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 [318]:

```
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 CIFA
R-10 dataset.

# Load matplotlib images inline
%matplotlib inline

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

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

In [319]:
```
# 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 da
ta.
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 [320]:
```
# 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', 'hors
e', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y) #return indices of the non-zer
o elements of the input array
    idxs = np.random.choice(idxs, samples_per_class, replace=False) #g
enerates a random sample from a given 1D array
    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 [321]:
```
# Subsample the data for more efficient code execution in this exercis
e
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 [322]:
```
# Import the KNN class

from nndl import KNN
```

In [323]:
```
# 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(np.array_equal(knn.compute_L2_distances_vectorized(X_test), kn
n.compute_distances(X_test)))
#knn.compute_L2_distances_vectorized(X_test)

#knn.predict_labels(dists=knn.compute_L2_distances_vectorized(X_test))
```

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) Function knn.train() takes two objects as inputs: a matrix of X_train as the training data self.X_train, and the corresponding labels y_train as the self.y_train to be called in the functions.

(2) Pros: no training time and thus simple. Cons: requires caching the entire training set, which could be impractical if large, is computationally expensive on testing new data, the curse of dimensionality may be at play and the data representation is very important.

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 [324]:
```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definit
ion of the norm
#   in the code, which is the 2-norm.
# You should only have to fill out the clearly marked sections.

import time
time_start =time.time()

dists_L2 = knn.compute_distances(X=X_test)

print('Time to run code: {}'.format(time.time()-time_start))
print('Frobenius norm of L2 distances: {}'.format(np.linalg.norm(dists
_L2, 'fro')))
```

```
Time to run code: 57.70186114311218
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 [325]:
```
# Implement the function compute_L2_distances_vectorized() in the KNN class.
# In this function, you ought to achieve the same L2 distance but WITHOUT any for loops.
# Note, this is SPECIFIC for the L2 norm.

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

```
Time to run code: 0.4430050849914551
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 [326]:

```
# Implement the function predict_labels in the KNN class.
# Calculate the training error (num_incorrect / total_samples)
#   from running knn.predict_labels with k=1

error = 1

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the error rate by calling predict_labels on the test
#   data with k = 1.  Store the error rate in the variable error.
# ================================================================ #
y_test_pred = knn.predict_labels(dists=knn.compute_L2_distances_vector
ized(X_test))
#print(y_test_pred)
#print(y_test)
error -= float(np.sum(y_test_pred == y_test))/num_test

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

print(error)
```

```
0.726
```

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

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

Optimizing KNN hyperparameters

In this section, we'll take the KNN classifier that you have constructed and perform cross-validation to choose a best value of k, as well as a best choice of norm. In k-fold cross-validation, the original sample is randomly partitioned into k equal size subsamples. Of the k subsamples, a single subsample is retained as the validation data for testing the model, and the remaining k-1 subsamples are used as training data. The cross-validation process is then repeated k times (the folds), with each of the k subsamples used exactly once as the validation data. The k results from the folds can then be averaged (or otherwise combined) to produce a single estimation. The advantage of this method is that all observations are used for both training and validation, and each observation is used for validation exactly once.

Create training and validation folds

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

In [236]:

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

X_train_folds = []
y_train_folds = []

# ================================================================== #
# YOUR CODE HERE:
#   Split the training data into num_folds (i.e., 5) folds.
#   X_train_folds is a list, where X_train_folds[i] contains the
#      data points in fold i.
#   y_train_folds is also a list, where y_train_folds[i] contains
#      the corresponding labels for the data in X_train_folds[i]
# ================================================================== #
cv_idx = np.arange(num_training)
np.random.shuffle(cv_idx)
ind = np.array_split(cv_idx, num_folds)
for i in ind:
    X_train_folds.append(X_train[i])
    y_train_folds.append(y_train[i])
    #print(X_train_folds)
    #print(y_train_folds)

#X_train_folds_noshuffle = np.array_split(X_train, num_folds)
# ================================================================== #
# END YOUR CODE HERE
# ================================================================== #
```

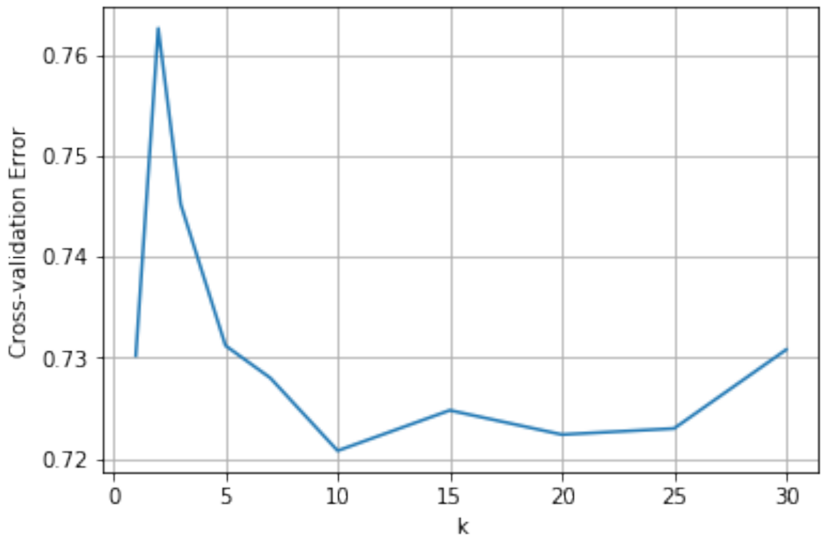
Optimizing the number of nearest neighbors hyperparameter.

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

In [237]:

```
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.
# ================================================================ #
k_error = {}
e = []
for k in ks:
    k_error[k] = []
    for f in range(num_folds):
        X_test = X_train_folds[f]
        y_test = y_train_folds[f]
        X_train = np.concatenate(X_train_folds[:f] + X_train_folds[(f+
1):])
        y_train = np.concatenate(y_train_folds[:f] + y_train_folds[(f+
1):])

        knn.train(X_train, y_train)

        dists = knn.compute_L2_distances_vectorized(X_test)
        y_test_pred = knn.predict_labels(dists, k=k)

        num_correct = np.sum(y_test_pred == y_test)
        error = 1 - float(num_correct) / X_test.shape[0]
        k_error[k].append(error)
#for k in k_error:
    #for avg_error in k_error[k]:
        #print (k,avg_error)
    avg_error = float(np.sum(k_error[k])/num_folds)
    print ('k =', k, 'Average error=', avg_error)
    e.append(avg_error)

plt.plot(ks, e, label = 'k-fold cross validation error')
plt.xlabel('k')
plt.ylabel('Cross-validation Error')
plt.grid()
plt.show()


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

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

```
k = 1 Average error= 0.7302000000000001
k = 2 Average error= 0.7626
k = 3 Average error= 0.7452
k = 5 Average error= 0.7312
k = 7 Average error= 0.728
k = 10 Average error= 0.7208
k = 15 Average error= 0.7248
k = 20 Average error= 0.7224
k = 25 Average error= 0.7230000000000001
k = 30 Average error= 0.7308
```




```
Computation time: 47.39
```

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) k = 10 is the best among the tested k's, with the lowest cross-validation error rate if I do not shuffle the training data prior to splitting. If shuffled/used randomization, according to the results above, the best k can vary. In the example above, still k = 10.

(2) 0.7198 for k=10 without shuffling data (#np.random.shuffle), 0.7208 for k =10 in the run above based on shuffled training data.

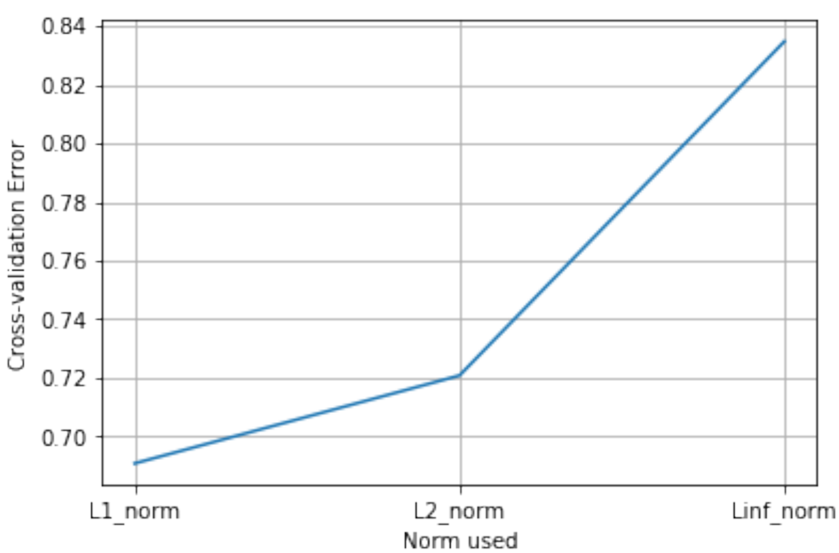
Optimizing the norm

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 [304]:
```
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, testi
ng
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of the norm used vs the cross-validation
error
#   Use the best cross-validation k from the previous part.
#
#   Feel free to use the compute_distances function.  We're testing ju
st
#   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- nor
ms
#   to speed this up, but it is not necessary.
# ================================================================ #
Error_avg = []
for norm in norms:
    errorsum = 0
    t = time.time()

    for f in range(num_folds):
        X_test = X_train_folds[f]
        y_test = y_train_folds[f]
        X_train = np.concatenate(X_train_folds[:f] + X_train_folds[(f+
1):])
        y_train = np.concatenate(y_train_folds[:f] + y_train_folds[(f+
1):])

        knn.train(X_train, y_train)
        dists = knn.compute_distances(X_test, norm)
        y_test_pred = knn.predict_labels(dists, k = 10)

        num_correct = np.sum(y_test_pred == y_test)
        error = 1 - (num_correct / X_test.shape[0])
        errorsum += error

    avg_error = errorsum/num_folds
```

```
    Error_avg.append(avg_error)
    print ('Average error: %s (%.2f seconds)' % (avg_error, time.time(
)-t))

print('Total time: %.2f seconds' % (time.time()-time_start))

norms_name = ['L1_norm','L2_norm','Linf_norm']
plt.plot(norms_name, Error_avg, label = 'k-fold cross validation error
')
plt.xlabel('Norm used')
plt.ylabel('Cross-validation Error')
plt.grid()
plt.show()

pass

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

```
Average error: 0.6908 (365.02 seconds)
Average error: 0.7208 (286.98 seconds)
Average error: 0.8348000000000001 (369.30 seconds)
Total time: 1021.30 seconds
```




```
Computation time: 1021.63
```

Questions:

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

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

Answers:

(1) L1_norm has the best cross-validation error.

(2) Cross-validation error is 0.6908 for the given norm and k = 10.

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 [327]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Evaluate the testing error of the k-nearest neighbors classifier
#   for your optimal hyperparameters found by 5-fold cross-validation.
# ================================================================ #
error = 1
optimal_k = 10
knn.train(X_train, y_train)
print (X_train.shape)
print (X_test.shape)

dists = knn.compute_distances(X = X_test, norm = L1_norm)
print(dists.shape)
y_test_pred = knn.predict_labels(dists, k = optimal_k)

error -= (np.sum(y_test_pred == y_test))/X_test.shape[0]

pass

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

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

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

Improved by around 0.01 in my case.