k-Nearest Neighbor (kNN) exercise

Complete and hand in this completed worksheet (including its outputs and any supporting code outside of the worksheet) with your assignment submission. For more details see the [assignments page]() on the course website.

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 transfering 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 [2]:

```python
# Run some setup code for this notebook.
from __future__ import print_function
import random
import numpy as np
from cs231n.data_utils import load_CIFAR10
import matplotlib.pyplot as plt


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

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

In [3]:

```python
# Load the raw CIFAR-10 data.
cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause memory issue)
try:
   del X_train, y_train
   del X_test, y_test
   print('Clear previously loaded data.')
except:
   pass
# load training and testing variables with new data
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', 'ship', 'truck']
num_classes = len(classes)
```

```
samples_per_class = 5
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 [6]:

```
# 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 [7]:

```
# Reshape the image data into rows (each image is one row, as the whole image pixels reduced to on
e row)
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 [19]:

```
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)
print ("done with KNN training on your data Mkhanyisi :) , O(1) operation")
```

```
done with KNN training on your data Mkhanyisi :) , O(1) operation
```

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, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** 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 computes the distance matrix one element at a time.

In [20]:

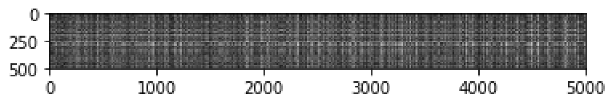
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.

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

```
(500, 5000)
```

In [21]:

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

- Distinctly bright rows are a result of the training images having a large distance between them and the test image in that row
- Distinctly bright columns are due to the training images having a large difference with the testing images

In [22]:

```
# 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))
# I got accuracy: 0.196000 first attempt, accuracy: 0.196000 second attempt
```

```
Got 98 / 500 correct => accuracy: 0.196000
```

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

In [23]:

```
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))
# better perfomance than with k=1
```

```
Got 106 / 500 correct => accuracy: 0.212000
```

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

Inline Question 2 We can also other distance metrics such as L1 distance. The performance of a Nearest Neighbor classifier that uses L1 distance will not change if (Select all that apply.):

1. The data is preprocessed by subtracting the mean.
2. The data is preprocessed by subtracting the mean and dividing by the standard deviation.
3. The coordinate axes for the data are rotated.
4. None of the above.

Your Answer: -> none of the above *Your explanation*: -> this is because the choice between the distance metrics affects the decision boundary. The mean, standard deviation and data rotation are different for these two distance metrics, as the nagitive values lead to different values than if L2 distance is used. The sensitivities of these two models overall are different, as L2 measures distance from origin/point, irrespective of axis and thus these parameters cannot be the same for these difference distance metrics. Rotation of the axes still leads to different statistical values

In [24]:

```
# Now lets 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 [25]:

```
# 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 [27]:

```
# 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
print ("starting timing\n")
two_loop_time = time_function(classifier.compute_distances_two_loops, X_test)
print('Two loop version took %f seconds' % two_loop_time)
print ("two loop done\n")
```

```
one_loop_time = time_function(classifier.compute_distances_one_loop, X_test)
print('One loop version took %f seconds' % one_loop_time)
print ("one loop done \n")
no_loop_time = time_function(classifier.compute_distances_no_loops, X_test)
print('No loop version took %f seconds' % no_loop_time)
print ("no-loop done")

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

```
Two loop version took 33.749934 seconds
One loop version took 34.856180 seconds
No loop version took 0.499656 seconds
```

In [28]:

```
print ("starting\n")
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

X_train_folds = []
y_train_folds = []
# Split up the training data into 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: Look up the numpy array_split function. -> (splits array into sub arrays)

X_train_folds = np.array_split(X_train,num_folds)
y_train_folds = np.array_split(y_train,num_folds)

# A dictionary holding the accuracies for different values of k that we find
# when running cross-validation. After running cross-validation,
# k_to_accuracies[k] should be a list of length num_folds giving the different
# accuracy values that we found when using that value of k.
k_to_accuracies = {}


# Store the accuracies for all fold and all
# values of k in the k_to_accuracies dictionary.

print ("starting cross validation\n")

# Perform k-fold cross validation to find the best value of k. For each possible value of k
for k_val in k_choices:
    k_to_accuracies[k_val] = np.zeros(num_folds)   # accuracies array for each fold
    for i in range(num_folds): # run the k-nearest-neighbor algorithm num_folds times
        # use all but(excpect last fold) one of the folds as training data
        x_tr = np.array(X_train_folds[:i] + X_train_folds[i + 1:])
        y_tr = np.array(y_train_folds[:i] + y_train_folds[i + 1:])
        # reshape train to one row array
        x_tr = x_tr.reshape(X_train_folds[i].shape[0] * 4, -1)
        y_tr = y_tr.reshape(y_train_folds[i].shape[0] * 4, -1)

        # last fold as a test set
        x_te = np.array(X_train_folds[i])
        y_te = np.array(y_train_folds[i])

        # train
        classifier.train(x_tr, y_tr)
        # use fastet algorithm for computing distances
        dists_ = classifier.compute_distances_no_loops(x_te)
        # predict labels
        y_pred = classifier.predict_labels(dists_, k_val)

        # find the accuracy for that k
        num_correct = np.sum(y_pred == y_te)
        accuracy = float(num_correct) / num_test
        k_to_accuracies[k_val][i] = accuracy # record accuracy
print ("done with cross validation\n\n")
# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))
```

```
starting
```

```
starting cross validation

done with cross validation


k = 1, accuracy = 0.422000
k = 1, accuracy = 0.358000
k = 1, accuracy = 0.386000
k = 1, accuracy = 0.380000
k = 1, accuracy = 0.382000
k = 3, accuracy = 0.412000
k = 3, accuracy = 0.408000
k = 3, accuracy = 0.448000
k = 3, accuracy = 0.418000
k = 3, accuracy = 0.386000
k = 5, accuracy = 0.412000
k = 5, accuracy = 0.414000
k = 5, accuracy = 0.374000
k = 5, accuracy = 0.444000
k = 5, accuracy = 0.398000
k = 8, accuracy = 0.412000
k = 8, accuracy = 0.404000
k = 8, accuracy = 0.388000
k = 8, accuracy = 0.450000
k = 8, accuracy = 0.398000
k = 10, accuracy = 0.384000
k = 10, accuracy = 0.404000
k = 10, accuracy = 0.404000
k = 10, accuracy = 0.426000
k = 10, accuracy = 0.388000
k = 12, accuracy = 0.428000
k = 12, accuracy = 0.408000
k = 12, accuracy = 0.404000
k = 12, accuracy = 0.420000
k = 12, accuracy = 0.384000
k = 15, accuracy = 0.418000
k = 15, accuracy = 0.402000
k = 15, accuracy = 0.404000
k = 15, accuracy = 0.222000
k = 15, accuracy = 0.404000
k = 20, accuracy = 0.418000
k = 20, accuracy = 0.408000
k = 20, accuracy = 0.194000
k = 20, accuracy = 0.222000
k = 20, accuracy = 0.364000
k = 50, accuracy = 0.198000
k = 50, accuracy = 0.208000
k = 50, accuracy = 0.210000
k = 50, accuracy = 0.222000
k = 50, accuracy = 0.194000
k = 100, accuracy = 0.214000
k = 100, accuracy = 0.208000
k = 100, accuracy = 0.210000
k = 100, accuracy = 0.222000
k = 100, accuracy = 0.194000
```

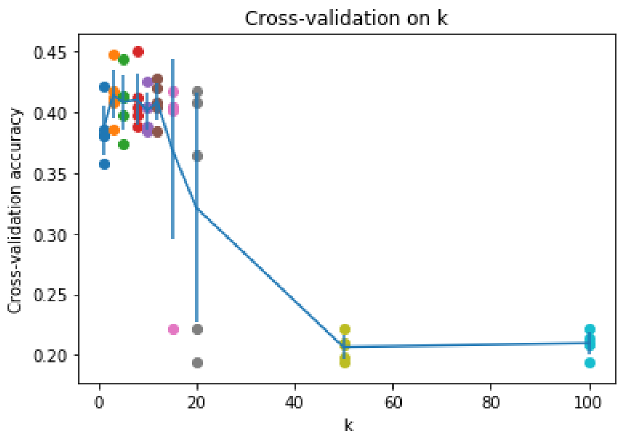
Cross-validation

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 [29]:

```
# 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 [30]:

```
# 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 be able to get above 28% accuracy on the test data.
best_k = 3

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 106 / 500 correct => accuracy: 0.212000
```

Inline Question 3 Which of the following statements about $k$-Nearest Neighbor ($k$-NN) are true in a classification setting, and for all $k$? Select all that apply.

1. The training error of a 1-NN will always be better than that of 5-NN.
2. The test error of a 1-NN will always be better than that of a 5-NN.
3. The decision boundary of the k-NN classifier is linear.
4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set.
5. None of the above.

Your Answer: 4 is true, *Your explanation*: The test/classification compares the test image to all training images, and thus the larger the number images, the longer the execution, as this is an order O(N) process.