knn_nosol

January 25, 2025

0.1 This is the k-nearest neighbors workbook for ECE C147/C247 Assignment #2

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.

0.2 Import the appropriate libraries

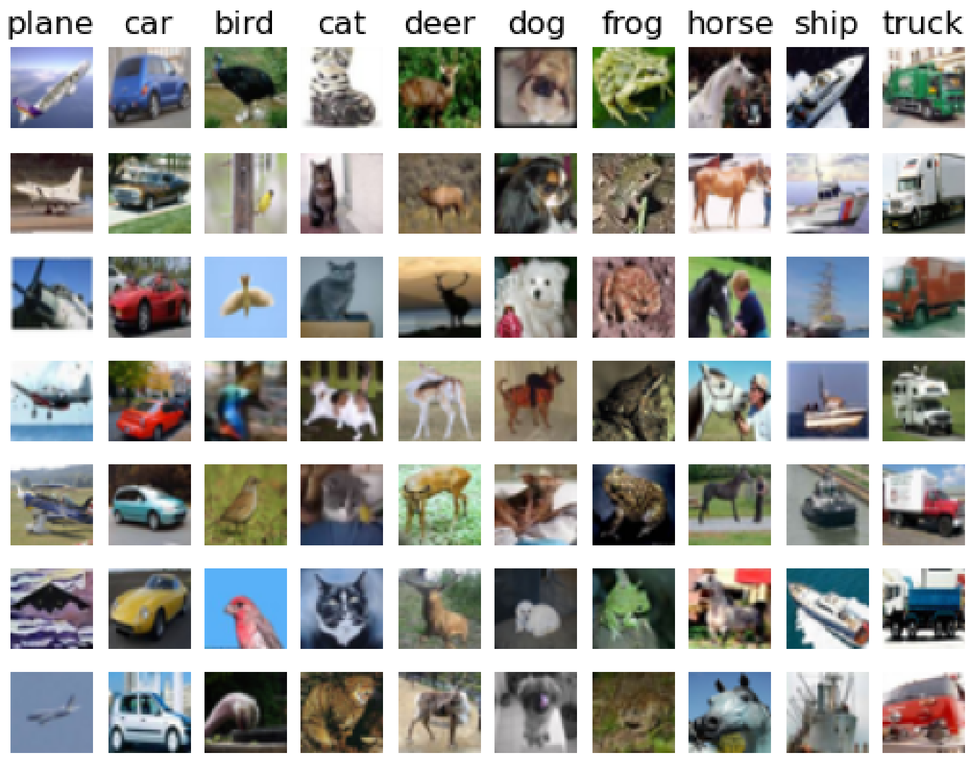
[2]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from utils.data_utils import load_CIFAR10 # function to load the CIFAR-10␣
↪dataset.

# 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
```

[4]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '/Users/ctang/Desktop/ECE_C147/HW2/cifar-10-batches-py' # You␣
↪need to update this line
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,)
```

[6]:
```
# 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', '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()
```



[8]:
```
# 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)
```

1 K-nearest neighbors

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

[15]:
```
# Import the KNN class

from nndl import KNN
```

[17]:
```
# 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)
```

1.1 Questions

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

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

1.2 Answers

(1) In the function knn.train(), the input features (X) and their corresponding labels (y) are stored in memory. Each row of X will represent a data point in which the dimensions correspond to the features. The points are plotted in the space as vectors and stored.

(2) The training step of the KNN is is very simple since storing the input data (X) and the labels corresponding to them (y) without needing to build a complex function. Additionally, the

model allows for minimal computation during training so it is easy handle updates to the dataset. However, storing all of the points and their corresponding features and labels means that it requires a lot of space and memory to store all of the data points. Plus, KNN will be sensistive to noisy data and irrelevant features so it will be less accurate on the predictions unless the data is preprocessed before training.

1.3 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.

[22]:
```
# 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: 14.366737127304077
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.

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

1.3.1 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.

[25]:
```
# 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: 0.17363977432250977
Difference in L2 distances between your KNN implementations (should be 0): 0.0
```

Speedup Depending on your computer speed, you should see a 10-100x speed up from vectorization. On our computer, the vectorized form took 0.36 seconds while the naive implementation took 38.3 seconds.

1.3.2 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.

[32]:
```
# 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.
# ================================================================ #
y_pred = knn.predict_labels(dists_L2_vectorized,1)
errors = (y_test-y_pred)
error = np.count_nonzero(errors)/float(len(y_test))
# ================================================================ #
# 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, considering that chance levels are 10%.

2 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.

2.0.1 Create training and validation folds

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

[34]:
```
# 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]
# ================================================================ #
batch = X_train.shape[0]//num_folds
X_train_folds = np.split(X_train,num_folds)
y_train_folds = np.split(y_train, num_folds)

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

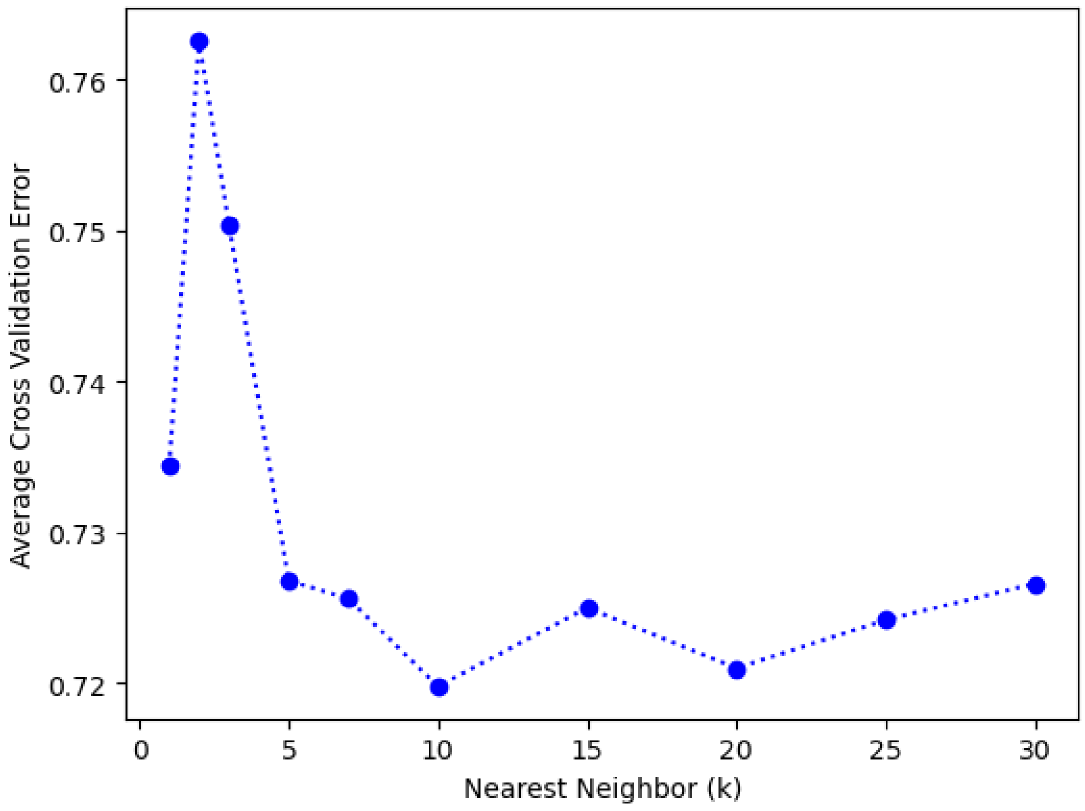
2.0.2 Optimizing the number of nearest neighbors hyperparameter.

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

[67]:
```
time_start =time.time()

ks = [1, 2, 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.
# ================================================================ #

def CV_error(num_training, num_of_folds, k_i, norm_i = None):
    cv_total_error = 0
    for i in range(num_of_folds):
        X_fold_validation = X_train_folds[i]
        y_fold_validation = y_train_folds[i]
        X_fold_train = np.vstack(np.delete(X_train_folds, i, axis = 0))
        y_fold_train = np.hstack(np.delete(y_train_folds, i, axis = 0))
        knn.train(X=X_fold_train, y = y_fold_train)
        if(norm_i):
            dist = knn.compute_distances(X = X_fold_validation, norm = norm_i)
        else:
            dist = knn.compute_L2_distances_vectorized(X=X_fold_validation)
        test_fold_number = num_training/num_of_folds
        y_pred = knn.predict_labels(dists = dist, k = k_i)
        wrong_cases = test_fold_number - np.count_nonzero(y_pred ==␣
↪y_fold_validation)
        cv_total_error += wrong_cases/test_fold_number
    return cv_total_error/num_of_folds
avg_errors = [CV_error(num_training,num_folds, ks[k]) for k in range(len(ks))]

[print("k = {}, Average cross validation error: {}".format(
    ks[k], avg_errors[k])) for k in range(len(ks))]

# Plotting the results
fig, ax = plt.subplots(1,1)
ax.plot(ks, avg_errors, "bo:")
ax.set_xlabel("Nearest Neighbor (k)")
ax.set_ylabel("Average Cross Validation Error")
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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

```
k = 1, Average cross validation error: 0.7344
k = 2, Average cross validation error: 0.7626000000000002
k = 3, Average cross validation error: 0.7504000000000001
k = 5, Average cross validation error: 0.7267999999999999
k = 7, Average cross validation error: 0.7256
k = 10, Average cross validation error: 0.7198
k = 15, Average cross validation error: 0.725
k = 20, Average cross validation error: 0.721
k = 25, Average cross validation error: 0.7242
k = 30, Average cross validation error: 0.7266
Computation time: 23.21
```

[68]:
```
plt.show()
```



2.1 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?

2.2 Answers:

(1) Based on the printed results for each value of k, the best value of k is ten as it has the lowest cross validation error out of all of the values.

(2) The average cross validation error for when k is 10 is 0.7198 or 71.98%

2.2.1 Optimizing the norm

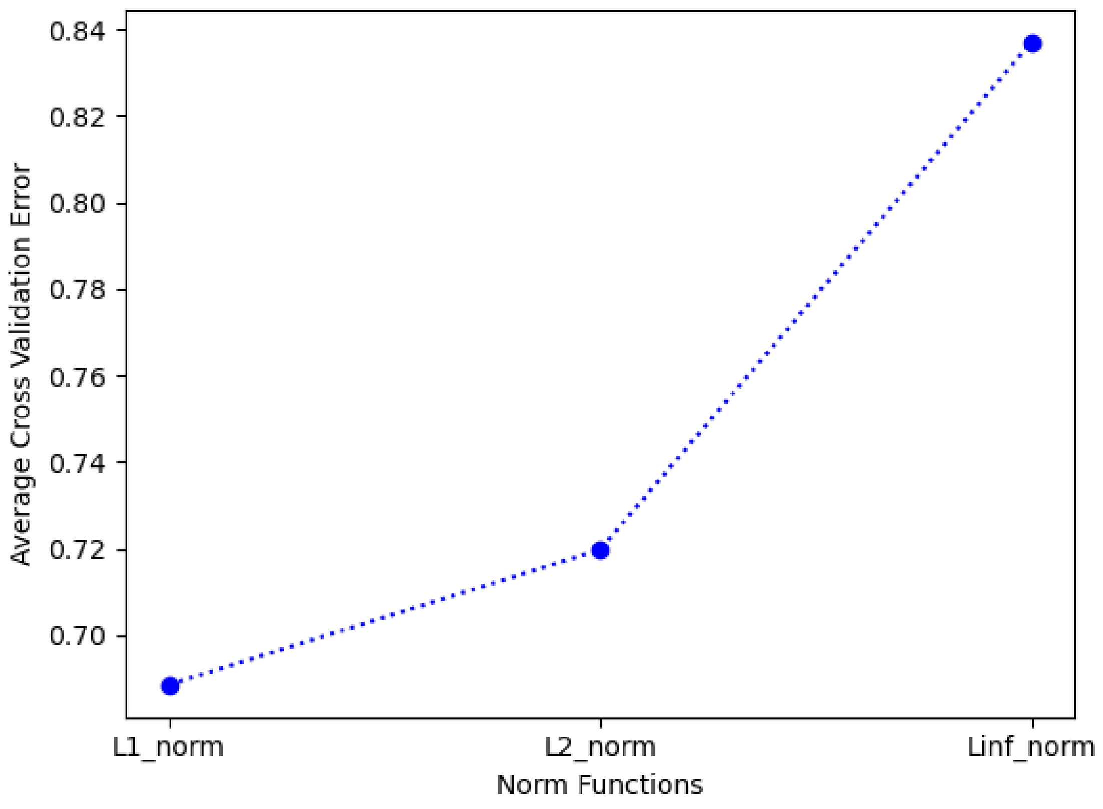
Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

[79]:
```
time_start =time.time()

L1_norm = lambda x: np.linalg.norm(x, ord=1)
L2_norm = lambda x: np.linalg.norm(x, ord=2)
Linf_norm = lambda x: np.linalg.norm(x, ord= np.inf)
```

```
norms = [L1_norm, L2_norm, Linf_norm]

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each norm in norms, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of the norm used vs the cross-validation error
#   Use the best cross-validation k from the previous part.
#
#   Feel free to use the compute_distances function.  We're testing just
#   three norms, but be advised that this could still take some time.
#   You're welcome to write a vectorized form of the L1- and Linf- norms
#   to speed this up, but it is not necessary.
# ================================================================ #
num_of_folds = num_folds
k=10
average_error = [CV_error(num_training, num_of_folds, k_i = k, norm_i = norm)␣
 ↪for norm in norms]
normalization_types = ["L1_norm", "L2_norm", "Linf_norm"]
[print("k = {}, Average cross validation error: {}".
 ↪format(normalization_types[k], avg_errors[k])) for k in range(len(norms))]
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
print('Computation time: %.2f'%(time.time()-time_start))
```

```
k = L1_norm, Average cross validation error: 0.7344
k = L2_norm, Average cross validation error: 0.7626000000000002
k = Linf_norm, Average cross validation error: 0.7504000000000001
Computation time: 380.45
```

[90]:
```
fig, ax = plt.subplots(1, 1)
ax.plot(average_error, "bo:")
ax.set_xlabel("Norm Functions")
ax.set_xticks(np.arange(3), normalization_types)
ax.set_ylabel("Average Cross Validation Error")
plt.show()
```



2.3 Questions:

(1) What norm has the best cross-validation error?

(2) What is the cross-validation error for your given norm and k?

2.4 Answers:

(1) The L1 norm has the best cross validation error out of the 3 types of norms possible.

(2) The cross validation error for my L1_norm was 0.7344 or 73.44%

3 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.

[102]:
```
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.
# ================================================================ #
# Optimal k = 10, L1 Norm
optimal_norm = lambda x: np.linalg.norm(x, ord= 1)
knn.train(X = X_train, y = y_train)
test_dists = knn.compute_distances(X = X_test, norm = optimal_norm)
test_predictions = knn.predict_labels(dists = test_dists, k = 10)

total_incorrect = num_test - np.count_nonzero(test_predictions == y_test)
total_error = total_incorrect/num_test

# Naive baseline: k = 1, L2-norm
baseline_norm = lambda x: np.linalg.norm(x, ord=2)
baseline_dists = knn.compute_distances(X=X_test, norm=baseline_norm)
baseline_predictions = knn.predict_labels(dists=baseline_dists, k=1)


print(f"Baseline error (k=1, L2-norm): {baseline_error}")
print(f"Optimal error (k=10, L1-norm): {total_error}")
print(f"Improvement in error: {error_improvement}")


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

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

```
Baseline error (k=1, L2-norm): 0.726
Optimal error (k=10, L1-norm): 0.722
Improvement in error: 0.0040000000000000036
Error rate achieved: 1
```

3.1 Question:

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

3.2 Answer:

There wasn't that much of an improvement. The baseline error rate was actually higher than the optimal error rate, but only by a marginal number of 0.0004. So in this case, naively choosing k=1 and L2 norm isn't much of an improvement for our error.