knn

May 14, 2017

1 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

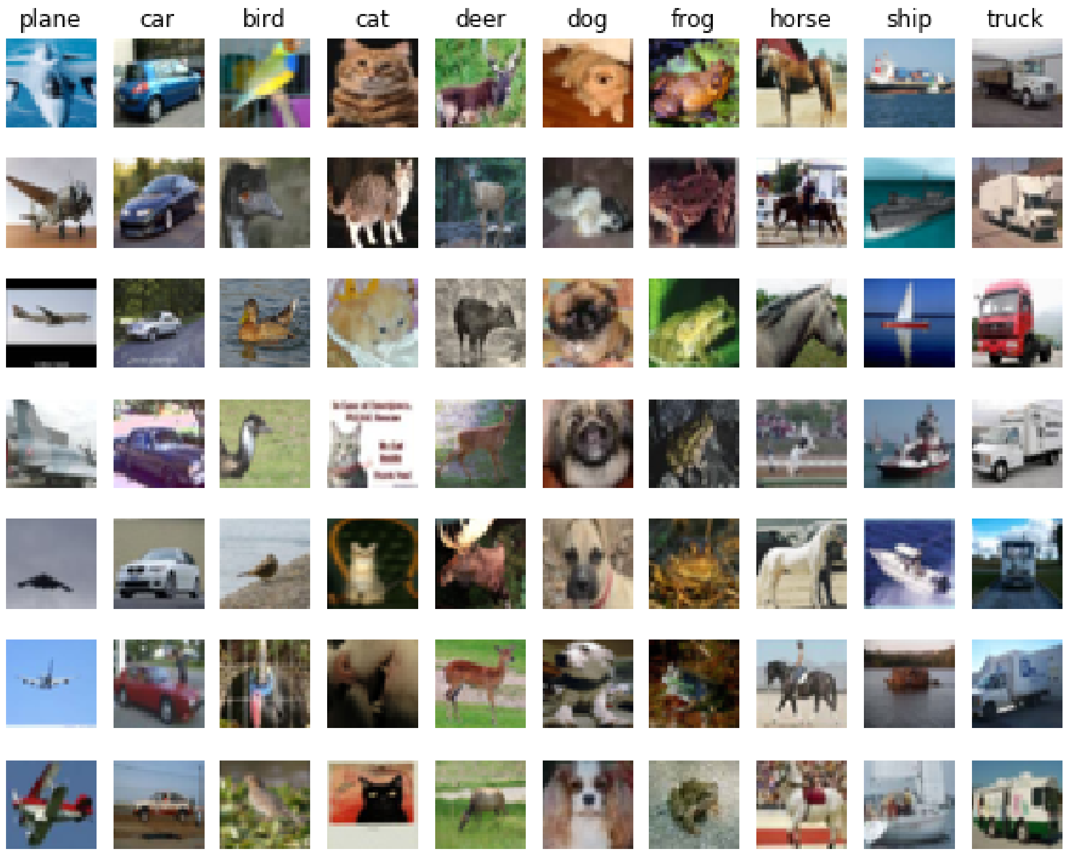
In this exercise you will implement these steps and understand the basic Image Classification pipeline, cross-validation, and gain proficiency in writing efficient, vectorized code.

```
In [1]: # Run some setup code for this notebook.

        import random
        import numpy as np
        from cs231n.data_utils import load_CIFAR10
        import matplotlib.pyplot as plt

        from __future__ import print_function

        # This is a bit of magic to make matplotlib figures appear inline in the no
        # 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-i
        %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', '
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 [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

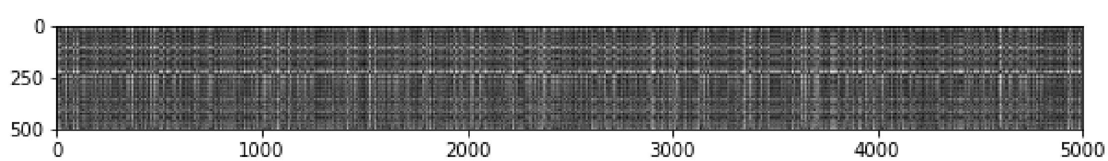
Lets begin with computing the distance matrix between all training and test examples. For example, if there are **Ntr** training examples and **Nte** test examples, this stage should result in a **Nte x Ntr** matrix where each element (i,j) is the distance between the i-th test and j-th train example.

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 [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 a
        # 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?

Your Answer: *fill this in.*

```
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, accur
```

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

```
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, accu
```

```
Got 145 / 500 correct => accuracy: 0.290000
```

You should expect to see approximately `27%` accuracy. Now lets try out a larger `k`, say `k = 5`: You should expect to see a slightly better performance than with `k = 1`.

```
In [11]: # Now lets speed up distance matrix computation by using partial vectoriza
         # with one loop. Implement the function compute_distances_one_loop and rur
         # code below:
         dists_one = classifier.compute_distances_one_loop(X_test)
         # To ensure that our vectorized implementation is correct, we make sure th
         # agrees with the naive implementation. There are many ways to decide whet
         # two matrices are similar; one of the simplest is the Frobenius norm. In
         # you haven't seen it before, the Frobenius norm of two matrices is the sc
         # root of the squared sum of differences of all elements; in other words,
         # the matrices into vectors and compute the Euclidean distance between the
         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_l
         # 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]: # 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 t
             """
             import time
             tic = time.time()
             f(*args)
             toc = time.time()
             return toc - tic

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

         one_loop_time = time_function(classifier.compute_distances_one_loop, X_tes
         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)

         # you should see significantly faster performance with the fully vectorize
```

```
Two loop version took 32.649811 seconds
One loop version took 58.582409 seconds
No loop version took 0.373447 seconds
```

1.0.1 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 [14]: num_folds = 5
         k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]
```

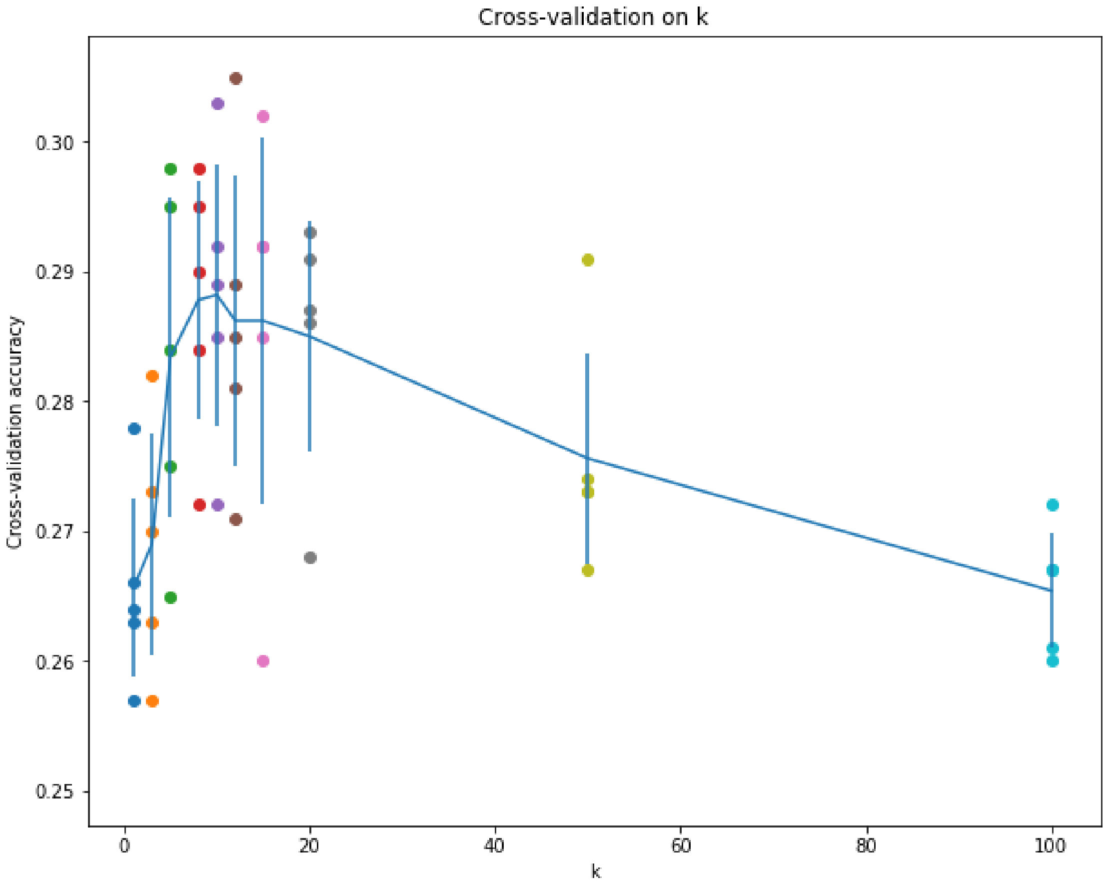
```
X_train_folds = []
y_train_folds = []
################################################################################
# TODO:
# Split up the training data into folds. After splitting, X_train_folds an
# y_train_folds should each be lists of length num_folds, where
# y_train_folds[i] is the label vector for the points in X_train_folds[i].
# Hint: Look up the numpy array_split function.
################################################################################
n = X_train.shape[0]
m = (int) (n / num_folds)
for i in range(num_folds):
    X_train_folds.append(X_train[i*m:(i+1)*m])
    y_train_folds.append(y_train[i*m:(i+1)*m])
################################################################################
#                                 END OF YOUR CODE
################################################################################

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


################################################################################
# TODO:
# Perform k-fold cross validation to find the best value of k. For each
# possible value of k, run the k-nearest-neighbor algorithm num_folds time
# where in each case you use all but one of the folds as training data and
# last fold as a validation set. Store the accuracies for all fold and al
# values of k in the k_to_accuracies dictionary.
################################################################################
for k in k_choices:
    k_to_accuracies[k] = []
    for i in range(num_folds):
        x = np.concatenate([X_train_folds[j] for j in range(num_folds) if
        y = np.concatenate([y_train_folds[j] for j in range(num_folds) if
        classifier.train(x, y)
        dists = classifier.compute_distances_no_loops(X_train_folds[i])
        y_test_pred = classifier.predict_labels(dists, k=k)
        num_correct = np.sum(y_test_pred ==  y_train_folds[i])
        accuracy = float(num_correct) /  len(y_train_folds[i])
        k_to_accuracies[k].append(accuracy)
################################################################################
#                                 END OF YOUR CODE
################################################################################
```

```python
# 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))
```

```
k = 1, accuracy = 0.263000
k = 1, accuracy = 0.257000
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.257000
k = 3, accuracy = 0.263000
k = 3, accuracy = 0.273000
k = 3, accuracy = 0.282000
k = 3, accuracy = 0.270000
k = 5, accuracy = 0.265000
k = 5, accuracy = 0.275000
k = 5, accuracy = 0.295000
k = 5, accuracy = 0.298000
k = 5, accuracy = 0.284000
k = 8, accuracy = 0.272000
k = 8, accuracy = 0.295000
k = 8, accuracy = 0.284000
k = 8, accuracy = 0.298000
k = 8, accuracy = 0.290000
k = 10, accuracy = 0.272000
k = 10, accuracy = 0.303000
k = 10, accuracy = 0.289000
k = 10, accuracy = 0.292000
k = 10, accuracy = 0.285000
k = 12, accuracy = 0.271000
k = 12, accuracy = 0.305000
k = 12, accuracy = 0.285000
k = 12, accuracy = 0.289000
k = 12, accuracy = 0.281000
k = 15, accuracy = 0.260000
k = 15, accuracy = 0.302000
k = 15, accuracy = 0.292000
k = 15, accuracy = 0.292000
k = 15, accuracy = 0.285000
k = 20, accuracy = 0.268000
k = 20, accuracy = 0.293000
k = 20, accuracy = 0.291000
k = 20, accuracy = 0.287000
k = 20, accuracy = 0.286000
k = 50, accuracy = 0.273000
k = 50, accuracy = 0.291000
```

```
k = 50, accuracy = 0.274000
k = 50, accuracy = 0.267000
k = 50, accuracy = 0.273000
k = 100, accuracy = 0.261000
k = 100, accuracy = 0.272000
k = 100, accuracy = 0.267000
k = 100, accuracy = 0.260000
k = 100, accuracy = 0.267000
```

```
In [15]: # 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 deviatio
         accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.i
         accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.ite
         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 [16]:
```
# 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 t
# data. You should be able to get above 28% accuracy on the test data.
best_k = 1

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, accu
```

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

In [ ]: