k-Nearest Neighbor (kNN) exercise

Complete and hand in the completed notebook (including the output) with your assignment submission. You will be submitting the homework as a zip file including all the parts on the Blackboard. The kNN classifier consists of two stages:

- During training, the classifier takes the training data and simply remembers it
- During testing, kNN classifies every test image by comparing to all training images and transferring the labels of the k most similar training examples
- The value of k is cross-validated

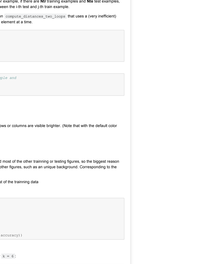
In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

In [1]:
```
# Run some setup code for this notebook.
%matplotlib inline
import random
import numpy as np
import matplotlib.pyplot as plt

# This is a bunch of magic to make matplotlib figures appear inline in the notebook
# rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figure(5) = 10.0, 8.0 # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# These more magic in that the notebook will reload external python modules:
%load_ext autoreload
%autoreload 2
```

In [2]:
```
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/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', 'ship', 'truck']
num_classes = len(classes)
num_samples = 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]:
```
# 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]
```

In [5]:
```
# 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)
```

In [6]:
```
from cs231n.classifiers import KNearestNeighbor

# Create a kNN classifier instance.
# Remember that training a kNN classifier is a noop:
# the Classifier simply remembers the data and does no further processing
classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

Lets begin with computing the distance matrix between all training and test examples. For example, there are **M** training examples and **N** test examples, this stage should result in a **M x N** matrix where each element *(i,j)* is the distance between the *i*th test and *j*th train example.

First, open `cs231n/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and calculates the distance matrix one element at time.

In [7]:
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

# Test your implementation:
dists = classifier.compute_distances_two_loops(X_test)
print(dists.shape)
```

```
(500, 5000)
```

In [8]:
```
# We can visualize the distance matrix: each row is a single test example and
# its distances to training examples
plt.imshow(dists, interpolation='none')
plt.show()
```


Inline Question #1: Notice the structured patterns in the distance matrix, where some rows or columns are visible brighter. (Note that with the default color scheme black indicates low distances while white indicates high distances.)

- What is the data is the cause behind the distinctly bright rows?
- What causes the columns?

Your Answer: Because the bright lines indicate a large distance between one picture and most of the other training or testing figures, so the biggest reason for that is that one picture contains some distinct patterns that are not shared by all the other figures, such as the unique background. Corresponding to the very bright rows.

- The bright rows are caused by "extreme" figure in test data that is different from most of the training data
- The bright columns are caused by "extreme" figure in training data

In [9]:
```
# Now implement the function predict_labels and run the code below:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists, k=1)

# Compute and print the fraction of correctly predicted examples
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f', % (num_correct, num_test, accuracy))
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

You should expect to see approximately `27%` accuracy. Now let's try out a larger `k`, say `k = 5`:

In [10]:
```
y_test_pred = classifier.predict_labels(dists, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f', % (num_correct, num_test, accuracy))
```

```
Got 141 / 500 correct => accuracy: 0.282000
```

You should expect to see a slightly better performance than with `k = 1`.

Inline Question 2

We can also use other distance metrics such as L1 distance. For pixel values $p_{ij}^{(k)}$ at location (i,j) of some image I_k

the mean μ across all pixels over all images is

$$\mu = \frac{1}{nhw}\sum_{k}\sum_{i}^{h}\sum_{j}^{w} p_{ij}^{(k)}$$

And the pixel-wise mean μ_{ij} across all images is

$$\mu_{ij} = \frac{1}{n}\sum_{k}^{n} p_{ij}^{(k)}.$$

The general standard deviation σ and pixel-wise standard deviation σ_{ij} is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply.

1. Subtracting the mean μ ($p_{ij}^{(k)} - \mu$).
2. Subtracting the per-pixel mean μ_{ij} ($p_{ij}^{(k)} - \mu_{ij}$).
3. Subtracting the mean μ and dividing by the standard deviation σ.
4. Subtracting the per-pixel mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}.
5. Rotating the coordinate axes of the data.

Your Answer: Only 5 will not change the performance

Your Explanation: 3 and 4 change every pixel with different amount, so the total L1 distance changes different amount for different images. 1 and 2 change all the pixel value with the same number, but after the norm() operation, the change for every pixel can be different. Only the rotation will not change the total L1 distance.

In [11]:
```
# Now speed up distance matrix computation by using partial vectorization
# with one loop. Implement the function compute_distances_one_loop and run the
# code below:
dists_one = classifier.compute_distances_one_loop(X_test)

# To ensure that our vectorized implementation is correct, we make sure that it
# agrees with the naive implementation. There are many ways to decide whether
# two matrices are similar; one of the simplest is the Frobenius norm. In case
# you haven't seen it before, the Frobenius norm of two matrices is the square
# root of the squared sum of differences of all elements; in other words, reshape
# the matrices into vectors and compute the Euclidean distance between them.
difference = np.linalg.norm(dists - dists_one, ord='fro')
print('Difference was: %f ', % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

In [12]:
```
# Now implement the fully vectorized version inside compute_distances_no_loops
# and run the code
dists_two = classifier.compute_distances_no_loops(X_test)

# check that the distance matrix agrees with the one we computed before:
difference = np.linalg.norm(dists - dists_two, ord='fro')
print('Difference was: %f ', % (difference, ))
if difference < 0.001:
    print('Good! The distance matrices are the same')
else:
    print('Uh-oh! The distance matrices are different')
```

```
Difference was: 0.000000
Good! The distance matrices are the same
```

In [13]:
```
# Now implement the L1 distance metric inside: compute_l1_distances.
# Then run the following timing of one loop and the code
dists_l1 = classifier.compute_l1_distances(X_test)
print(dists_l1.shape)
```

```
(500, 5000)
```

In [14]:
```
# Now run the function predict_labels:
# We use k = 1 (which is Nearest Neighbor).
y_test_pred = classifier.predict_labels(dists_l1, k=1)

# Compute and print the fraction of correctly predicted examples.
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f', % (num_correct, num_test, accuracy))
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

In [15]:
```
# Now let's use k = 5. You should expect a slightly higher accuracy.
y_test_pred = classifier.predict_labels(dists_l1, k=5)
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f', % (num_correct, num_test, accuracy))
```

```
Got 141 / 500 correct => accuracy: 0.282000
```

In [16]:
```
# Now let's compare how fast the implementations are
def time_function(f, *args):
    """
    Call a function f with args and return the time (in seconds) that it took to execute.
    """
    import time
    tic = time.time()
    f(*args)
    toc = time.time()
    return toc - tic

two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)

one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)

no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)

# uncomment this when you have implemented the l1 metric
# l1_loop_time = time_function(classifier.compute_l1_distances, X_test)
# print('L1 loop version took %f seconds' % l1_loop_time)

# you should see significantly faster performance with the fully vectorized implementation
```

```
Two loop version took 32.451205 seconds
One loop version took 75.134616 seconds
No loop version took 0.773219 seconds
L1 loop version took 94.164875 seconds
```

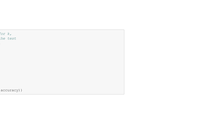
Cross-validation

We have implemented the k-Nearest Neighbor classifier but we set the value k = 5 arbitrarily. We will now determine the best value of this hyperparameter with cross-validation.

In [17]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
################################################################################
# TODO: Split the training data into 5 folds. After splitting, X_train_folds and    #
# y_train_folds should each be lists of length num_folds, where                      #
# y_train_folds[i] is the label vector for the points in X_train_folds[i]. Hint:     #
# Find out the numpy array split function.                                           #
################################################################################
pass
# Write your code here
X_train = np.array(X_train)
y_train = np.array(y_train)
for i in range(num_folds):
    # find the indices of the validation data into one train and test data
    X_train_i = np.vstack(X_train[:int(i*len(X_train)/5)], X_train[int(i+1)*
    y_train_i = np.hstack(y_train[:int(i*len(X_train)/5)], y_train[int(i+1)*
    X_train_folds.append(X_train_i)
    y_train_folds.append(y_train_i)

    # print if(X_train_i.shape, y_train_i.shape)

    # apply KNN
    classifier = KNearestNeighbor()
    classifier.train(X_train_i, y_train_i)
    dists = classifier.compute_distances_no_loops(X_test_i)
    y_test_pred = classifier.predict_labels(dists, k=5)
    num_correct = np.sum(y_test_pred == y_test_i)
    accuracy[i] = float(num_correct) / num_test

k_to_accuracies[k] = accuracy
################################################################################
# TODO:                                                                        #
# Now print the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

```
k = 1, accuracy = 0.526000
k = 1, accuracy = 0.514000
k = 1, accuracy = 0.528000
k = 1, accuracy = 0.556000
k = 1, accuracy = 0.489000
k = 3, accuracy = 0.478000
k = 3, accuracy = 0.498000
k = 3, accuracy = 0.505000
k = 3, accuracy = 0.535000
k = 3, accuracy = 0.496000
k = 5, accuracy = 0.506000
k = 5, accuracy = 0.498000
k = 5, accuracy = 0.542000
k = 5, accuracy = 0.560000
k = 5, accuracy = 0.548000
k = 8, accuracy = 0.524000
k = 8, accuracy = 0.542000
k = 8, accuracy = 0.560000
k = 8, accuracy = 0.568000
k = 10, accuracy = 0.532000
k = 10, accuracy = 0.542000
k = 10, accuracy = 0.560000
k = 10, accuracy = 0.568000
k = 12, accuracy = 0.538000
k = 12, accuracy = 0.542000
k = 12, accuracy = 0.560000
k = 12, accuracy = 0.568000
k = 15, accuracy = 0.538000
k = 15, accuracy = 0.542000
k = 15, accuracy = 0.560000
k = 15, accuracy = 0.568000
k = 20, accuracy = 0.538000
k = 20, accuracy = 0.542000
k = 20, accuracy = 0.560000
k = 20, accuracy = 0.568000
k = 50, accuracy = 0.570000
k = 50, accuracy = 0.582000
k = 50, accuracy = 0.576000
k = 50, accuracy = 0.594000
k = 50, accuracy = 0.596000
k = 100, accuracy = 0.572000
k = 100, accuracy = 0.582000
k = 100, accuracy = 0.576000
k = 100, accuracy = 0.594000
k = 100, accuracy = 0.596000
```

In [18]:
```
# plot the raw observations
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.items())]
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.items())]
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```



In [19]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should see at least a 139 accuracy on the test data.
best_k = 10

classifier = KNearestNeighbor()
classifier.train(X_train, y_train)
y_test_pred = classifier.predict(X_test, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f', % (num_correct, num_test, accuracy))
```

```
Got 141 / 500 correct => accuracy: 0.282000
```

Inline Question #3:

- What was the best value of k?
- What are the advantages and disadvantages of using a high value of k?

Your Answer:

- The best value of k is 10
- The advantage is high k can average out extreme point because of the voting. The disadvantage is that higher k will introduce more wrong comparisons thus can fall into a wrong region. Depending on the quality of training data, the k should balance the extreme point and the number of close points

In [ ]: