01_PW_Sol_knn_mnist

November 7, 2018

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 [8]:
```
# Run some setup code for this notebook.

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import os

# 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, 10.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

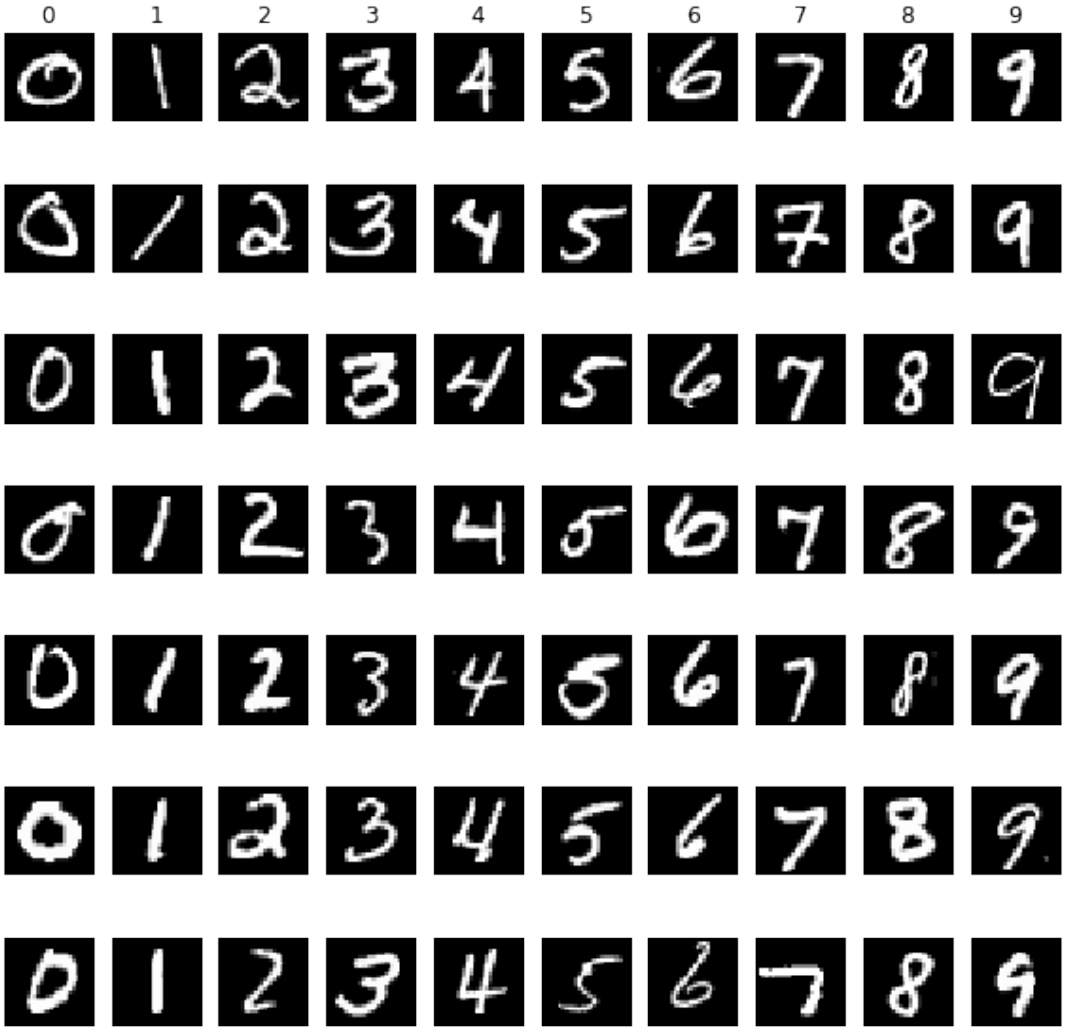
In [9]:
```
def load_MNIST(ROOT):
  '''load all of mnist
  training set first'''
  Xtr = []
  train = pd.read_csv(os.path.join(ROOT, 'mnist_train.csv'))
  X = np.array(train.drop('label', axis=1))
  Ytr = np.array(train['label'])
  # With this for-loop we give the data a shape of the acctual image (28x28)
  # instead of the shape in file (1x784)
```

```
        for row in X:
            Xtr.append(row.reshape(28,28))
        # load test set second
        Xte = []
        test = pd.read_csv(os.path.join(ROOT, 'mnist_test.csv'))
        X = np.array(test.drop('label', axis=1))
        Yte = np.array(test['label'])
        # same reshaping
        for row in X:
            Xte.append(row.reshape(28,28))

        return np.array(Xtr), np.array(Ytr), np.array(Xte), np.array(Yte)
```

In [10]:
```
# Load the raw MNIST data.
mnist_dir = './PW02/ex3-knn-mnist/mnist'
X_train, y_train, X_test, y_test = load_MNIST(mnist_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:  (60000, 28, 28)
Training labels shape:  (60000,)
Test data shape:  (10000, 28, 28)
Test labels shape:  (10000,)
```

In [11]:
```
classes = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes): # y and cls takes values from 0-9
    idxs = np.flatnonzero(y_train == y) # gets the indices of samples that correspond
    idxs = np.random.choice(idxs, samples_per_class, replace=False) # picks randomly
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1   # determines the sub-plot index
        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 [12]:
```
# Subsample the data for more efficient code execution in this exercise
# Just to make it go faster of course you can run on it on the total data set
num_training = 5000
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

# 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:  (5000, 28, 28)
Training labels shape:  (5000,)
Test data shape:  (500, 28, 28)
Test labels shape:  (500,)
```

```
In [13]: # Shape the images data back into rows (we could also take the original data)
         X_train = np.reshape(X_train, (X_train.shape[0], -1)) # when reshaping, -1 means "inf
         X_test = np.reshape(X_test, (X_test.shape[0], -1))    # in this case it flattens the
         print(X_train.shape, X_test.shape)
```

```
(5000, 784) (500, 784)
```

```
In [14]: import numpy as np
         from collections import Counter  #added by JH

         class KNearestNeighbor(object):
           """ a kNN classifier with L2 distance """

           def __init__(self):
             pass

           def train(self, X, y):
             """
             Train the classifier. For k-nearest neighbors this is just
             memorizing the training data.

             Inputs:
             - X: A numpy array of shape (num_train, D) containing the training data
               consisting of num_train samples each of dimension D.
             - y: A numpy array of shape (N,) containing the training labels, where
                  y[i] is the label for X[i].
             """
             self.X_train = X
             self.y_train = y

           def predict(self, X, k=1, num_loops=0):
             """
             Predict labels for test data using this classifier.

             Inputs:
             - X: A numpy array of shape (num_test, D) containing test data consisting
                  of num_test samples each of dimension D.
             - k: The number of nearest neighbors that vote for the predicted labels.
```

```
    - num_loops: Determines which implementation to use to compute distances
      between training points and testing points.

    Returns:
    - y: A numpy array of shape (num_test,) containing predicted labels for the
      test data, where y[i] is the predicted label for the test point X[i].
    """
    if num_loops == 0:
      dists = self.compute_distances_no_loops(X)
    elif num_loops == 1:
      dists = self.compute_distances_one_loop(X)
    elif num_loops == 2:
      dists = self.compute_distances_two_loops(X)
    else:
      raise ValueError('Invalid value %d for num_loops' % num_loops)

    return self.predict_labels(dists, k=k)

  def compute_distances_two_loops(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using a nested loop over both the training data and the
    test data.

    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 training
      point.
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
      for j in range(num_train):
        #####################################################################
        # TODO:                                                             #
        # Compute the l2 distance between the ith test point and the jth    #
        # training point, and store the result in dists[i, j]. You should   #
        # not use a loop over dimension.                                    #
        #####################################################################
        #dists[i][j] = np.linalg.norm(X[i]-X_train[j])

        dists[i][j] = np.sqrt(np.sum(np.square(X[i]-self.X_train[j])))

        #####################################################################
```

```
      #                         END OF YOUR CODE                            #
      #########################################################################
    return dists

  def compute_distances_one_loop(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using a single loop over the test data.

    Input / Output: Same as compute_distances_two_loops
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
      #######################################################################
      # TODO:                                                               #
      # Compute the l2 distance between the ith test point and all training #
      # points, and store the result in dists[i, :].                        #
      #######################################################################
      # X[i]-X_train to broadcast the subtraction, leads to a shape 5000,3072
      # np.linalg.norm with axis=1 will compute the norm for all lines (along dim 307
      # dists[i, :] = np.linalg.norm(X[i]-X_train, axis=1)

      dists[i,:] = np.sqrt(np.sum(np.square(X[i]-self.X_train), axis=1))

      #######################################################################
      #                         END OF YOUR CODE                            #
      #######################################################################
    return dists

  def compute_distances_no_loops(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using no explicit loops.

    Input / Output: Same as compute_distances_two_loops
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    #########################################################################
    # TODO:                                                                 #
    # Compute the l2 distance between all test points and all training      #
    # points without using any explicit loops, and store the result in      #
    # dists.                                                                #
    #                                                                       #
    # You should implement this function using only basic array operations; #
```

```
    # in particular you should not use functions from scipy.                #
    #                                                                       #
    # HINT: Try to formulate the l2 distance using matrix multiplication    #
    #       and two broadcast sums.                                         #
    #########################################################################
    # split (p-q)^2 to p^2 + q^2 - 2pq

    dists = np.sqrt((X**2).sum(axis=1, keepdims=True) + (self.X_train**2).sum(axis=1)

    #########################################################################
    #                         END OF YOUR CODE                              #
    #########################################################################
    return dists

  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 the
      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 range(num_test):
      # A list of length k storing the labels of the k nearest neighbors to
      # the ith test point.
      closest_y = []
      #########################################################################
      # TODO:                                                                 #
      # Use the distance matrix to find the k nearest neighbors of the ith    #
      # testing point, and use self.y_train to find the labels of these       #
      # neighbors. Store these labels in closest_y.                           #
      # Hint: Look up the function numpy.argsort.                             #
      #########################################################################

      closest_idx = np.argsort(dists[i])    # compute the indices that would sort the
      closest_y = self.y_train[closest_idx] # compute the sorted array of labels acco
      closest_y = closest_y[:k]             # retain the k nearest

      #########################################################################
      # TODO:                                                                 #
      # Now that you have found the labels of the k nearest neighbors, you    #
```

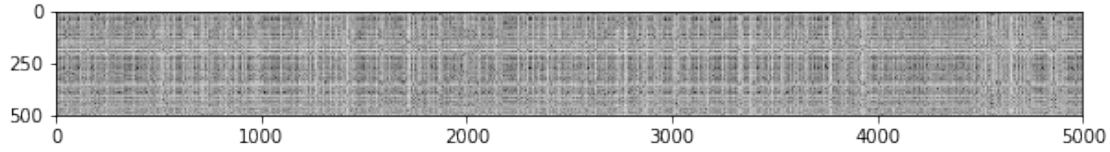
```
            # need to find the most common label in the list closest_y of labels.   #
            # Store this label in y_pred[i]. Break ties by choosing the smaller     #
            # label.                                                                #
            #########################################################################

            winner,num_vote = Counter(closest_y).most_common(1)[0]
            y_pred[i] = winner

            #########################################################################
            #                           END OF YOUR CODE                            #
            #########################################################################

        return y_pred
```

In [15]:
```
# 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)
```

In [16]:
```
# 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 [17]:
```
# 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?

Your Answer:
The distinct bright rows are caused by a number in the testset being far from all numbers in the training set. The same goes for the bright columns.This means that the number represented by this row/column has no good match in the test/training sets. A reason might be that the person that has writen this number has a very ugly handwriting.

```
In [18]: # 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' % (int(num_correct), num_test, accuracy))
```

```
Got 453 / 500 correct => accuracy: 0.906000
```

You should expect to see approximately 90% accuracy. Now lets try out a larger `k`, say `k = 5`:

```
In [19]: 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' % (int(num_correct), num_test, accuracy))
```

```
Got 456 / 500 correct => accuracy: 0.912000
```

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

```
In [20]: # 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 [21]: # 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 [22]: # 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 execut
           """
           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)

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

```
Two loop version took 28.840394 seconds
One loop version took 10.589184 seconds
No loop version took 1.235887 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.

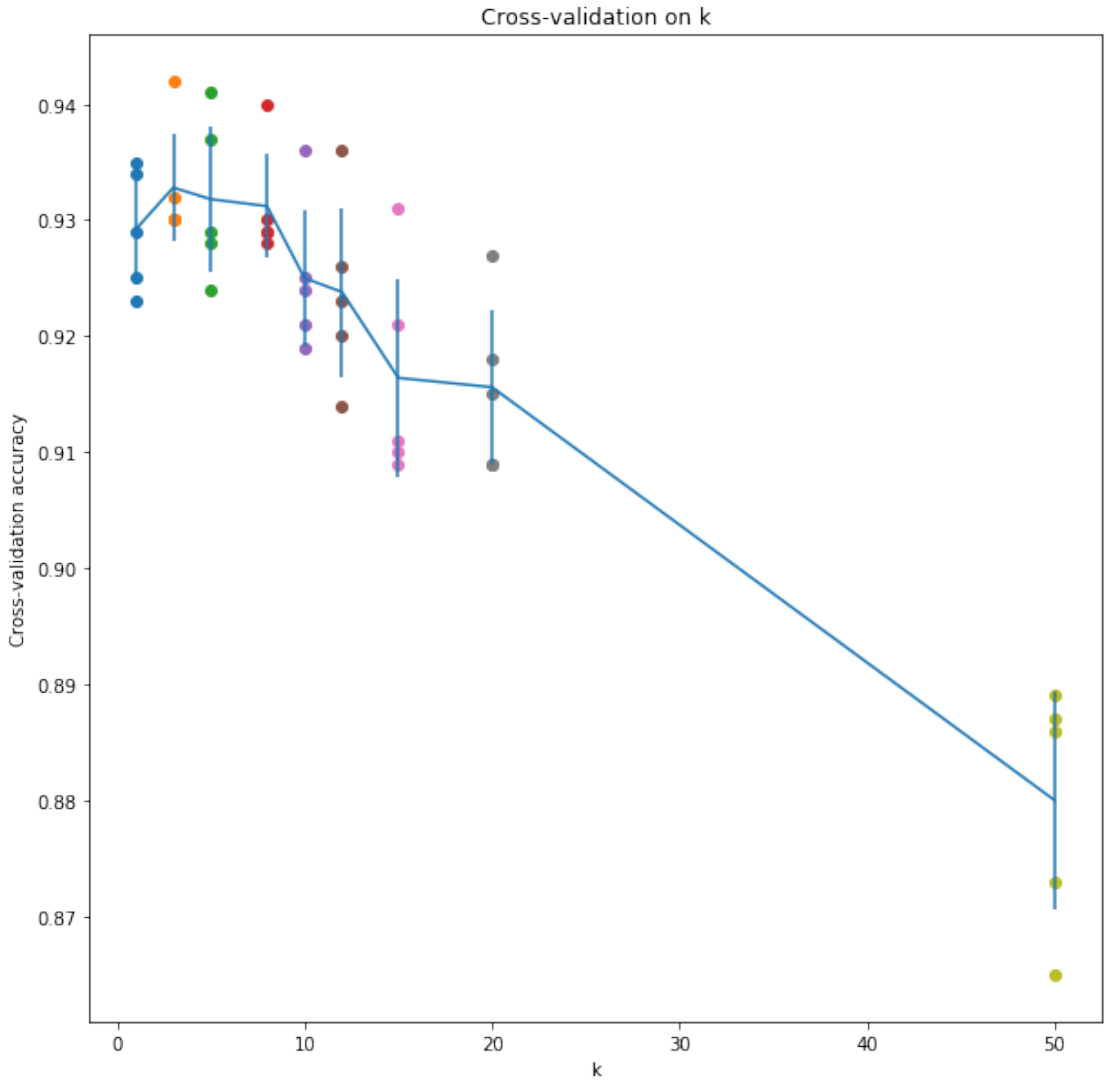
In [23]:
```
num_folds = 5
k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50]

#X_train_folds = []
#y_train_folds = []
################################################################################
# TODO:                                                                        #
# 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.                                #
################################################################################

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

################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

# 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 = {}


################################################################################
# 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 times,   #
# where in each case you use all but one of the folds as training data and the #
# last fold as a validation set. Store the accuracies for all fold and all     #
# values of k in the k_to_accuracies dictionary.                               #
################################################################################
acc_k = np.zeros((len(k_choices), num_folds), dtype=np.float)

for ik ,k in enumerate(k_choices):
    for i in range(num_folds):
        train_set = np.concatenate((X_train_folds[:i]+X_train_folds[i+1:]))
        label_set = np.concatenate((y_train_folds[:i]+y_train_folds[i+1:]))
        classifier.train(train_set, label_set)
        y_pred_fold = classifier.predict(X_train_folds[i], k=k, num_loops=0)
        num_correct = np.sum(y_pred_fold == y_train_folds[i])
```

```
                acc_k[ik, i] = float(num_correct) / y_pred_fold.shape[0]
            k_to_accuracies[k] = acc_k[ik]

        ################################################################################
        #                                 END OF YOUR CODE                             #
        ################################################################################

        # 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 = 0.923000
k = 1, accuracy = 0.925000
k = 1, accuracy = 0.935000
k = 1, accuracy = 0.929000
k = 1, accuracy = 0.934000
k = 3, accuracy = 0.930000
k = 3, accuracy = 0.930000
k = 3, accuracy = 0.942000
k = 3, accuracy = 0.932000
k = 3, accuracy = 0.930000
k = 5, accuracy = 0.924000
k = 5, accuracy = 0.937000
k = 5, accuracy = 0.941000
k = 5, accuracy = 0.929000
k = 5, accuracy = 0.928000
k = 8, accuracy = 0.930000
k = 8, accuracy = 0.929000
k = 8, accuracy = 0.940000
k = 8, accuracy = 0.929000
k = 8, accuracy = 0.928000
k = 10, accuracy = 0.919000
k = 10, accuracy = 0.924000
k = 10, accuracy = 0.936000
k = 10, accuracy = 0.925000
k = 10, accuracy = 0.921000
k = 12, accuracy = 0.914000
k = 12, accuracy = 0.926000
k = 12, accuracy = 0.936000
k = 12, accuracy = 0.923000
k = 12, accuracy = 0.920000
k = 15, accuracy = 0.909000
k = 15, accuracy = 0.921000
k = 15, accuracy = 0.931000
k = 15, accuracy = 0.911000
k = 15, accuracy = 0.910000
k = 20, accuracy = 0.909000
```

```
k = 20, accuracy = 0.918000
k = 20, accuracy = 0.927000
k = 20, accuracy = 0.909000
k = 20, accuracy = 0.915000
k = 50, accuracy = 0.865000
k = 50, accuracy = 0.889000
k = 50, accuracy = 0.887000
k = 50, accuracy = 0.873000
k = 50, accuracy = 0.886000
```

In [24]:
```
# 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 [20]:
```
# 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 90% accuracy on the test data.
best_k = 5

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' % (int(num_correct), num_test, accuracy))
```

```
Got 456 / 500 correct => accuracy: 0.912000
```