data shape: ', X_train.shape)
print('Training labels shape: ', y_train.shape) #1*50000 labels
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

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

In [3]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
num_classes = len(classes)
# print(num_classes) # =10
samples_per_class = 7
for y, cls in enumerate(classes): # y indicates the no. of class, and cls
    idxs = np.flatnonzero(y_train == y) # find the positions of elements
    idxs = np.random.choice(idxs, samples_per_class, replace=False) # fin
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

In [4]:
```
# 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 [5]:
```
# Import the KNN class

from nndl import KNN
```

In [6]:
```
# 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)
# print(knn.X_train.shape, knn.y_train.shape)
# print(knn)
# print(X_test[0].shape) = 3072
# print(X_train[0].shape) = 3072
```

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) By calling knn.train(), values of inputs X(=X_train) and y(=y_train) are passed to the instance knn.

(2) Pros: simple, constant time O(1). Cons: Take a large space of memory.

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 [7]:
```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition
#   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(dists_L2.shape)
# print(dists_L2[0][1])
print('Time to run code: {}'.format(time.time()-time_start))
print('Frobenius norm of L2 distances: {}'.format(np.linalg.norm(dists_L2
```

```
Time to run code: 43.79638695716858
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 [8]:
```
# Implement the function compute_L2_distances_vectorized() in the KNN cla
# In this function, you ought to achieve the same L2 distance but WITHOUT
# 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 (shoul
```

```
Time to run code: 0.28377795219421387
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 [9]:
```
# 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

# ================================================================ #
# 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_pred = knn.predict_labels(dists = dists_L2_vectorized)
# print(y_pred.shape)
# print(y_test.shape)
error = np.mean(y_pred != y_test)

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

# ================================================================ #

print(error)
```

```
0.726
```

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

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

Optimizing KNN hyperparameters

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

Create training and validation folds

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

In [10]:
```
# 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]
# ================================================================ #
X_train_folds = np.vsplit(X_train, num_folds)
y_train_folds = np.hsplit(y_train, num_folds)


pass

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

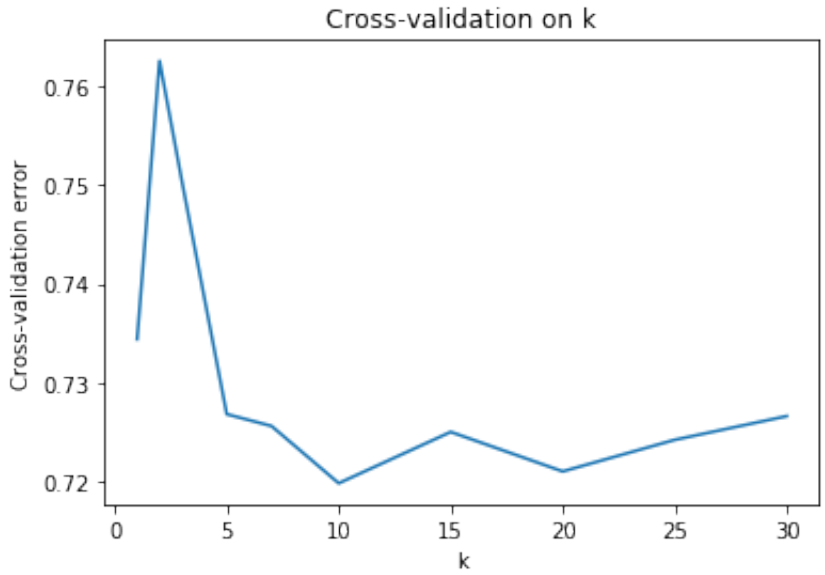
Optimizing the number of nearest neighbors hyperparameter.

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

In [11]:
```
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 = np.zeros(len(ks))

for i in np.arange(num_folds):
    X_train_cv = np.vstack(X_train_folds[:i] + X_train_folds[i+1:])
```

```
    X_train_cv = np.vstack(X_train_folds[:i] + X_train_folds[i+1:])
    X_val_cv = X_train_folds[i]
    y_train_cv = np.hstack(y_train_folds[:i] + y_train_folds[i+1:])
    y_val_cv = y_train_folds[i]

    knn_cv = KNN()
    knn_cv.train(X_train_cv, y_train_cv)
    dists_cv = knn_cv.compute_L2_distances_vectorized(X = X_val_cv)

    for index, K in enumerate(ks):
        y_pred_cv = knn_cv.predict_labels(dists_cv, k=K)
        error = np.mean(y_pred_cv != y_val_cv)
#         print(error)
        errors[index] += error
#     print(errors)
errors_avg = errors / num_folds
print(errors_avg)

plt.plot(ks, errors_avg)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation error')
plt.show()

pass

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

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

```
[0.7344 0.7626 0.7504 0.7268 0.7256 0.7198 0.725  0.721  0.7242 0.7266
]
```



```
Computation time: 16.25
```

Questions:

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

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

Answers:

(1) The best k is 10 amongst the tested k's since it has the smallest cross-validation error.

(2) The cross-validation error for k = 10 is 0.7198.

Optimizing the norm

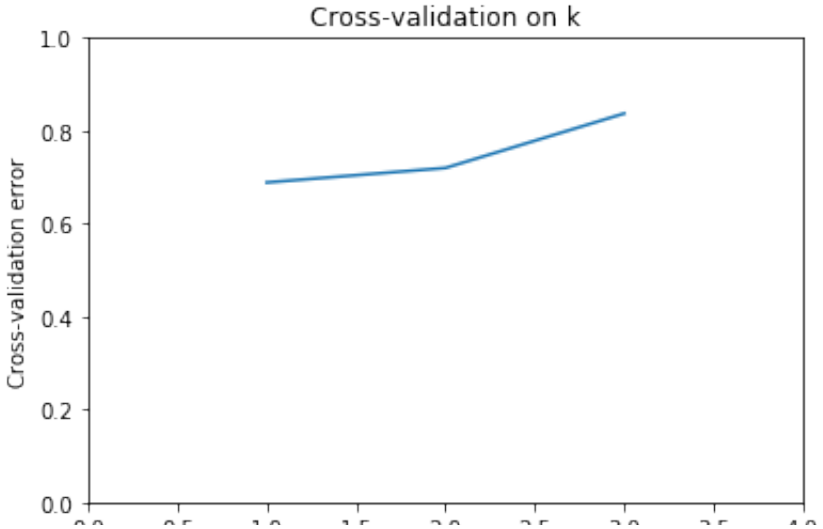
Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

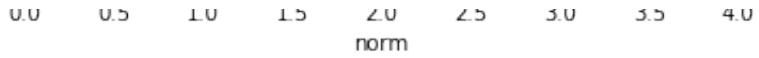
In [12]:

```
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 err
#   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.
# ================================================================ #
pos_k = np.argmin(errors_avg)
chosen_k = ks[pos_k]
print(chosen_k)
errors_norm = np.zeros(len(norms))
for i in np.arange(num_folds):
    X_train_cv = np.vstack(X_train_folds[:i] + X_train_folds[i+1:])
```

```
    X_val_cv = X_train_folds[i]
    y_train_cv = np.hstack(y_train_folds[:i] + y_train_folds[i+1:])
    y_val_cv = y_train_folds[i]

    knn_norm_cv = KNN()
    knn_norm_cv.train(X_train_cv, y_train_cv)

    for j in range(len(norms)):
        dist_L = knn_norm_cv.compute_distances(X = X_val_cv, norm = norms
#         print(dist_L.shape)
        y_pred = knn_norm_cv.predict_labels(dists = dist_L, k = chosen_k)
        error = np.mean(y_pred != y_val_cv)
        errors_norm[j] = errors_norm[j] + error

errors_norm_avg = errors_norm/num_folds

print(errors_norm_avg)

plt.plot([1,2,3], errors_norm_avg)
plt.title('Cross-validation on k')
plt.axis([0, 4, 0, 1])
plt.xlabel('norm')
plt.ylabel('Cross-validation error')
plt.show()
pass

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

# print(dist_L.shape)
```

```
10
[0.6886 0.7198 0.837 ]
```






```
Computation time: 856.88
```

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 gives the best cross-validation error.

(2) Given k = 10 and L1-norm, the cross-validation error is 0.6886.

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 [13]:
```
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.
# ================================================================ #

pos_norm = np.argmin(errors_norm_avg)
chosen_norm = norms[pos_norm]
print(chosen_norm)
print(chosen_k)


knn = KNN()
knn.train(X_train, y_train)
dists_L = knn.compute_distances(X = X_test, norm = chosen_norm)
y_pred = knn.predict_labels(dists_L, k = chosen_k)
# print(y_pred.shape)
# print(y_test.shape)
error = np.mean(y_pred != y_test)
print(error)
pass

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

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

```
<function <lambda> at 0x1084f3048>
10
0.722
Error rate achieved: 0.722
```

Question:

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

Answer:

Previously, the error using k = 1 and L2-norm is 0.726, and currently, by using L1-norm and the chosen k = 10, the error rate is improved to 0.722. Therefore, it improved by (0.726-0.722)/0.726*100%, which is 0.55%.

```
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].
        # ================================================================ #
        dist = norm(X[i] - self.X_train[j])
        dists[i][j] = dist
        pass
```

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

    return dists

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

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

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

    # ========================================================== #
    # YOUR CODE HERE:
    #   Compute the L2 distance between the ith test point and the jth
    #   training point and store the result in dists[i, j].  You may
    #    NOT use a for loop (or list comprehension).  You may only use
    #     numpy operations.
    #
    #     HINT: use broadcasting.  If you have a shape (N,1) array and
    #   a shape (M,) array, adding them together produces a shape (N, M)
    #   array.
    # ========================================================== #

    # Output: sqrt((test_pic - train_pic)^2)
    # (test_pic-train_pic)^2 = test_pic^2 + train_pic^2 - 2*test_pic*train_pic

    test_sum = np.sum(X**2, axis = 1) # shape = (num_test, ) = 500 # adding by rows
    train_sum = np.sum(self.X_train**2, axis = 1) # shape = (num_train, ) = 5000
    test_train = np.dot(X, self.X_train.T) # shape = num_test * num_train
    dists = np.sqrt(test_sum.reshape(-1, 1) + train_sum - 2*test_train) # (N, 1) +
     (M,) - (N, M) array


    pass

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

    return dists
```

```
  def predict_labels(self, dists, k=1):
    """
    Given a matrix of distances between test points and training points,
    predict a label for each test point.

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

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

      # ========================================================== #
      # 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 [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-1
import pdb

# Load matplotlib images inline
%matplotlib inline

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

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

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

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

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

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

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

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

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

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

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

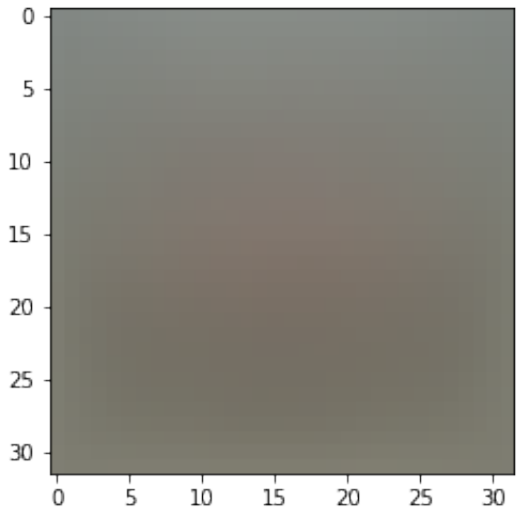
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

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

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

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

In [6]:
```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
# print(mean_image.shape) = (3072, )
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
plt.show()
```

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



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

In [8]:
```
# third: append the bias dimension of ones (i.e. bias trick) so that our
# 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) Generally, we should normalize the data (or perform mean-subtraction on the data) when the scale of a feature is irrelevant or misleading, and not normalize when the scale is meaningful. As for SVM, we perform mean-subtraction to center the data, however for KNN, we use Euclidean distance to measure how far the testing data is from the training data, which indicates the scale is meaningful.

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 [31]:
```
from nndl.svm import SVM
```

In [32]:
```
# 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 us

np.random.seed(1)

num_classes = len(np.unique(y_train)) # = 10
num_features = X_train.shape[1] # = 3072

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

SVM loss

In [33]:
```
## Implement the loss function for in the SVM class(nndl/svm.py), svm.los
# tic = time.time()
loss = svm.loss(X_train, y_train)
print('The training set loss is {}.'.format(loss))
# toc = time.time()
# print ('Naive loss and gradient: computed in %fs' % (toc - tic))
# If you implemented the loss correctly, it should be 15569.98
```

```
The training set loss is 15569.97791541023.
```

SVM gradient

In [34]:
```
## 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)
# print(X_dev[0][0])
# print(loss)
# print(grad)
# print(grad.shape)
# print(np.linalg.norm(grad, 'fro'))
# print(grad[0][0])
# Compare your gradient to a numerical gradient check.
# You should see relative gradient errors on the order of 1e-07 or less i
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -11.246262 analytic: -11.246262, relative error: 2.249944e-
09
numerical: 5.186229 analytic: 5.186229, relative error: 1.630023e-09
numerical: -2.490384 analytic: -2.490384, relative error: 1.205736e-09
numerical: 4.578491 analytic: 4.578491, relative error: 1.036558e-08
numerical: -8.306767 analytic: -8.306767, relative error: 3.932551e-09
numerical: 4.852505 analytic: 4.852505, relative error: 7.428111e-10
numerical: 5.507388 analytic: 5.507389, relative error: 1.428190e-08
numerical: -14.231407 analytic: -14.231407, relative error: 2.042413e-
09
numerical: -7.068918 analytic: -7.068918, relative error: 3.936076e-09
numerical: -13.755380 analytic: -13.755380, relative error: 7.735600e-
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 [35]:
```
import time
```

In [36]:
```
## Implement svm.fast_loss_and_grad which calculates the loss and gradien
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = svm.loss_and_grad(X_dev, y_dev)
# print(X_dev[0][0])
# print(grad.shape)
# print(grad[0][0])
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np.

tic = time.time()
loss_vectorized, grad_vectorized = svm.fast_loss_and_grad(X_dev, y_dev)
# print(grad_vectorized.shape)
# print(grad_vectorized[0][0])
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vecto

# The losses should match but your vectorized implementation should be mu
print('difference in loss / grad: {} / {}'.format(loss - loss_vectorized,

# You should notice a speedup with the same output, i.e., differences on
```

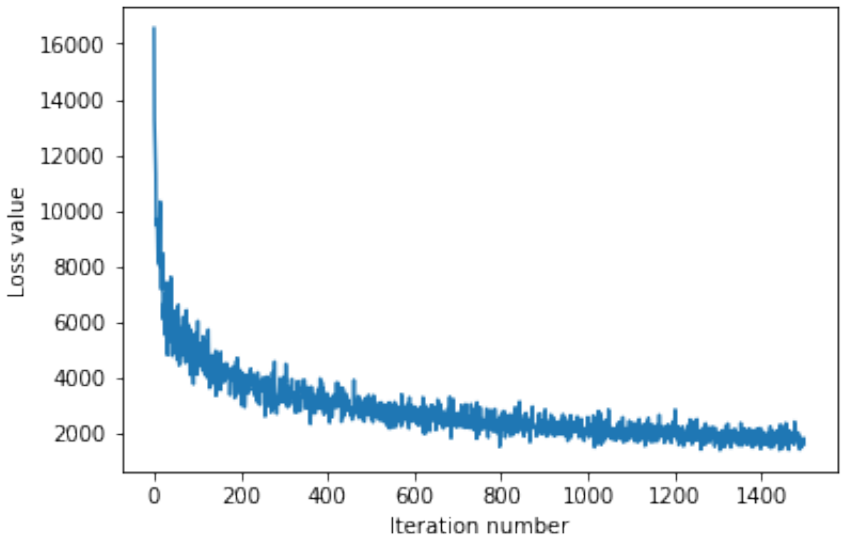
```
Normal loss / grad_norm: 13999.630452635061 / 2048.18940097428 compute
d in 0.05036282539367676s
Vectorized loss / grad: 13999.630452635061 / 2048.1894009742796 comput
ed in 0.004767179489135742s
difference in loss / grad: 0.0 / 2.5154507727668504e-12
```

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 [37]:
```
# 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))
# print(len(loss_hist))
plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 16557.380001909158
iteration 100 / 1500: loss 4701.089451272714
iteration 200 / 1500: loss 4017.333137942788
iteration 300 / 1500: loss 3681.9226471953625
iteration 400 / 1500: loss 2732.616437398899
iteration 500 / 1500: loss 2786.6378424645054
iteration 600 / 1500: loss 2837.0357842782664
iteration 700 / 1500: loss 2206.2348687399326
iteration 800 / 1500: loss 2269.03882411698
iteration 900 / 1500: loss 2543.23781538592
iteration 1000 / 1500: loss 2566.6921357268266
iteration 1100 / 1500: loss 2182.0689059051633
iteration 1200 / 1500: loss 1861.118224425044
iteration 1300 / 1500: loss 1982.9013858528256
iteration 1400 / 1500: loss 1927.5204158582114
That took 3.043776035308838s
```



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

In [38]:
```
## Implement svm.predict() and use it to compute the training and testing

y_train_pred = svm.predict(X_train)
# print(y_train_pred[:10], y_train[:10])
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pre
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 [40]:
```
# ================================================================ #
# 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
#
#   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 = np.array([5e-6, 8e-6, 1e-5, 5e-5, 8e-5, 1e-4, 5e-4, 8e-4
# learning_rates = np.array([7e-7])
# print(lr[0])
best_svm = None
best_val_acc = -1
best_lr = 0

for lr in learning_rates:
    svm = SVM(dims=[num_classes, num_features])
    losses = svm.train(X_train, y_train, learning_rate = lr,
                       num_iters=1500, verbose=True)
    y_train_pred = svm.predict(X_train)
    print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train
    y_val_pred = svm.predict(X_val)
    val_acc = np.mean(np.equal(y_val, y_val_pred))
```

```
    print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_

    if val_acc > best_val_acc:
        best_svm = svm
        best_val_acc = val_acc
        best_lr = lr
print('The best learning rate is {}'.format(lr, ))
print('The best validation accuracy is {} and its corresponding validatio

y_test_pred = best_svm.predict(X_test)
test_acc = np.mean(np.equal(y_test, y_test_pred))
print('testing accuracy: {}'.format(test_acc, ))



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

```
iteration 0 / 1500: loss 15658.054145417169
iteration 100 / 1500: loss 14242.049657996204
iteration 200 / 1500: loss 12999.715353665571
iteration 300 / 1500: loss 10538.180112933615
iteration 400 / 1500: loss 10286.567991053184
iteration 500 / 1500: loss 11864.392203922114
iteration 600 / 1500: loss 8748.991554380966
iteration 700 / 1500: loss 10365.671096386988
iteration 800 / 1500: loss 8974.786133569609
iteration 900 / 1500: loss 8564.14627988812
iteration 1000 / 1500: loss 7841.7792116224955
iteration 1100 / 1500: loss 9025.290033030535
iteration 1200 / 1500: loss 9334.831641553408
iteration 1300 / 1500: loss 8184.371643932415
iteration 1400 / 1500: loss 7184.292310306516
training accuracy: 0.18755102040816327
validation accuracy: 0.206
iteration 0 / 1500: loss 16409.02482607994
iteration 100 / 1500: loss 17161.9866271389
iteration 200 / 1500: loss 11600.392953096214
iteration 300 / 1500: loss 12994.295693589236
iteration 400 / 1500: loss 9685.6131095873
iteration 500 / 1500: loss 9801.677952888918
iteration 600 / 1500: loss 9938.963082617583
iteration 700 / 1500: loss 8585.069789627989
iteration 800 / 1500: loss 8026.595887017292
iteration 900 / 1500: loss 8785.667490177657
iteration 1000 / 1500: loss 8520.481294442534
iteration 1100 / 1500: loss 9886.230559998778
iteration 1200 / 1500: loss 7176.0734298244815
```

```
iteration 1300 / 1500: loss 7643.173475522494
iteration 1400 / 1500: loss 8082.74433363298
training accuracy: 0.19208163265306122
validation accuracy: 0.165
iteration 0 / 1500: loss 16965.92017072196
iteration 100 / 1500: loss 13241.267181836445
iteration 200 / 1500: loss 9782.150495897133
iteration 300 / 1500: loss 9436.614217241558
iteration 400 / 1500: loss 9560.581023682797
iteration 500 / 1500: loss 9421.168462190435
iteration 600 / 1500: loss 9128.983403691296
iteration 700 / 1500: loss 9079.691635210442
iteration 800 / 1500: loss 6727.872424648836
iteration 900 / 1500: loss 7878.181874800411
iteration 1000 / 1500: loss 8319.263663070924
iteration 1100 / 1500: loss 7294.666145203879
iteration 1200 / 1500: loss 7025.540397134533
iteration 1300 / 1500: loss 6072.6697761872165
iteration 1400 / 1500: loss 5943.778634834087
training accuracy: 0.1990204081632653
validation accuracy: 0.219
iteration 0 / 1500: loss 17131.943492505638
iteration 100 / 1500: loss 9409.765300524099
iteration 200 / 1500: loss 7203.865319404643
iteration 300 / 1500: loss 6034.943598654187
iteration 400 / 1500: loss 6745.82004418117
iteration 500 / 1500: loss 5670.455158034281
iteration 600 / 1500: loss 5896.141137483634
iteration 700 / 1500: loss 5138.626257543199
iteration 800 / 1500: loss 4687.057624831159
iteration 900 / 1500: loss 5753.858369064131
iteration 1000 / 1500: loss 4893.345794452343
iteration 1100 / 1500: loss 4559.692621283092
iteration 1200 / 1500: loss 5127.9571601601
iteration 1300 / 1500: loss 4620.760032907103
iteration 1400 / 1500: loss 4538.825331754093
training accuracy: 0.24283673469387754
validation accuracy: 0.239
iteration 0 / 1500: loss 15053.531771301045
iteration 100 / 1500: loss 8548.27538426734
iteration 200 / 1500: loss 6430.751154991212
iteration 300 / 1500: loss 6266.1164642258245
iteration 400 / 1500: loss 4599.613637503997
iteration 500 / 1500: loss 5974.490158454784
iteration 600 / 1500: loss 4660.5231430064605
iteration 700 / 1500: loss 4985.933388183541
iteration 800 / 1500: loss 3825.8806909438044
iteration 900 / 1500: loss 4419.375276787923
iteration 1000 / 1500: loss 3778.805647728728
iteration 1100 / 1500: loss 4314.161554437147
```

```
iteration 1200 / 1500: loss 4178.345836798539
iteration 1300 / 1500: loss 3883.6383150637657
iteration 1400 / 1500: loss 3723.6860896329654
training accuracy: 0.2616734693877551
validation accuracy: 0.289
iteration 0 / 1500: loss 15486.370619706
iteration 100 / 1500: loss 7598.664926819008
iteration 200 / 1500: loss 6464.694178830311
iteration 300 / 1500: loss 6573.470184852069
iteration 400 / 1500: loss 4488.957222967042
iteration 500 / 1500: loss 5312.264469758165
iteration 600 / 1500: loss 5008.3494014087855
iteration 700 / 1500: loss 4226.553441522881
iteration 800 / 1500: loss 4008.37078310916
iteration 900 / 1500: loss 3920.2236802780444
iteration 1000 / 1500: loss 3459.0430244524714
iteration 1100 / 1500: loss 4159.028161737634
iteration 1200 / 1500: loss 3717.538592912142
iteration 1300 / 1500: loss 3271.164998538439
iteration 1400 / 1500: loss 3855.4419267411267
training accuracy: 0.26193877551020406
validation accuracy: 0.262
iteration 0 / 1500: loss 14115.646035437041
iteration 100 / 1500: loss 5016.086890251574
iteration 200 / 1500: loss 3916.7498368166257
iteration 300 / 1500: loss 3360.551087726868
iteration 400 / 1500: loss 3314.097752815921
iteration 500 / 1500: loss 2921.0202953081916
iteration 600 / 1500: loss 2779.798949464384
iteration 700 / 1500: loss 2831.2160545899287
iteration 800 / 1500: loss 2068.6456613162136
iteration 900 / 1500: loss 2339.112428000442
iteration 1000 / 1500: loss 1779.0262248358026
iteration 1100 / 1500: loss 1765.5903230297836
iteration 1200 / 1500: loss 1971.8541271736024
iteration 1300 / 1500: loss 1830.2517370712987
iteration 1400 / 1500: loss 1618.720837519034
training accuracy: 0.28681632653061223
validation accuracy: 0.283
iteration 0 / 1500: loss 15500.773674459997
iteration 100 / 1500: loss 3624.5922360114073
iteration 200 / 1500: loss 4343.961495827747
iteration 300 / 1500: loss 3560.7895022717144
iteration 400 / 1500: loss 2805.5071394041374
iteration 500 / 1500: loss 2306.8103829786837
iteration 600 / 1500: loss 2224.9232990383807
iteration 700 / 1500: loss 2188.916391473069
iteration 800 / 1500: loss 2266.7133723585125
iteration 900 / 1500: loss 2332.4208940134804
iteration 1000 / 1500: loss 2162.282582603412
```

```
iteration 1100 / 1500: loss 1771.2063745628884
iteration 1200 / 1500: loss 1981.022152186711
iteration 1300 / 1500: loss 1644.920880524143
iteration 1400 / 1500: loss 1817.3431195249425
training accuracy: 0.28753061224489795
validation accuracy: 0.291
iteration 0 / 1500: loss 14366.939852858146
iteration 100 / 1500: loss 4097.419944971674
iteration 200 / 1500: loss 4169.394931893793
iteration 300 / 1500: loss 2959.2867961443844
iteration 400 / 1500: loss 2812.2811461175006
iteration 500 / 1500: loss 2057.166534030488
iteration 600 / 1500: loss 2302.0978656105453
iteration 700 / 1500: loss 2208.333129706169
iteration 800 / 1500: loss 2577.666764598992
iteration 900 / 1500: loss 2118.8357795914653
iteration 1000 / 1500: loss 2188.4634790977493
iteration 1100 / 1500: loss 2157.654618869853
iteration 1200 / 1500: loss 1957.6248272100063
iteration 1300 / 1500: loss 1941.4776528681127
iteration 1400 / 1500: loss 1851.4610202592694
training accuracy: 0.29475510204081634
validation accuracy: 0.286
iteration 0 / 1500: loss 17376.25774264116
iteration 100 / 1500: loss 8478.345173905172
iteration 200 / 1500: loss 9984.016614609951
iteration 300 / 1500: loss 10648.755516124982
iteration 400 / 1500: loss 11807.343750031527
iteration 500 / 1500: loss 7923.3802144688625
iteration 600 / 1500: loss 7964.524670984784
iteration 700 / 1500: loss 8457.783570224037
iteration 800 / 1500: loss 11117.136942419613
iteration 900 / 1500: loss 7942.514939660153
iteration 1000 / 1500: loss 9728.265369219092
iteration 1100 / 1500: loss 7475.149334345299
iteration 1200 / 1500: loss 8774.045743536513
iteration 1300 / 1500: loss 10006.357867329512
iteration 1400 / 1500: loss 10995.244866043186
training accuracy: 0.3222857142857143
validation accuracy: 0.31
iteration 0 / 1500: loss 13671.730311763002
iteration 100 / 1500: loss 13568.61472717571
iteration 200 / 1500: loss 18858.707093058416
iteration 300 / 1500: loss 15017.082958133842
iteration 400 / 1500: loss 9663.565128809954
iteration 500 / 1500: loss 20711.68307613329
iteration 600 / 1500: loss 21258.101155363034
iteration 700 / 1500: loss 10449.750647941044
iteration 800 / 1500: loss 14288.761509318589
iteration 900 / 1500: loss 16999.314728781927
```

```
iteration 1000 / 1500: loss 9742.530917681514
iteration 1100 / 1500: loss 15853.653217545927
iteration 1200 / 1500: loss 17213.626105314437
iteration 1300 / 1500: loss 16620.277121523613
iteration 1400 / 1500: loss 10166.822256093983
training accuracy: 0.2734489795918367
validation accuracy: 0.27
The best learning rate is 0.008
The best validation accuracy is 0.31 and its corresponding validation
error is 0.69
testing accuracy: 0.312
```

In [ ]:

```
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
    hinges = np.zeros(num_train)

    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.)
    # ============================================================= #
        hinge = 0
        for j in np.arange(num_classes):
            if j == y[i]:
                continue
            hinge_per_j =  1 + np.dot(self.W[j], X[i]) - np.dot(self.W[y[i]], X[i])
```

```
            if hinge_per_j > 0:
                hinge += hinge_per_j
        hinges[i] = hinge

    loss = np.sum(hinges)/num_train

    pass
    # ============================================================= #
    # 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]
#     print(X.shape) : 500*3073
#     print(self.W.shape) : 10*3073
    loss = 0.0
    grad = np.zeros_like(self.W)

    hinges = np.zeros(num_train)

    for i in np.arange(num_train):
    # ============================================================= #
    # YOUR CODE HERE:
    #   Calculate the SVM loss and the gradient.  Store the gradient in
    #   the variable grad.
    # ============================================================= #

        hinge = 0
        for j in np.arange(num_classes):
            if j == y[i]:
                continue
            hinge_per_j =  1 + np.dot(self.W[j], X[i]) - np.dot(self.W[y[i]], X[i])

            if hinge_per_j > 0:
                hinge += hinge_per_j
                grad[j] += X[i]
                grad[y[i]] -= X[i]

        hinges[i] = hinge


    loss = np.sum(hinges)
```

```
    pass

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

    loss /= num_train
    grad /= num_train

    return loss, grad

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

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

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

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

  def fast_loss_and_grad(self, X, y):
    """
    A vectorized implementation of loss_and_grad. It shares the same
    inputs and ouptuts as loss_and_grad.
    """
    # compute the loss and the gradient
    num_classes = self.W.shape[0]
    num_train = X.shape[0]
    loss = 0.0
    grad = np.zeros(self.W.shape) # initialize the gradient as zero
#     print(X.shape) : 500*3073
#     print(self.W.shape) : 10*3073
    # ============================================================= #
    # YOUR CODE HERE:
    #   Calculate the SVM loss WITHOUT any for loops.
    # ============================================================= #
    scores = np.dot(self.W, X.T)   #  scores.shape = 10*500
    correct_scores = np.ones(scores.shape) * scores[y, np.arange(0,
     scores.shape[1])]
    margin = 1 + scores - correct_scores
```

```
    margins = np.maximum(0, margin)
    margins[y, np.arange(0, scores.shape[1])] = 0 # y[i] = j position to 0
    loss = np.sum(margins)

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



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

    margins_copy = margins
#     print(margins_copy.shape) = 10*500
    margins_copy[margins > 0] = 1
    margins_copy[margins < 0] = 0
    margins_copy[y, np.arange(0, scores.shape[1])] = 0
    margins_copy[y, np.arange(0, scores.shape[1])] = -1*np.sum(margins_copy, axis =
     0)
    grad = np.dot(margins_copy, X)

    # ============================================================= #
    # END YOUR CODE HERE
    # ============================================================= #
    loss /= num_train
    grad /= num_train
    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 # dim = 3072
    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.
        # ========================================================== #
        idx = np.random.choice(num_train, batch_size)
        X_batch = X[idx]
        y_batch = y[idx]
        # ========================================================== #
        # END YOUR CODE HERE
        # ========================================================== #

        # evaluate loss and gradient
        loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
#         loss, grad = self.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.
    # ================================================================ #
#     print(self.W.shape)
    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 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 [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 [2]:
```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=10
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prep
    it for the linear classifier. These are the same steps as we used for
    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]
```

```
    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_CIFAR1
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 [3]:
```
from nndl import Softmax
```

In [4]:
```
# 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

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 [5]:
```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

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

In [6]:
```
print(loss)
```

```
2.3277607028048757
```

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 have already normlized the data and the inital softmax classifier favors each class equally, the loss returned should be ~log(num_classes).

Softmax gradient

In [7]:
```
## 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 the
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = softmax.loss_and_grad(X_dev,y_dev)
# loss, grad = softmax.loss_and_grad(X_train,y_train)
# print(loss)

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

```
numerical: 0.398346 analytic: 0.398346, relative error: 3.517939e-08
numerical: -0.057680 analytic: -0.057680, relative error: 5.681091e-07
numerical: -1.167630 analytic: -1.167630, relative error: 1.616099e-08
numerical: 0.706282 analytic: 0.706281, relative error: 7.127798e-08
numerical: 0.421403 analytic: 0.421402, relative error: 8.744608e-08
numerical: 2.197265 analytic: 2.197265, relative error: 1.091924e-08
numerical: 0.279155 analytic: 0.279155, relative error: 2.508954e-07
numerical: -0.616502 analytic: -0.616502, relative error: 6.510035e-08
numerical: 1.314428 analytic: 1.314428, relative error: 7.188993e-09
numerical: -3.702145 analytic: -3.702145, relative error: 1.568336e-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 [8]:
```
import time
```

In [9]:
```
## Implement softmax.fast_loss_and_grad which calculates the loss and gra
#    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.

tic = time.time()
loss_vectorized, grad_vectorized = softmax.fast_loss_and_grad(X_dev, y_de
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vecto

# The losses should match but your vectorized implementation should be mu
print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized,

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

```
Normal loss / grad_norm: 2.3339288796696986 / 320.57262196944913 compu
ted in 0.16233110427856445s
Vectorized loss / grad: 2.3339288796697 / 320.57262196944913 computed
in 0.0158689022064209s
difference in loss / grad: -1.3322676295501878e-15 /2.8103635486604383
e-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:

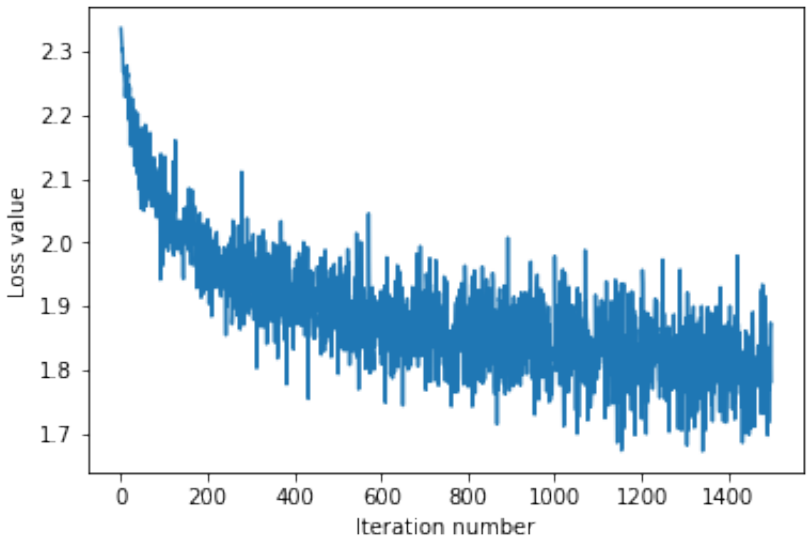
How should the softmax gradient descent training step differ from the svm training step, if at all?

Answer:

Actually there is no difference between softmax gradient descent training step and the svm training step. Both of them are updating the weights by subtracting the gradients multiplied by the learning rate. However, there exists a difference between their gradient function. For svm, when the label is correct (j = y[i]), the gradient descent of it is just X[i], and when the label is incorrect, the gradient descent is -X[i]. However for softmax, it is like to calculate a condition probability (p) on each X[i], then if the label is correct, the gradient descent is (p-1)X[i], and if it is incorrect, the gradient descent will be p*X[i].

In [10]:
```
# Implement softmax.train() by filling in the code to extract a batch of
# and perform the gradient step.
import time


tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7, num_iters
# loss_hist = softmax.train(X_train, y_train, learning_rate=5e-7, num_ite
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 1.835306222372583
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.8992158530357484
iteration 1000 / 1500: loss 1.9783503540252299
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 12.67657995223999s
```



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

In [11]:
```
## Implement softmax.predict() and use it to compute the training and tes

y_train_pred = softmax.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pre
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 [12]:
```
np.finfo(float).eps
```

Out[12]:
```
2.220446049250313e-16
```

In [13]:
```
# ================================================================ #
# 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
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
learning_rates = np.array([7e-7, 1e-6, 4e-6, 7e-6, 1e-5, 4e-5, 7e-5])

best_softmax = None
best_val_acc = -1
best_lr = 0

for lr in learning_rates:
    softmax = Softmax(dims=[num_classes, num_features])
    losses = softmax.train(X_train, y_train, learning_rate = lr,
                      num_iters=1500, verbose=True)
    y_train_pred = softmax.predict(X_train)
    print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train
    y_val_pred = softmax.predict(X_val)
    val_acc = np.mean(np.equal(y_val, y_val_pred))
```

```
    print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_

    if val_acc > best_val_acc:
        best_softmax = softmax
        best_val_acc = val_acc
        best_lr = lr
print('The best learning rate is {}'.format(lr, ))
print('The best validation accuracy is {} and its corresponding validatio

y_test_pred = best_softmax.predict(X_test)
test_acc = np.mean(np.equal(y_test, y_test_pred))
print('testing accuracy: {}'.format(test_acc, ))



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

```
iteration 0 / 1500: loss 2.3647455566656226
iteration 100 / 1500: loss 1.7895958179038585
iteration 200 / 1500: loss 1.7683277130834936
iteration 300 / 1500: loss 1.838658622343926
iteration 400 / 1500: loss 1.817211364117334
iteration 500 / 1500: loss 1.7015302844759808
iteration 600 / 1500: loss 1.752257123113929
iteration 700 / 1500: loss 1.8358793374061855
iteration 800 / 1500: loss 1.8166590744536828
iteration 900 / 1500: loss 1.8122447397395467
iteration 1000 / 1500: loss 1.5755750355461282
iteration 1100 / 1500: loss 1.6795700274481435
iteration 1200 / 1500: loss 1.772104885974278
iteration 1300 / 1500: loss 1.7883418458233162
iteration 1400 / 1500: loss 1.7304260831823877
training accuracy: 0.41746938775510206
validation accuracy: 0.408
iteration 0 / 1500: loss 2.4021441573551026
iteration 100 / 1500: loss 1.8350557682218942
iteration 200 / 1500: loss 1.8500911414944952
iteration 300 / 1500: loss 1.822562814641484
iteration 400 / 1500: loss 1.7362610000911636
iteration 500 / 1500: loss 1.632724842221387
iteration 600 / 1500: loss 1.741192868375939
iteration 700 / 1500: loss 1.7105161206348083
iteration 800 / 1500: loss 1.7763574005831406
iteration 900 / 1500: loss 1.6656420604264717
iteration 1000 / 1500: loss 1.8009178789683227
iteration 1100 / 1500: loss 1.6875567651068213
iteration 1200 / 1500: loss 1.7327007403888408
```

```
iteration 1300 / 1500: loss 1.8576128822480065
iteration 1400 / 1500: loss 1.8452184563054466
training accuracy: 0.4177959183673469
validation accuracy: 0.417
iteration 0 / 1500: loss 2.331228877278063
iteration 100 / 1500: loss 1.788273851695311
iteration 200 / 1500: loss 1.7464719551706238
iteration 300 / 1500: loss 1.805693485140309
iteration 400 / 1500: loss 1.7601206529852218
iteration 500 / 1500: loss 1.909601946275884
iteration 600 / 1500: loss 1.7402858805900119
iteration 700 / 1500: loss 1.782950951380771
iteration 800 / 1500: loss 1.7036318565188777
iteration 900 / 1500: loss 1.5391680329835185
iteration 1000 / 1500: loss 1.798695198692907
iteration 1100 / 1500: loss 1.815714091251636
iteration 1200 / 1500: loss 1.6152719560415247
iteration 1300 / 1500: loss 1.8079648779103832
iteration 1400 / 1500: loss 1.6045247431428533
training accuracy: 0.4084081632653061
validation accuracy: 0.379
iteration 0 / 1500: loss 2.32982808647334
iteration 100 / 1500: loss 2.0084662313206683
iteration 200 / 1500: loss 2.079507224909775
iteration 300 / 1500: loss 2.351430080415239
iteration 400 / 1500: loss 2.3319968607119286
iteration 500 / 1500: loss 1.873352297761005
iteration 600 / 1500: loss 1.844645269173414
iteration 700 / 1500: loss 1.9031083296540061
iteration 800 / 1500: loss 1.9655154084523345
iteration 900 / 1500: loss 2.0568005022309266
iteration 1000 / 1500: loss 1.7650636881295305
iteration 1100 / 1500: loss 1.6819870674176918
iteration 1200 / 1500: loss 1.9596063535403434
iteration 1300 / 1500: loss 2.1668979454398127
iteration 1400 / 1500: loss 2.0094836889541057
training accuracy: 0.3425918367346939
validation accuracy: 0.324
iteration 0 / 1500: loss 2.3513886692503307
iteration 100 / 1500: loss 3.1478513657837515
iteration 200 / 1500: loss 3.267136912291495
iteration 300 / 1500: loss 3.072345433838428
iteration 400 / 1500: loss 2.536594587155592
iteration 500 / 1500: loss 2.7572891534713277
iteration 600 / 1500: loss 2.4319050147283456
iteration 700 / 1500: loss 3.833296547068013
iteration 800 / 1500: loss 3.804956421316697
iteration 900 / 1500: loss 2.855061328521807
iteration 1000 / 1500: loss 2.946975481449491
iteration 1100 / 1500: loss 2.4120322189815138
```

```
iteration 1200 / 1500: loss 2.9626942298947974
iteration 1300 / 1500: loss 2.9693062309965184
iteration 1400 / 1500: loss 2.4267529908755106
training accuracy: 0.34916326530612246
validation accuracy: 0.33
iteration 0 / 1500: loss 2.4432746460289296
iteration 100 / 1500: loss 9.61018702035632
iteration 200 / 1500: loss 8.005908726793542
iteration 300 / 1500: loss 12.33982943107916
iteration 400 / 1500: loss 12.978447087251993
iteration 500 / 1500: loss 9.803076737241955
iteration 600 / 1500: loss 10.361817623902203
iteration 700 / 1500: loss 10.569011114255977
iteration 800 / 1500: loss 10.643609885548862
iteration 900 / 1500: loss 9.767047745673263
iteration 1000 / 1500: loss 15.337638751806058
iteration 1100 / 1500: loss 11.611471097628073
iteration 1200 / 1500: loss 13.10893384939679
iteration 1300 / 1500: loss 9.628758516567421
iteration 1400 / 1500: loss 7.557085070114904
training accuracy: 0.3154489795918367
validation accuracy: 0.331
iteration 0 / 1500: loss 2.337055619175014
iteration 100 / 1500: loss 17.209318676554066
iteration 200 / 1500: loss 15.61893876899142
iteration 300 / 1500: loss 14.415547976680195
iteration 400 / 1500: loss 14.615521748806886
iteration 500 / 1500: loss 18.59092410264801
iteration 600 / 1500: loss 23.615464000792368
iteration 700 / 1500: loss 20.578560825637222
iteration 800 / 1500: loss 21.13015109438893
iteration 900 / 1500: loss 19.142806157881633
iteration 1000 / 1500: loss 14.995445305670877
iteration 1100 / 1500: loss 21.990219469963154
iteration 1200 / 1500: loss 18.494082304350176
iteration 1300 / 1500: loss 24.607680667951435
iteration 1400 / 1500: loss 16.18395865037472
training accuracy: 0.30948979591836734
validation accuracy: 0.296
The best learning rate is 7e-05
The best validation accuracy is 0.417 and its corresponding validation
error is 0.583
testing accuracy: 0.389
```

In [ ]:

```
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.)
    # ============================================================= #
    pass
    num_classes = self.W.shape[0]
    num_train = X.shape[0]

    losses = np.zeros(num_train)
    for i in np.arange(num_train):
        mar = 0
        for j in np.arange(num_classes):
            mar += np.exp(np.dot(self.W[j], X[i].T))
        margin = np.log(mar) - np.dot(self.W[y[i]], X[i].T)
        losses[i] = margin

    loss = np.sum(losses)/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.
    """

    # 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.
    # ============================================================= #
    pass
    num_classes = self.W.shape[0]
    num_train = X.shape[0]
#     print(self.W.shape) = 10*3073
#     print(X.shape) = 500*3073
    losses = 0.0
    mars = np.zeros(num_train)
    for i in np.arange(num_train):
        mar = 0
        for j in np.arange(num_classes):
            mar += np.exp(np.dot(self.W[j], X[i].T))
        margin = np.log(mar) - np.dot(self.W[y[i]], X[i].T)

        losses += margin

        for j in np.arange(num_classes):
            grad[j] += np.dot(np.exp(np.dot(self.W[j], X[i].T)), X[i].T)/mar
#             print(mar)
            if j == y[i]:
                grad[j] -= X[i].T


    loss = losses/num_train
    grad = grad/num_train
    # ============================================================= #
    # END YOUR CODE HERE
    # ============================================================= #

    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.
    # ============================================================= #
    pass

    num_classes = self.W.shape[0]
    num_train = X.shape[0]
#     print(self.W.shape) = 10*3073
#     print(X.shape) = 500*3073
    losses = 0.0
    mars = np.zeros(num_train)
    scores = np.dot(self.W, X.T)

    mars = np.sum(np.exp(scores), axis = 0)
    margin = np.log(mars) - scores[y, np.arange(0, scores.shape[1])]
    losses = np.sum(margin)
#     print(mars.shape) = (500,)

#     print(np.exp(scores).shape)
#     print(scores.shape)
#     print((np.exp(scores)/mars).shape)
```

```
#     print(X.shape)
    grad = np.dot(np.exp(scores)/mars, X)

    # elminate X[i] for j == y[i]
    eliminate = np.zeros((num_classes, num_train))
    eliminate[y, np.arange(0,scores.shape[1])] = 1
    grad -= np.dot(eliminate, X)

    loss = losses/num_train
    grad = 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):
    """
    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.
        # ================================================================ #
        idx = np.random.choice(num_train, batch_size)
        X_batch = X[idx]
#         print(X_batch.shape)
        y_batch = y[idx]
#         print(y_batch.shape)
        pass
        # ================================================================ #
        # 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
        # ================================================================ #
        pass
        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.
    # ================================================================ #
    pass
    scores = np.dot(self.W, X.T)
    y_pred = np.argmax(scores, axis=0)
```

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

return y_pred
```