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

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

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with the data, training and evaluating a simple classifier, k-fold cross validation, and as a Python refresher.

Import the appropriate libraries

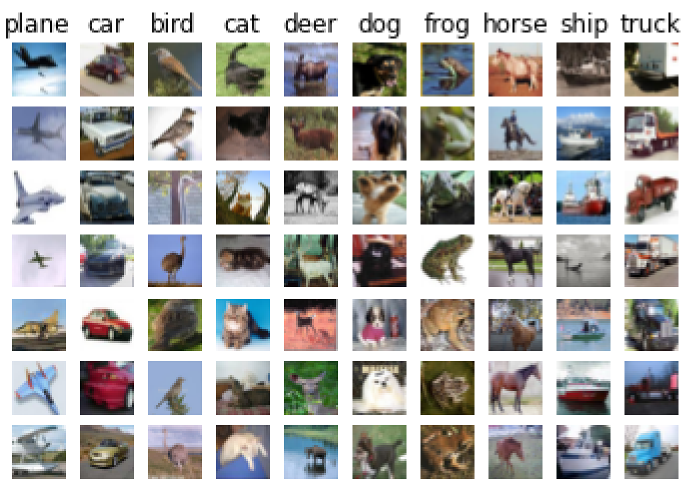
In [1]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.data_utils import load_CIFAR10 # function to load the CIFAR-10 dat
aset.

# 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-ipyt
hon
%load_ext autoreload
%autoreload 2
```

In [2]:
```
np.random.seed(123)
```

In [3]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = r'C:\Users\lpott\Desktop\UCLA\ECENGR247C-80\HW2\cifar-10-batches
-py' # You need to update this line
X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

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

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

In [4]:
```python
# 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', 'shi
p', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```



In [5]:
```python
# 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 [6]:
```
# Import the KNN class

from nndl import KNN
```

In [7]:
```
# 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) All of the training data (images and labels) is loaded into the class variables.

(2) The pros are that the training complexity is O(1), but the cons are that the memory is intensive because we need to store all the data and time complexity is O(n) (these scale with the data).

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 [8]:
```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition of t
he 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, 'fr
o')))
```

```
Time to run code: 32.51972723007202
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 [9]:
```
# 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: 0.16555547714233398
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 [10]:
```
# 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.
# ================================================================ #
error = np.mean((np.array(knn.predict_labels(dists_L2_vectorized,k=1)) != y_te
st))

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.

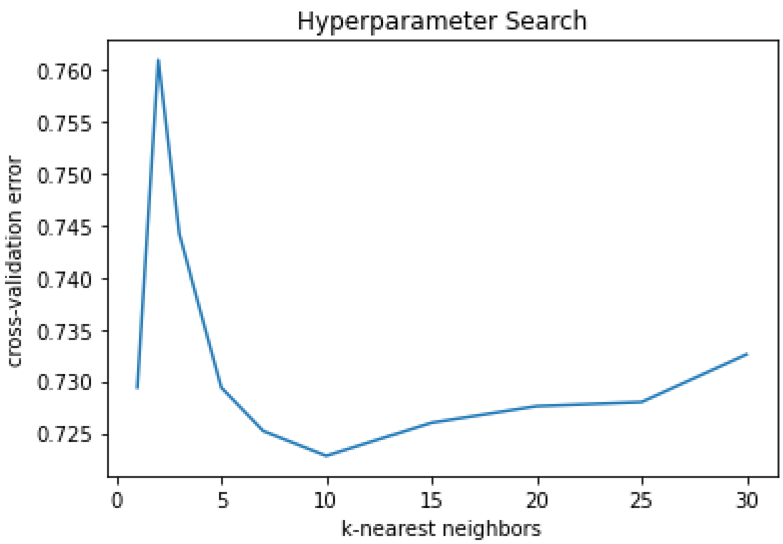
In [11]:
```
# Create the dataset folds for cross-valdiation.
num_folds = 5

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(X_train.shape[0])
np.random.shuffle(cv_idx)
fold_size = X_train.shape[0] // 5
for i in np.arange(num_folds):
    index = cv_idx[i*fold_size:(i+1)*fold_size]
    X_train_folds.append(X_train[index,:])
    y_train_folds.append(y_train[index])
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 [12]:
```
time_start =time.time()

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

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each k in ks, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of k vs. cross-validation error. Since
#   we are assuming L2 distance here, please use the vectorized code!
#   Otherwise, you might be waiting a long time.
# ================================================================ #
errors = []
for j,k in enumerate(ks):
    cv_error = []
    for i in range(num_folds):
        Xtr_cv = np.vstack(X_train_folds[:i] + X_train_folds[i+1:])
        ytr_cv = np.hstack(y_train_folds[:i] + y_train_folds[i+1:])

        Xte_cv = X_train_folds[i]
        yte_cv = y_train_folds[i]

        knn.train(Xtr_cv,ytr_cv)

        dists_L2 = knn.compute_L2_distances_vectorized(X=Xte_cv)

        cv_error.append(np.mean(np.array(knn.predict_labels(dists_L2,k)) != yt
e_cv))

    errors.append(np.mean(cv_error))
    print("k={}, error={}".format(k,errors[j]))

plt.plot(ks,errors)
plt.xlabel("k-nearest neighbors")
plt.ylabel("cross-validation error")
plt.title("Hyperparameter Search")
pass

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

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

```
k=1, error=0.7293999999999999
k=2, error=0.761
k=3, error=0.7442000000000001
k=5, error=0.7293999999999999
k=7, error=0.7252
k=10, error=0.7228000000000001
k=15, error=0.726
k=20, error=0.7276
k=25, error=0.728
k=30, error=0.7326
Computation time: 24.65
```



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) From the plot $k = 10$ is the best among the tests k's

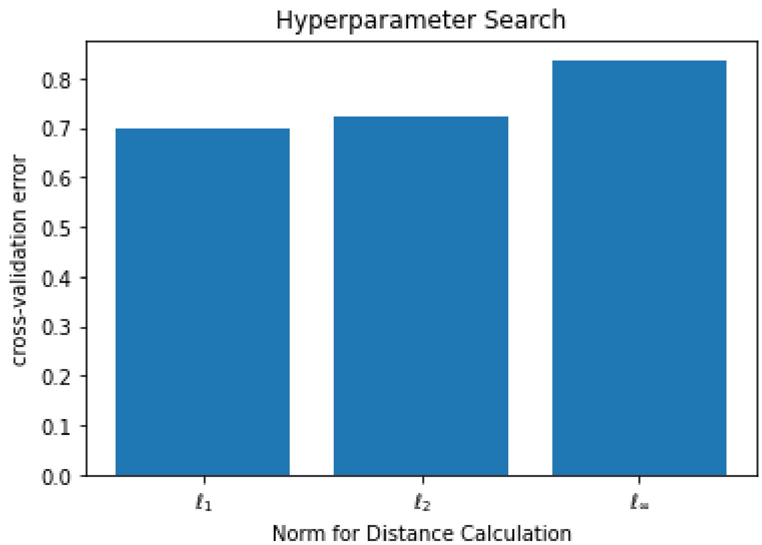
(2) The cross-validation error for $k = 10$ is 0.72280

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 [14]:
```
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]
k_best = 10
# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each norm in norms, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of the norm used vs the cross-validation error
#   Use the best cross-validation k from the previous part.
#
#   Feel free to use the compute_distances function.  We're testing just
#   three norms, but be advised that this could still take some time.
#   You're welcome to write a vectorized form of the L1- and Linf- norms
#   to speed this up, but it is not necessary.
# ================================================================ #
errors = []
axis_labels = ["$\ell_1$","$\ell_2$","$\ell_\infty$"]
for j,norm in enumerate(norms):
    cv_error = []
    for i in range(num_folds):
        Xtr_cv = np.vstack(X_train_folds[:i] + X_train_folds[i+1:])
        ytr_cv = np.hstack(y_train_folds[:i] + y_train_folds[i+1:])

        Xte_cv = X_train_folds[i]
        yte_cv = y_train_folds[i]

        knn.train(Xtr_cv,ytr_cv)

        dists_L2 = knn.compute_distances(X=Xte_cv,norm=norm)

        cv_error.append(np.mean(np.array(knn.predict_labels(dists_L2,k_best))
!= yte_cv))

    errors.append(np.mean(cv_error))
    print("norm={}, error={}".format(axis_labels[j],errors[j]))

plt.bar(np.arange(len(norms)),errors)
plt.xlabel("Norm for Distance Calculation")
plt.ylabel("cross-validation error")
plt.xticks(ticks=np.arange(len(axis_labels)),labels=axis_labels)
plt.title("Hyperparameter Search")

pass

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

```
norm=$\ell_1$, error=0.6988000000000001
norm=$\ell_2$, error=0.7228000000000001
norm=$\ell_\infty$, error=0.8364
Computation time: 642.51
```



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) The ℓ_1 norm had the best cross-validation error.

(2) The best cross-validation error for k_{best}=10 and ℓ_1 norm is 0.6988

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 [15]:
```
error = 1

# ================================================================ #
# YOUR CODE HERE:
#   Evaluate the testing error of the k-nearest neighbors classifier
#   for your optimal hyperparameters found by 5-fold cross-validation.
# ================================================================ #
knn.train(X_train,y_train)
dists = knn.compute_distances(X_test,norm=L1_norm)
error = np.mean((np.array(knn.predict_labels(dists,k=10)) != y_test))
pass

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

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

```
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:

The error had a 0.550964% decrease, which is very minimal...

knn.py

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

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

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth trainin
g
      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 jt
h
        #   training point using norm(), and store the result in dists[i, j].
        # ================================================================ #
        dists[i,j]= norm(self.X_train[j,:] - X[i,:])
        pass

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

    return dists
```

In [ ]:
```
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 trainin
g
      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.
        # ================================================================ #
    dists = np.sqrt(-2*np.dot(X,self.X_train.T) + np.sum(self.X_train**2,1) +
np.sum(X**2,1)[:,np.newaxis])
    pass

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

    return dists
```

In [ ]:
```
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 th
e
      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 exampl
e
          #   as y_pred[i].  Break ties by choosing the smaller label.
          # ================================================================ #
      closest_k_neighbors = self.y_train[np.argsort(dists[i,:])[:k]]
      y_pred[i] = np.argmax(np.bincount(closest_k_neighbors))

      pass

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

    return y_pred
```