CS145 Howework 3, Part 1: kNN

Important Note: HW3 is due on **11:59 PM PT, Nov 9 (Monday, Week 6)**. Please submit through GradeScope.

Note that, Howework #3 has two jupyter notebooks to complete (Part 1: kNN and Part 2: Neural Network).

Print Out Your Name and UID

Name: Devyan Biswas, UID: 804988161

Before You Start

You need to first create HW2 conda environment by the given `cs145hw3.yml` file, which provides the name and necessary packages for this tasks. If you have `conda` properly installed, you may create, activate or deactivate by the following commands:

```
conda env create -f cs145hw3.yml
conda activate hw3
conda deactivate
```

OR

```
conda env create --name NAMEOFYOURCHOICE -f cs145hw3.yml
conda activate NAMEOFYOURCHOICE
conda deactivate
```

To view the list of your environments, use the following command:

```
conda env list
```

More useful information about managing environments can be found here.

You may also quickly review the usage of basic Python and Numpy package, if needed in coding for matrix operations.

In this notebook, you must not delete any code cells in this notebook. If you change any code outside the blocks (such as hyperparameters) that you are allowed to edit (between `START/END YOUR CODE HERE`), you need to highlight these changes. You may add some additional cells to help explain your results and observations.

Download and prepare the dataset

Download the CIFAR-10 dataset (file size: ~163M). Run the following from the HW3 directory:

```
cd hw3/data/datasets
./get_datasets.sh
```

Make sure you put the dataset downloaded under hw3/data/datasets folder. After downloading the dataset, you can start your notebook from the HW3 directory. Note that the dataset is used in both jupyter notebooks (kNN and Neural Networks). You only need to download the dataset once for HW3.

Import the appropriate libraries

```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from data.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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Now, to verify that the dataset has been successfully set up, the following code will print out the shape of train/test data and labels. The output shapes for train/test data are (50000, 32, 32, 3) and (10000, 32, 32, 3), while the labels are (50000,) and (10000,) respectively.

```
# Set the path to the CIFAR-10 data
cifar10_dir = './data/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,)
```

Now we visualize some examples from the dataset by showing 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 = 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()
```

```
# 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)
```

Implement K-nearest neighbors algorithms

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

```
# Import the KNN class
from hw3code import KNN
```

```
# 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)
```

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. The `knn.train()` function is just assigning the passed in X and y values into the model's X_train and y_train. We are using a lazy learning approach to KNN here, meaning that we don't do any real training; instead, we store the training values and then do our prediction/classification only on receiving a new test tuple.
2. The biggest cons to this method are that the time in prediction goes up significantly, as well as the amount of memory needed. However, the "training" time is much less. Additionally, unlike the eager approach, the lazy learning approach doesn't stick to one hypothesis. Instead, it is able to use many local linear functions that can actually emulate a global function for the dataset (what the notes term as a global approximation to the target function).

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.

```
# 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: 50.123364210128784
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.

```
# 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.

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): {}'.format(np.linalg.norm(dists_L2 - dists_L2_vectorized, 'fro')))
```

```
Time to run code: 0.388016939163208
Difference in L2 distances between your KNN implementations (should be 0):
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.

```
# 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

# ================================================================ #
# START 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.
# ================================================================ #
pred_labels = knn.predict_labels(dists_L2_vectorized, k = 1)
# print(pred_labels)
num_samples = pred_labels.shape[0]
count = 0
for i in range(num_samples):
    count += (pred_labels[i] != y_test[i])
error = count / num_samples
# ================================================================ #
# 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.

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 30 words.

Answers:

One way to resolve this is to do some processing on the dataset to make it easier to classify, like with feature/data scaling.

Optimizing KNN hyperparameters $k$

In this section, we'll take the KNN classifier that you have constructed and perform cross validation to choose a best value of $k$.

If you are not familiar with cross validation, cross-validation is a technique for evaluating ML models by training several ML models on subsets of the available input data and evaluating them on the complementary subset of the data. Use cross-validation to detect overfitting, ie, failing to generalize a pattern. More specifically, in k-fold cross-validation, you evenly split the input data into k subsets of data (also known as folds). You train an ML model on all but one (k-1) of the subsets, and then evaluate the model on the subset that was not used for training. This process is repeated k times, with a different subset reserved for evaluation (and excluded from training) each time.

More details of cross validation can be found here. However, you are not allowed to use sklean in your implementation.

Create training and validation folds

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

```
# Create the dataset folds for cross-valdiation.
num_folds = 5

X_train_folds = []
y_train_folds =  []

# ================================================================ #
# START 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]
# ================================================================ #
num_training_exs = X_train.shape[0]
entries_per_fold = int(num_training_exs / num_folds)
rand_indices = np.random.permutation(num_training_exs)
X_train_folds = np.split(X_train[rand_indices], num_folds)
y_train_folds = np.split(y_train[rand_indices], num_folds)
```

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

print(X_train_folds)
print(y_train_folds)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
[[[122.  98.  62. ... 184. 117.  76.]
  [206. 235. 226. ... 135. 120. 101.]
  [121. 106. 102. ... 127. 107.  98.]
  ...
  [113. 123.  64. ... 179. 171. 143.]
  [ 53.  65.  53. ...  49.  50.  41.]
  [135. 163. 168. ...  86.  67.  57.]]

 [[209. 205. 197. ... 193. 161.  92.]
  [ 73. 102.  95. ...  40.  65.  38.]
  [186. 180. 184. ... 112. 117. 103.]
  ...
  [219. 205. 205. ...  18.  16.  30.]
  [ 41.  53.  40. ... 186. 190. 179.]
  [ 48. 112. 172. ...  50.  51.  48.]]

 [[ 67.  60.  44. ... 181. 153. 120.]
  [ 49.  41.  30. ... 204. 184. 178.]
  [150. 161. 174. ...  94. 108. 109.]
  ...
  [ 32.  48.  96. ...  27.  42.  83.]
  [149. 158. 189. ... 127. 132. 140.]
  [190. 195. 181. ... 155. 140. 122.]]

 [[ 16.  37.  79. ...  47.  96. 153.]
  [182. 200. 212. ...  74. 163. 135.]
  [125. 127. 126. ... 106. 132.  91.]
  ...
  [152. 161. 175. ...  96.  79.  79.]
  [ 35.  41.  55. ... 104. 120. 107.]
  [ 32.  43.  60. ...  18.  25.  41.]]

 [[180.  63.  80. ...  20.  19.  17.]
  [114. 110. 104. ...  51.  43.  32.]
  [250. 253. 249. ... 254. 253. 254.]
  ...
  [253. 253. 253. ... 172. 163. 102.]
  [ 55.  60.  68. ... 116. 122. 115.]
  [232. 215. 187. ... 215. 209. 197.]]]
[[2 9 2 ... 7 4 9]
```

```
 [2 4 7 ... 9 8 0]
 [4 6 8 ... 6 8 5]
 [8 8 2 ... 2 4 8]
 [9 3 8 ... 4 5 9]]
```

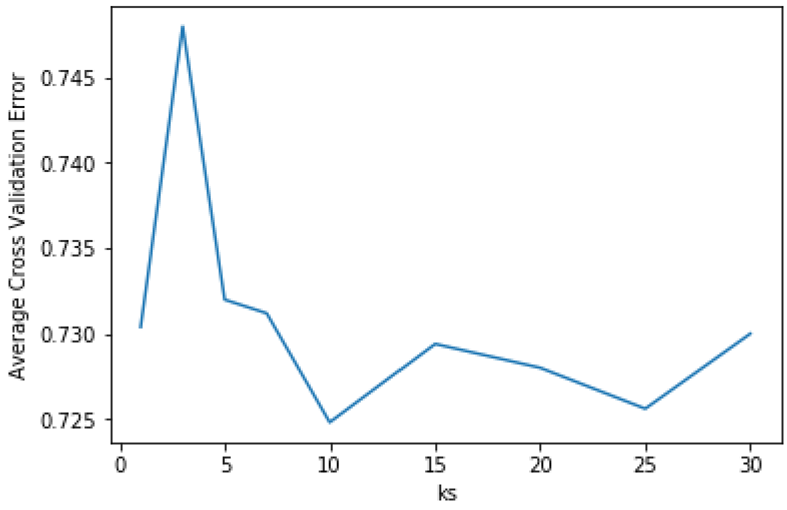
Optimizing the number of nearest neighbors hyperparameter.

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

```
time_start =time.time()

ks = [1, 3, 5, 7, 10, 15, 20, 25, 30]

# ================================================================ #
# START 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. average cross-validation error.
#   Since we assume L2 distance here, please use the vectorized code!
#   Otherwise, you might be waiting a long time.
# ================================================================ #
knn = KNN()
res = np.zeros(len(ks))
for index,k in enumerate(ks):
    error = 0
    for j in range(num_folds):
        x_t_folds = np.concatenate([X_train_folds[fold] for fold in
range(num_folds) if fold != j])
        y_t_folds = np.concatenate([y_train_folds[fold] for fold in
range(num_folds) if fold != j])
        x_test_fold = X_train_folds[j]
        y_test_fold = y_train_folds[j]
        knn.train(X=x_t_folds, y=y_t_folds) #train on the n-1 folds
        distances = knn.compute_L2_distances_vectorized(X=x_test_fold) #
check distances
        pred = knn.predict_labels(distances, k=k) # run prediction
        num_incorrect = np.sum(pred != y_test_fold) #check number of wrong
cases
        error += num_incorrect / y_test_fold.shape[0] #create average
error rate for model
    res[index] = error / num_folds # add average error rate to red[index]

ks_min = ks[np.argsort(res)[0]]
results_min = min(res)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
print('Set k = {0} and get minimum error as
{1}'.format(ks_min,results_min))
```

```
plt.plot(ks,res)
plt.xlabel('ks')
plt.ylabel('Average Cross Validation Error')
plt.show()

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

```
Set k = 10 and get minimum error as 0.7247999999999999
```



```
Computation time: 47.53
```

Questions:

(1) Why do we typically choose k as an odd number (for exmple in `ks`)

(2) What value of k is best amongst the tested k's? What is the cross-validation error for this value of k?

Answers

1. An odd number for `k` is generally done to avoid situations in which a datapoint has multiple options for the label it could be assigned. Essentially, it's to overcome ties.
2. This value depends on the subset we took of the data at the start, but for the version on this notebook, the best `k=10`, and the cross-validation error is ≈0.7248

Evaluating the model on the testing dataset.

Now, given the optimal k which you have learned, evaluate the testing error of the k-nearest neighbors model.

```
error = 1

# ================================================================ #
# START 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_vectorized = knn.compute_L2_distances_vectorized(X=X_test)
pred_labels = knn.predict_labels(dists_L2_vectorized,k=ks_min)
num_samples = pred_labels.shape[0]
num_errors = 0
for i in range(num_samples):
    if pred_labels[i] != y_test[i]:
        num_errors = num_errors + 1
error = num_errors/num_samples

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

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

```
Error rate achieved: 0.718
```

Question:

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

Answers Well, the error went from 0.726 to 0.718, so it's an improvement of 0.08. The dataset may be difficult to separate or classify without further processing of the data, but it's still a pretty good improvement.

End of Homework 3, Part 1 😃

After you've finished both parts the homework, please print out the both of the entire `ipynb` notebooks and `py` files into one PDF file. Make sure you include the output of code cells and answers for questions. Prepare submit it to GradeScope. Do not include any dataset in your submission.