KNN Workbook for CS145 Homework 3

Name: Kevin Chuang UID: 704769121

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.

Import the appropriate libraries

In [1]:

```
mport numpy as np # for doing most of our calculations
mport matplotlib.pyplot as plt# for plotting
rom cs145.data_utils import load_CIFAR10 # function to load the CIFAR-10 data
et.

 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-ipyt
on
load_ext autoreload
autoreload 2
```

In [2]:

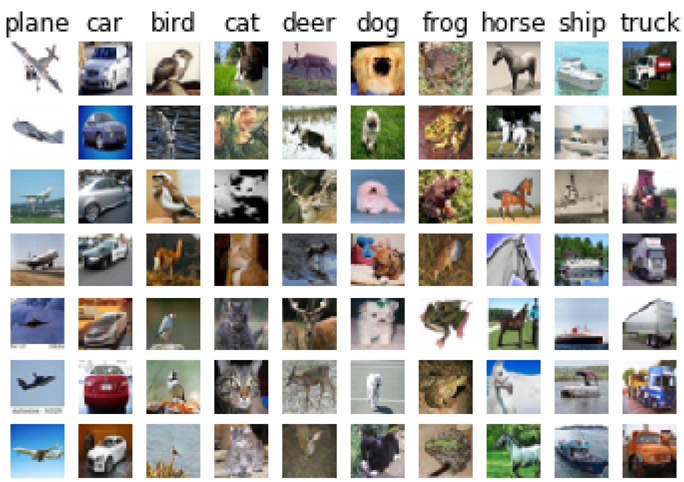
```
 Set the path to the CIFAR-10 data
ifar10_dir = './cs145/datasets/cifar-10-batches-py'
_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

 As a sanity check, we print out the size of the training and test data.
rint('Training data shape: ', X_train.shape)
rint('Training labels shape: ', y_train.shape)
rint('Test data shape: ', X_test.shape)
rint('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,)
```

I like CS 145.

$y = \sigma(\quad) + 1$

In [3]:
```
 Visualize some examples from the dataset.
 We show a few examples of training images from each class.
lasses = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'shi
', 'truck']
um_classes = len(classes)
amples_per_class = 7
or 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)
lt.show()
```



In [4]:
```
 Subsample the data for more efficient code execution in this exercise
um_training = 5000
ask = list(range(num_training))
_train = X_train[mask]
_train = y_train[mask]

um_test = 500
ask = list(range(num_test))
_test = X_test[mask]
_test = y_test[mask]

 Reshape the image data into rows
_train = np.reshape(X_train, (X_train.shape[0], -1))
_test = np.reshape(X_test, (X_test.shape[0], -1))
rint(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

K-nearest neighbors

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

In [5]:
```
 Import the KNN class
rom lib import KNN
```

In [6]:
```
 Declare an instance of the knn class.
nn = 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.
nn.train(X=X_train, y=y_train)
```

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?

Answers

1. knn.train() is just setting our training variables to X_train and y_train, which is lazy learning
2. Pros: Uses richer hypothesis space with multiple linear functions in comparison to eager learning which must commit to a single hypothesis. Cons: Takes less time training and more time predicting

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 [7]:
```
 Implement the function compute_distances() in the KNN class.
 Do not worry about the input 'norm' for now; use the default definition of t
e norm
   in the code, which is the 2-norm.
 You should only have to fill out the clearly marked sections.

mport time
ime_start =time.time()

ists_L2 = knn.compute_distances(X=X_test)

rint('Time to run code: {}'.format(time.time()-time_start))
rint('Frobenius norm of L2 distances: {}'.format(np.linalg.norm(dists_L2, 'fr
')))
```

```
Time to run code: 34.27286958694458
Frobenius norm of L2 distances: 7906696.077040902
```

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

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 [8]:
```
 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 loops.
 Note, this is SPECIFIC for the L2 norm.

ime_start =time.time()
ists_L2_vectorized = knn.compute_L2_distances_vectorized(X=X_test)
rint('Time to run code: {}'.format(time.time()-time_start))
rint('Difference in L2 distances between your KNN implementations (should be
0): {}'.format(np.linalg.norm(dists_L2 - dists_L2_vectorized, 'fro')))
```

```
ime to run code: 0.29451894760131836
ifference in L2 distances between your KNN implementations (should be 0): 0.
```

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.

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

rror = 1
_labels = knn.predict_labels(dists_L2,1)

 Compute and display the accuracy
um_correct = np.sum(y_labels == y_test)
rror = 1-float(num_correct) / num_test
 ================================================================ #
 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.
 ================================================================ #



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

rint(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.

Questions:

What could you do to improve the accuracy of the k-nearest neighbor classifier you just implemented? Write down your answer in less than 20 words.

Answers:

Increase the number of nearest neighbors.

The End of KNN Workbook

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