knn

November 9, 2018

1 KNN Workbook for CS145 Homework 3

PRINT YOUR NAME AND UID HERE!
NAME: [Lee, Hun]
UID: [604958834]

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

Please print out the workbook entirely when completed.

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.

1.1 Import the appropriate libraries

```
In [2]: import numpy as np # for doing most of our calculations
        import matplotlib.pyplot as plt# for plotting
        from cs145.data_utils import load_CIFAR10 # function to load the CIFAR-10 dataset.

        # 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-ipython
        %load_ext autoreload
        %autoreload 2
```

```
In [3]: # Set the path to the CIFAR-10 data
        cifar10_dir = './cs145/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 [4]:
```
# 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', 'truc
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]: # 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)
```

2 K-nearest neighbors

In the following cells, you will build a KNN classifier and choose hyperparameters via k-fold cross-validation.

In [6]:
```python
# Import the KNN class
from lib import KNN
```

In [7]:
```python
# 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)
```

2.0.1 Questions

(1) Describe what is going on in the function knn.train().

(2) What are the pros and cons of this training step of KNN?

2.0.2 Answers

1. the training function is simply storing the testing data. It is known as lazy training
2. lazy learning takes less time to train the data. this method also effectively uses a richer hypothesis sapce since it uses many local linear functions to form an implicit global approximation to the targe function. The cons(downside) of this method is that it takes more time to predict the outcome.

2.1 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]:
```python
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition of the 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, 'fro')))
```

```
Time to run code: 34.045605182647705
Frobenius norm of L2 distances: 7906696.077040902
```

2.1.1 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. Normally it may takes 20-40 seconds.

If you implemented this correctly, evaluating np.linalg.norm(dists_L2, 'fro') should return: ~7906696

2.1.2 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 loop
        # 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): {}'.
```

```
Time to run code: 0.2533841133117676
Difference in L2 distances between your KNN implementations (should be 0): 1.4651847440245846e
```

2.1.3 Speedup

Depending on your computer speed, you should see a 20-100x speed up from vectorization and no difference in L2 distances between two implementations.

On our computer, the vectorized form took 0.20 seconds while the naive implementation took 26.88 seconds.

2.2 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.
         # ================================================================ #
```

```
    YPred = knn.predict_labels(dists_L2_vectorized)
    diffNum = np.sum(YPred != y_test)
    error = diffNum / YPred.size

    # ================================================================ #
    # 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.

3 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.

3.0.1 Create training and validation folds

First, we will create the training and validation folds for use in k-fold cross validation.

```
In [41]: # 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]
         # ================================================================ #

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

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

         X_train_folds = np.asarray(X_train_folds)
         y_train_folds = np.asarray(y_train_folds)
```

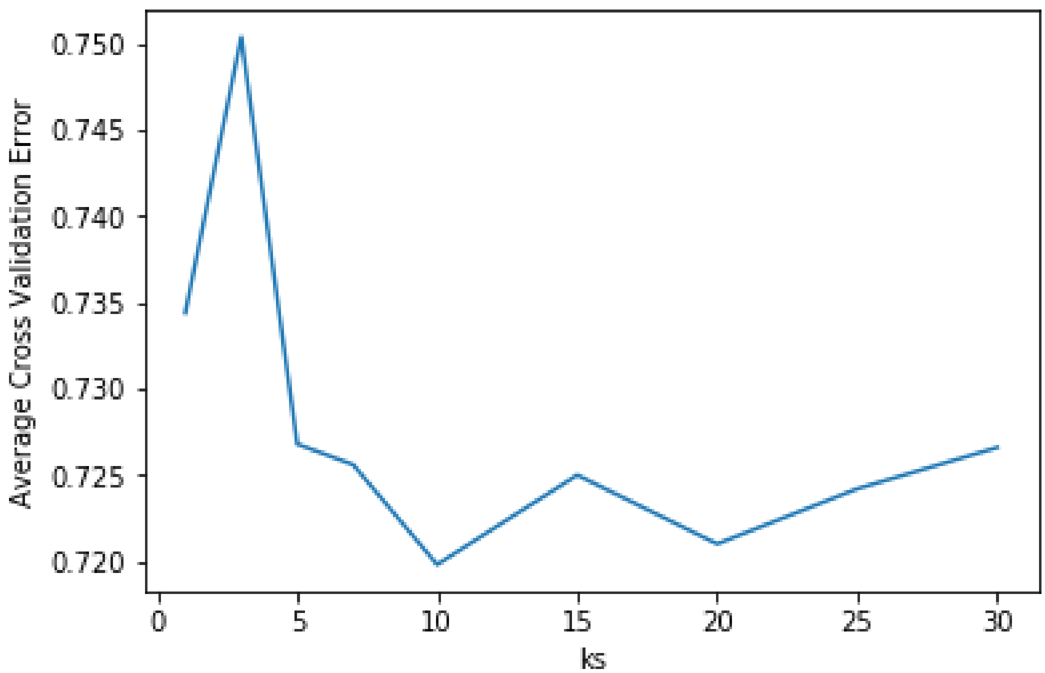
3.0.2 Optimizing the number of nearest neighbors hyperparameter.

In this section, we select different numbers of nearest neighbors and assess which one has the lowest k-fold cross validation error.

In [42]:
```
time_start =time.time()

ks = [1, 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.
# ================================================================ #

results = np.zeros(len(ks))
numX = X_train_folds[0].shape[0] * (num_folds -1)

numX1 = X_train_folds[0].shape[1]

numY = y_train_folds[0].shape[0] * (num_folds -1)

i = 0

for numK in (ks):
    for folds in range(num_folds):
        testX = X_train_folds[folds]
        trainX = np.delete(X_train_folds, folds, 0)
        trainX = trainX.reshape(numX, numX1)
        testY = y_train_folds[folds]
        TrainY = np.delete(y_train_folds, folds, 0).reshape(numY)

        knn.train(X=trainX, y=TrainY)
        L2dist = knn.compute_L2_distances_vectorized(X=testX)
        YPredC = knn.predict_labels(L2dist, numK)
        diffNumC = np.sum(YPredC != testY)
        errorC = diffNumC / YPredC.size
        results[i] = results[i] + errorC
    i += 1

results = results / num_folds


ks_min = ks[np.argsort(results)[0]]
results_min = min(results)
```

```
print('Set k = {0} and get minimum error as {1}'.format(ks_min,results_min))
# Plot of k vs. cross-validation error
plt.plot(ks,results)
plt.xlabel('ks')
plt.ylabel('Average Cross Validation Error')
plt.show()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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

```
Set k = 10 and get minimum error as 0.7198
```




```
Computation time: 41.87
```

3.0.3 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?

3.0.4 Answers:

1. I found out that best K = 10. Error% for other Ks were higher. This means that 10 closest neighbors were appropriate number to use in the test.
2. cross validation error for it is 0.7198. lowest error rate because k=10 is the more appropriate value than other ks.

4 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 [21]: 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=X_train, y=y_train)
         dists_L2 = knn.compute_L2_distances_vectorized(X=X_test)
         YPre = knn.predict_labels(dists_L2, ks_min)
         diffNum = np.sum(YPre != y_test)
         error = diffNum / YPre.size

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

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

```
Error rate achieved: 0.718
```

4.1 Question:

How much did your error improve by cross-validation over naively choosing $k = 1$ and using the L2-norm?

4.2 Answer:

1. the error rate for k =1 was 0.726. with k=10, the error rate decreased. The error rate for k=10 is 0.718. That is 0.008 improvement in the error rate. This error rate improvemnet could be achieved because we found the value for K that minize the error rate using the cross-validation.

4.3 The End of KNN Workbook

Please export this workbook as PDF file (see instructions) after completion.