C247 HW2

Zhuohao Li

zhuohaol@ucla.edu

1. (a) $\hat{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \theta^T(x^{(i)} + \delta^{(i)}))^2$
   $\iff \hat{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}[(y^{(i)} - \theta^T x^{(i)})^2 - 2(y^{(i)} - \theta^T x^{(i)})\theta^T\delta^{(i)} + (\theta^T\delta^{(i)})^2]$
   The expectation $E_{\delta\sim\mathbf{N}}[\hat{L}(\theta)] = \frac{1}{N}\sum_{i=1}^{N}[(y^{(i)} - \theta^T x^{(i)})^2 + E_{\delta\sim\mathbf{N}}(\theta^T\delta^{(i)})^2]$ bacause $y^{(i)} - \theta^T x^{(i)}$ is independent of δ.
   $(\theta^T\delta^{(i)})^2 = \theta^T\delta^{(i)}\delta^{(i)T}\theta$ so $E_{\delta\sim\mathbf{N}}(\theta^T\delta^{(i)})^2 = \theta^T(\sigma^2 I)\theta$
   $E_{\delta\sim\mathbf{N}}[\hat{L}(\theta)] = L(\theta) + \sigma^2\theta^T\theta$

   (b) The regulation is similar to L2 norm. This term penalizes larger values of the weights θ.

   (c) The variance of the added noise decreases, there is no noise added to the feature. Risk of overfitting.

   (d) The variance of the added noise increases, dominating the weights of the feature. Risk of underfitting.

2. check the code.

3. Log likelihood:
   $L(\theta) = \sum_{j=1}^{m} logPr(y^{(i)}|x^j, \theta) = w_i^T x^{(j)} + b_i - log(\sum_{k=1}^{c} e^{w_k^T x^{(j)} + b_k})$
   Let $a_i^{(j)} = w_i^T x^{(j)} + b_i$, $1\{y^{(j)} = i\} = 1$ only when $y^{(j)} = i$ and otherwise equals to 0.
   Gradient for w_i:

   $$\nabla_{w_i} L = \sum_{j=1}^{m}(x^{(j)}(1\{y^{(j)} = i\} - \frac{e^{a_i^{(j)}}}{\sum_{k=1}^{c} e^{a_k^{(j)}}})$$

   Gradient for b_i:

   $$\nabla_{b_i} L = \sum_{j=1}^{m}((1\{y^{(j)} = i\} - \frac{e^{a_i^{(j)}}}{\sum_{k=1}^{c} e^{a_k^{(j)}}})$$

   Unified Gradient:
   Let $\hat{x}^{(j)} = [x^{(j)}, 1]^T, \hat{w}_i = [w_i, b_i]^T$

   $$\nabla_{\hat{w}_i} L = \sum_{j=1}^{m}(\hat{x}^{(j)}(1\{y^{(j)} = i\} - \frac{e^{\hat{w}_i^T\hat{x}^{(j)}}}{\sum_{k=1}^{c} e^{\hat{w}_i^T\hat{x}^{(j)}}}))$$

4. $\nabla_w L = \frac{1}{K}\sum_{i=1}^{K}\nabla_w max(0, 1 - y^{(i)}(w^T x^{(i)} + b))$
   when $1 - y^{(i)}(w^T x^{(i)} + b) > 0$, $\nabla_w L = -y^{(i)}x^{(i)}$
   otherwise, the gradient is zero. So the unified gradient is $\nabla_w L = \frac{1}{K}\sum_{i=1}^{K} -y^{(i)}x^{(i)}1\{1 - y^{(i)}(w^T x^{(i)} + b) > 0\}$
   Similarly, the gradient of b is $\nabla_b L = \frac{1}{K}\sum_{i=1}^{K} -y^{(i)}1\{1 - y^{(i)}(w^T x^{(i)} + b) > 0\}$

5. check the code.

This is the k-nearest neighbors workbook for ECE C147/C247 Assignment #2

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

Please print out the workbook entirely when completed.

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

# 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
```

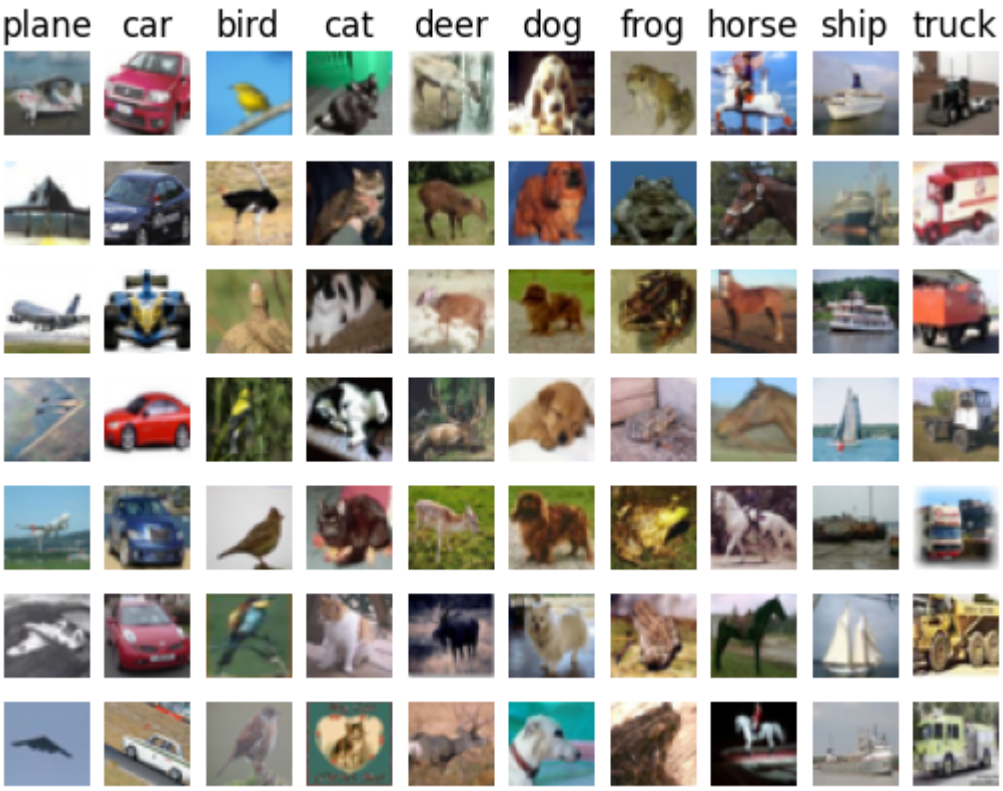
```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

In [ ]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = './cifar-10-batches-py' # You need to update this line
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

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

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

from nndl import KNN
```

In [ ]:
```
# 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) It just did the training and inference concurrently, put the labels with the input images.

(2) Pros: simply. Cons: slow in backward and extreme high overheads in memory and computation

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 [ ]:
```
# 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('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: 33.028862953186035
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 [ ]:
```
# 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: 0.15729999542236328
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 [ ]:
```
# 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.
# ================================================================ #
k = 1
predicted_labels = knn.predict_labels(dists_L2, k)

num_incorrect = np.sum(predicted_labels != y_test)

error = num_incorrect / y_test.shape[0]

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

print(error)
```

```
0.726
```

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

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

Optimizing KNN hyperparameters

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

Create training and validation folds

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

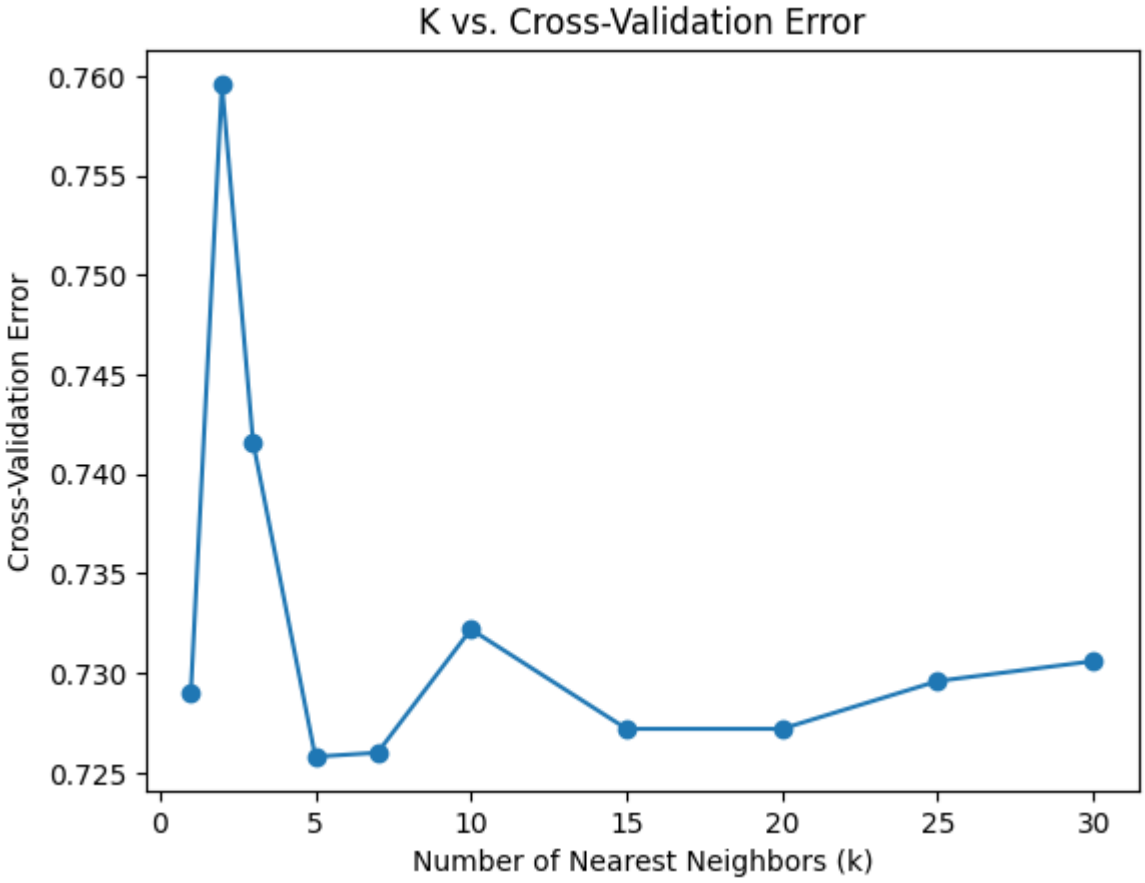
In [ ]:
```
# Create the dataset folds for cross-valdiation.
num_folds = 5

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]
# ================================================================ #
num_examples = X_train.shape[0]
shuffle_indices = np.random.permutation(num_examples)
X_train_shuffled = X_train[shuffle_indices]
y_train_shuffled = y_train[shuffle_indices]

fold_size = num_examples // num_folds
X_train_folds = np.array_split(X_train_shuffled, num_folds, axis=0)
y_train_folds = np.array_split(y_train_shuffled, num_folds, axis=0)




# ================================================================ #
# 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 [ ]:
```
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 = []

k_to_accuracies = {}

for k in ks:
    fold_errors = []
    for fold in range(num_folds):
        # Form the training and validation sets for this fold
        X_val_current = X_train_folds[fold]
        y_val_current = y_train_folds[fold]
        X_train_current = np.concatenate(np.delete(X_train_folds, fold, a
        y_train_current = np.concatenate(np.delete(y_train_folds, fold, a

        # Train KNN and predict labels
        knn = KNN()
        knn.train(X_train_current, y_train_current)
        dists_current = knn.compute_L2_distances_vectorized(X_val_current
        y_val_pred = knn.predict_labels(dists_current, k)

        # Compute and store the error for this fold
        num_val = y_val_current.shape[0]
        fold_error = np.sum(y_val_pred != y_val_current)
        fold_errors.append(fold_error / num_val)

    # Average the error across all folds and store it
    cross_validation_errors.append(np.mean(fold_errors))

# Plotting
plt.plot(ks, cross_validation_errors, marker='o')
plt.xlabel('Number of Nearest Neighbors (k)')
plt.ylabel('Cross-Validation Error')
plt.title('K vs. Cross-Validation Error')
plt.show()

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

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




```
Computation time: 25.32
```

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) 5

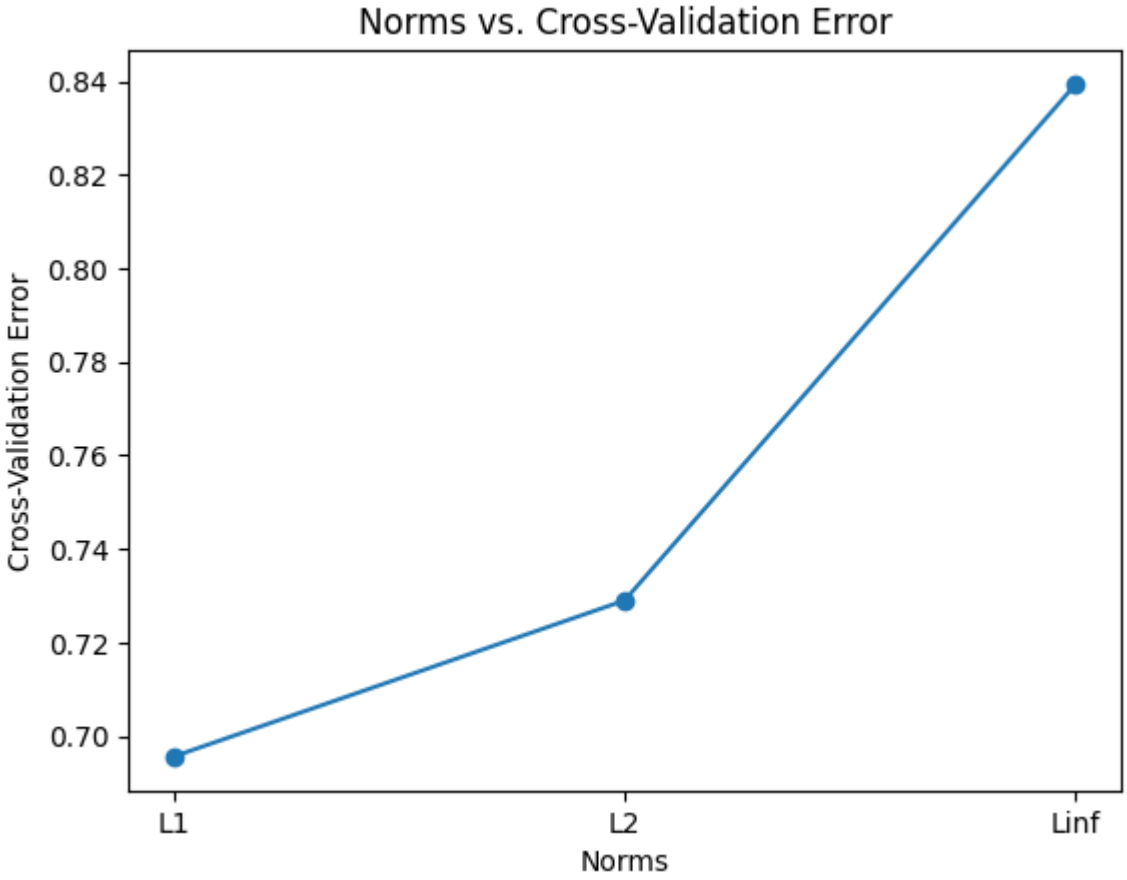
(2) About 0.726

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 [ ]:
```
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.
# ================================================================ #
cross_validation_errors = []
norm_labels = ['L1', 'L2', 'Linf']

for norm in norms:
    fold_errors = []
    for fold in range(num_folds):
        # Form the training and validation sets for this fold
        X_val_current = X_train_folds[fold]
        y_val_current = y_train_folds[fold]
        X_train_current = np.concatenate(np.delete(X_train_folds, fold, a
        y_train_current = np.concatenate(np.delete(y_train_folds, fold, a

        knn = KNN()
        knn.train(X_train_current, y_train_current)
        dists_current = knn.compute_distances(X_val_current, norm)
        # if norm == L1_norm:
        #     dists_current = knn.compute_L1_distances_vectorized(X_val_c
        # elif norm == L2_norm:
        #     dists_current = knn.compute_L2_distances_vectorized(X_val_c
        # else:
        #     dists_current = knn.compute_Linf_distances_vectorized(X_val

        y_val_pred = knn.predict_labels(dists_current, 5)

        num_val = y_val_current.shape[0]
        fold_error = np.sum(y_val_pred != y_val_current)
        fold_errors.append(fold_error / num_val)

    cross_validation_errors.append(np.mean(fold_errors))

# Plotting
plt.plot(norm_labels, cross_validation_errors, marker='o')
plt.xlabel('Norms')
plt.ylabel('Cross-Validation Error')
plt.title('Norms vs. Cross-Validation Error')
plt.show()



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




```
Computation time: 603.29
```

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

(2) About 0.695

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 [ ]:
```
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.
```

```
# ================================================================ #

k = 5
norm = L1_norm

knn = KNN()
knn.train(X_train, y_train)
dists = knn.compute_distances(X_test, norm)
y_test_pred = knn.predict_labels(dists, k)

num_test = y_test.shape[0]
num_error = np.sum(y_test_pred != y_test)
error = float(num_error) / num_test


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

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

```
Error rate achieved: 0.698
```

Question:

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

Answer:

About 4% (from 72.6% to 68.8%)

KNN implementation

In [ ]:

```
import numpy as np
import pdb


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 p
```

```
    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,
      is the Euclidean distance between the ith test point and the jth tr
      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 t
                #   training point using norm(), and store the result in
                # ================================================================

                diff = X[i] - self.X_train[j]
                dists[i, j] = norm(diff)

                # ================================================================
                # 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 p
    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,
      is the Euclidean distance between the ith test point and the jth tr
      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 jt
        #   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,
        #   array.
        # ================================================================

        dists = np.reshape(np.sum(X**2, axis=1), [num_test, 1]) + np.sum(
            self.X_train**2, axis=1) - 2 * np.dot(X, self.X_train.T)

        dists = np.sqrt(dists)

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

        return dists

    def predict_labels(self, dists, k):
        """
    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,
      gives the distance between the ith test point and the jth training

    Returns:
    - y: A numpy array of shape (num_test,) containing predicted labels f
      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 neig
            # the ith test point.
            closest_y = []
            # ========================================================
            # YOUR CODE HERE:
            #   Use the distances to calculate and then store the labels
            #   the k-nearest neighbors to the ith test point.  The funct
            #   numpy.argsort may be useful.
            #
            #   After doing this, find the most common label of the k-nea
            #   neighbors.  Store the predicted label of the ith training
            #   as y_pred[i].  Break ties by choosing the smaller label.
            # ========================================================

            closest_idxs = np.argsort(dists[i])[0:k]
            closest_y = self.y_train[closest_idxs]

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

        return y_pred

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

        return y_pred

    def compute_L1_distances_vectorized(self, X):
```

```
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))

        dists = np.sum(np.abs(X[:, np.newaxis] - self.X_train), axis=2)

        return dists

    def compute_Linf_distances_vectorized(self, X):
        num_test = X.shape[0]
        num_train = self.X_train.shape[0]
        dists = np.zeros((num_test, num_train))

        dists = np.max(np.abs(X[:, np.newaxis] - self.X_train), axis=2)

        return dists
```

This is the softmax workbook for ECE C147/C247 Assignment #2

Please follow the notebook linearly to implement a softmax classifier.

Please print out the workbook entirely when completed.

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

In [ ]:
```
import random
import numpy as np
from utils.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 [ ]:
```
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/' # You need to update this line
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

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

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

```
    X_dev -= mean_image

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

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


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev = get_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 [ ]:
```
from nndl import Softmax
```

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

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

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

```
2.3277607028048757
```

Question:

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

Answer:

At this place, we calculate the loss based on random weights, it expected that the initial loss has to be close to -log(0.1) because initially all the classes are equally likely to be chosen. CIFAR-10 has 10 classes, the probability of the correct class will be 1/10 = 0.1.

Softmax gradient

In [ ]:
```
## 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)

# 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: -1.874057 analytic: -1.874057, relative error: 1.263239e-08
numerical: 1.387515 analytic: 1.387515, relative error: 2.624873e-09
numerical: -0.567321 analytic: -0.567321, relative error: 2.466017e-08
numerical: 1.851400 analytic: 1.851400, relative error: 2.119540e-08
numerical: -0.740015 analytic: -0.740015, relative error: 6.777993e-08
numerical: 0.345675 analytic: 0.345675, relative error: 4.726264e-08
numerical: 0.954805 analytic: 0.954805, relative error: 5.123092e-08
numerical: -1.092516 analytic: -1.092516, relative error: 2.377785e-08
numerical: 0.374384 analytic: 0.374384, relative error: 3.802900e-08
numerical: -3.277800 analytic: -3.277800, relative error: 1.537155e-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 [ ]:
```
import time
```

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

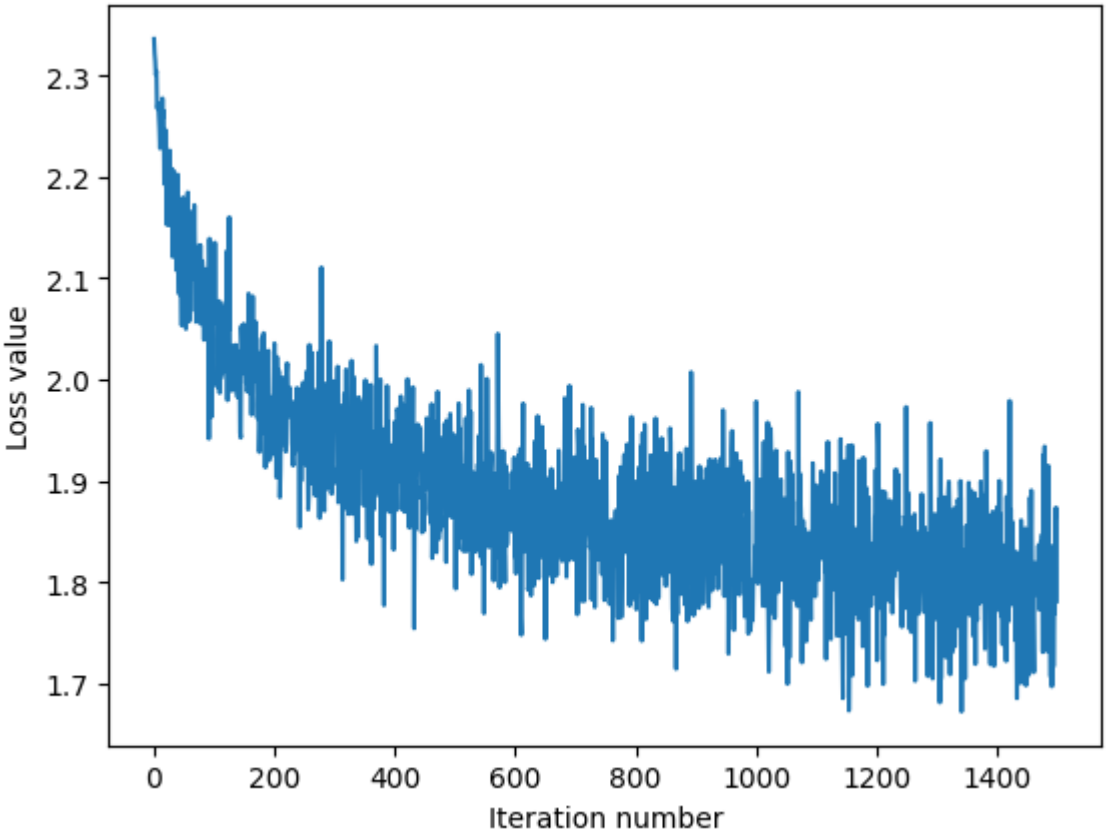
```
Normal loss / grad_norm: 2.320651464777037 / 341.63022045708004 computed i
n 0.009704828262329102s
Vectorized loss / grad: 2.320651464777037 / 341.63022045708004 computed in
0.003211975097656625s
difference in loss / grad: 0.0 /0.0
```

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 [ ]:
```
# 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=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 1.9583142443981612
iteration 500 / 1500: loss 1.862265307354135
iteration 600 / 1500: loss 1.8532611454359382
iteration 700 / 1500: loss 1.8353062223725827
iteration 800 / 1500: loss 1.829389246882764
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 1.79104024957921
iteration 1400 / 1500: loss 1.8705803029382257
That took 3.5548441410064697s
```



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

In [ ]:
```
## 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

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

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

In [ ]:
```
# ================================================================ #
# 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 = [1e-3, 5e-4, 1e-4, 5e-5, 1e-5]
best_lr = None
best_val_accuracy = -1
best_softmax = None

for lr in learning_rates:
    softmax = Softmax(dims=[num_classes, X_train.shape[1]])  # num_classe
    loss_history = softmax.train(X_train, y_train, learning_rate=lr, num_

    # Predict on the validation set
    y_val_pred = softmax.predict(X_val)
    val_accuracy = np.mean(y_val_pred == y_val)
    print(f'Learning rate: {lr}, Validation accuracy: {val_accuracy}')

    # Update best validation accuracy and best learning rate
    if val_accuracy > best_val_accuracy:
        best_val_accuracy = val_accuracy
        best_lr = lr
        best_softmax = softmax

print(f'Best learning rate: {best_lr}')
print(f'Best validation accuracy: {best_val_accuracy}')

# Evaluate the best softmax on test data
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test_pred == y_test)
test_error_rate = 1 - test_accuracy
print(f'Test error rate: {test_error_rate}')



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

```
Learning rate: 0.001, Validation accuracy: 0.297
Learning rate: 0.0005, Validation accuracy: 0.329
Learning rate: 0.0001, Validation accuracy: 0.274
Learning rate: 5e-05, Validation accuracy: 0.257
Learning rate: 1e-05, Validation accuracy: 0.33
Best learning rate: 1e-05
Best validation accuracy: 0.33
Test error rate: 0.6859999999999999
```

Softmax Implementation

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 minib
    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
      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 varia
        #   (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.)
        # ============================================================
        # dW = np.zeros_like(self.W)
        scores = X.dot(self.W.T)
        scores -= np.max(scores, axis=1, keepdims=True)  # safe softmax
        exp_scores = np.exp(scores)
        pr = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
        correct_log_pr = -np.log(pr[np.arange(X.shape[0]), y])
        loss = np.sum(correct_log_pr) / 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)

    # ============================================================
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the gradie
    #   as the variable grad.
    # ============================================================
    scores = X.dot(self.W.T)
    scores -= np.max(scores, axis=1, keepdims=True)  # safe softmax
    exp_scores = np.exp(scores)
    pr = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)
    correct_log_pr = -np.log(pr[np.arange(X.shape[0]), y])
    loss = np.sum(correct_log_pr) / X.shape[0]
    dscores = pr
    dscores[np.arange(X.shape[0]), y] -= 1
    dscores /= X.shape[0]
    grad = dscores.T.dot(X)

    return loss, grad

    # ============================================================
    # 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 ouputs as loss_and_grad.
    """
        loss = 0.0
        grad = np.zeros(self.W.shape)  # initialize the gradient as zero

        # ============================================================
        # YOUR CODE HERE:
        #   Calculate the softmax loss and gradient WITHOUT any for loops
        # ============================================================
        # Compute scores
        scores = X.dot(self.W.T)
        scores -= np.max(scores, axis=1,
                         keepdims=True)  # for numerical stability
        exp_scores = np.exp(scores)
        probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

        # Compute loss
        correct_log_probs = -np.log(probs[np.arange(X.shape[0]), y])
        loss = np.sum(correct_log_probs) / X.shape[0]

        # Compute gradient
        dscores = probs
        dscores[np.arange(X.shape[0]), y] -= 1
        dscores /= X.shape[0]
        grad = dscores.T.dot(X)

        # ============================================================
        # 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 ar
      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 st
    - verbose: (boolean) If true, print progress during optimization.

    Outputs:
    A list containing the value of the loss function at each training ite
    """
        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
            #     gradient descent.  After sampling,
            #     - X_batch should have shape: (batch_size, dim)
            #     - y_batch should have shape: (batch_size,)
            #   The indices should be randomly generated to reduce correl
            #   in the dataset.  Use np.random.choice.  It's okay to samp
            #   replacement.
            # ========================================================
            indices = np.random.choice(np.arange(num_train), batch_size)
            X_batch = X[indices]
            y_batch = y[indices]
            # ========================================================
            # END YOUR CODE HERE
            # ========================================================

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

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

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

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

        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-dimension
      array of length N, and each element is an integer giving the predic
      class.
    """
        y_pred = np.zeros(X.shape[1])
        # ========================================================
```

```
        # YOUR CODE HERE:
        #   Predict the labels given the training data.
        # ================================================================
        scores = X.dot(self.W.T)
        y_pred = np.argmax(scores, axis=1)
        # ================================================================
        # END YOUR CODE HERE
        # ================================================================

        return y_pred
```