knn

October 21, 2019

Assignment1

1. http://cs231n.github.io/assignments2019/assignment1/ 참고하여 아래의 문제를 해결하세요.

a. Q1: K-Nearest Neighbor classifier 의 결과를 작성한 코드와 함께 출력하세요. (10점)

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

        import random
        import numpy as np
        from cs231n.data_utils import load_CIFAR10
        import matplotlib.pyplot as plt

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

        # Some more magic so that the notebook will reload external python modules;
        # see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
        %load_ext autoreload
        %autoreload 2
```

```
In [2]: # Load the raw CIFAR-10 data.
        cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'

        # Cleaning up variables to prevent loading data multiple times (which may cause memory
        try:
           del X_train, y_train
           del X_test, y_test
           print('Clear previously loaded data.')
        except:
           pass

        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 [3]: # 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', 'tru
        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 [4]: # 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)
```

```
In [5]: 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)
```

We would now like to classify the test data with the kNN classifier. Recall that we can break down this process into two steps:

1. First we must compute the distances between all test examples and all train examples.
2. Given these distances, for each test example we find the k nearest examples and have them vote for the label

Lets begin with computing the distance matrix between all training and test examples. For example, if there are Ntr training examples and Nte test examples, this stage should result in a Nte x Ntr matrix where each element (i,j) is the distance between the i-th test and j-th train example.

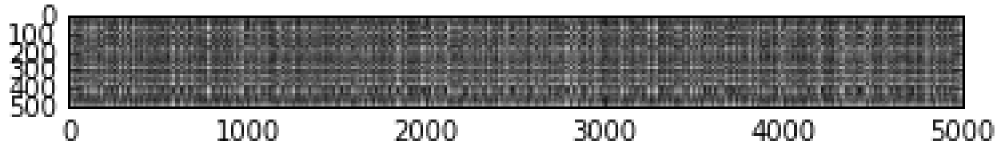
Note: For the three distance computations that we require you to implement in this notebook, you may not use the np.linalg.norm() function that numpy provides.

First, open `cs231n/classifiers/k_nearest_neighbor.py` and implement the function `compute_distances_two_loops` that uses a (very inefficient) double loop over all pairs of (test, train) examples and computes the distance matrix one element at a time.

```
In [6]: # 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 [7]: # 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?

YourAnswer : 현재의 작업은 image의 pixel 값을 기반으로 L2 distance를 계산을 하기 때문에 dists matrix의 row는 테스트 image를 나타내고, column은 train data를 나타낸다. 즉, dists[i, j]는 ith 테스트 image point 와 jth training point 사이의 Euclidean distance이다. bright rows는 하나의 테스트 image가 모든 trainig images와 거리가 high distance를 의미한다. 그리고 columns는 하나의 trainig image와 모든 테스트 images와 Euclidean distance를 기준으로 비교한 값을 의미한다.

```
In [8]: # 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' % (num_correct, num_test, accuracy))
```

```
Got 137 / 500 correct => accuracy: 0.274000
```

You should expect to see approximately 27% accuracy. Now lets try out a larger `k`, say `k = 5`:

```
In [9]: 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' % (num_correct, num_test, accuracy))
```

```
Got 139 / 500 correct => accuracy: 0.278000
```

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

Inline Question 2

We can also use other distance metrics such as L1 distance. For pixel values $p_{ij}^{(k)}$ at location (i,j) of some image I_k,

the mean μ across all pixels over all images is

$$\mu = \frac{1}{nhw}\sum_{k=1}^{n}\sum_{i=1}^{h}\sum_{j=1}^{w} p_{ij}^{(k)}$$

And the pixel-wise mean μ_{ij} across all images is

$$\mu_{ij} = \frac{1}{n}\sum_{k=1}^{n} p_{ij}^{(k)}.$$

The general standard deviation σ and pixel-wise standard deviation σ_{ij} is defined similarly.

Which of the following preprocessing steps will not change the performance of a Nearest Neighbor classifier that uses L1 distance? Select all that apply. 1. Subtracting the mean μ $(\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu.)$ 2. Subtracting the per pixel mean μ_{ij} $(\tilde{p}_{ij}^{(k)} = p_{ij}^{(k)} - \mu_{ij}.)$ 3. Subtracting the mean μ and dividing by the standard deviation σ. 4. Subtracting the pixel-wise mean μ_{ij} and dividing by the pixel-wise standard deviation σ_{ij}. 5. Rotating the coordinate axes of the data.

$\mathit{YourAnswer}:$ 5

$\mathit{YourExplanation}:$

1, 2, 3, 4는 data의 전처리 과정의 normalization 또는 standardization으로 데이터 전처리의 과정들이다. 이러한 normalization은 input feature가 다른 규모(scale) 또는 단위를 가지는 경우에는 유용하다. 하지만 이미지의 경우에는 pixel의 경우에은 상대적인 크기 규모가 0에서 255의 값으로 분포하고 단위도 같기 때문에 꼭 필요한거는 아니지만 zero-center는 일반적으로 성능에 영향을 끼친다.

하지만 5번의 경우에는 단순이 데이터를 축으로 회전을 하는 것이기 때문에 우리가 구분하고자 하는 데이터 분포는 유지하기 때문에 성능에 영향을 끼치지 않는다. 예를 들어, 3차원 공간에서 데이터를 z 축을 중심으로 구나 도넛 모양의 데이터를 회전을 한다고 해도 원래의 데이터 분포는 유지하기 때문에 성능에 영향을 끼치지 않는다.

```
In [10]: # 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('One loop 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')
```

```
One loop difference was: 0.000000
Good! The distance matrices are the same
```

```
In [11]: # 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('No loop 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')
```

```
No loop difference was: 0.000000
Good! The distance matrices are the same
```

```
In [12]: # 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 exec
             """
             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

    # NOTE: depending on what machine you're using,
    # you might not see a speedup when you go from two loops to one loop,
    # and might even see a slow-down.
```

```
Two loop version took 24.089093 seconds
One loop version took 61.188157 seconds
No loop version took 0.260299 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.

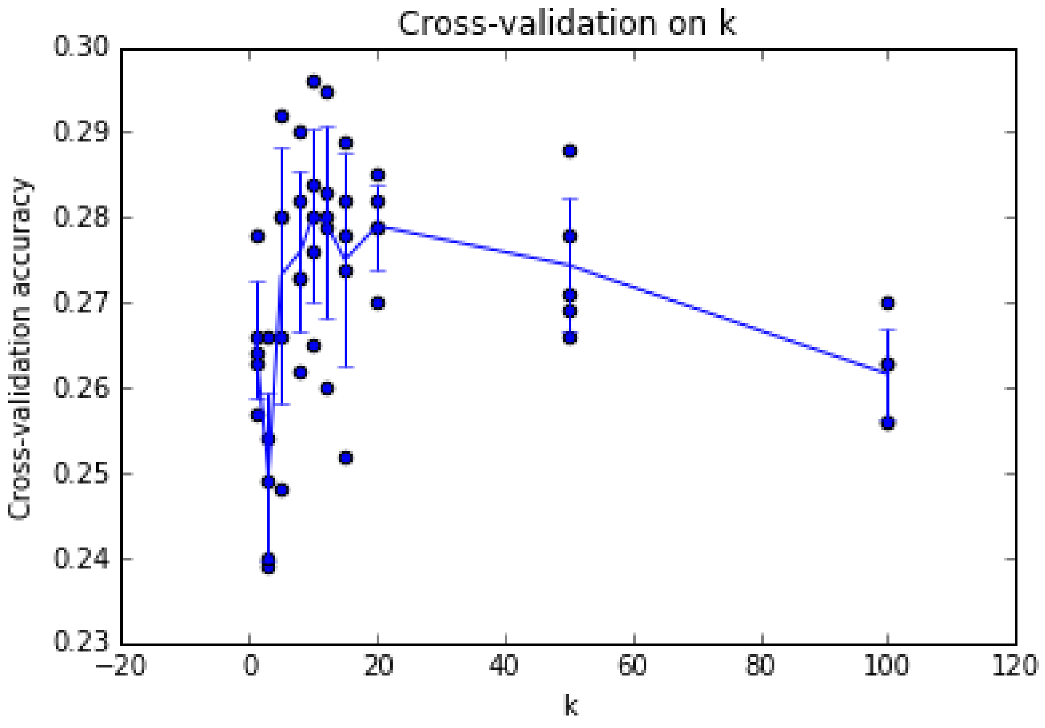
```
In [13]: num_folds = 5
         k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

         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.                                #
         ################################################################################
         # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

         #reference split fucntion https://docs.scipy.org/doc/numpy/reference/generated/numpy.
         X_train_folds = np.split(X_train, num_folds)
         y_train_folds = np.split(y_train, num_folds)
```

```
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # 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.                               #
        ################################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        for k in k_choices:
            k_to_accuracies[k] = []
            for i in range(num_folds):
                # for training with 4 folds
                # numpy concatenate function
                # reference https://docs.scipy.org/doc/numpy/reference/generated/numpy.concat
                X_train_fold = np.concatenate([val for idx, val in enumerate(X_train_folds) i
                y_train_fold = np.concatenate([val for idx, val in enumerate(y_train_folds) i

                # train with k-1 folds
                classifier.train(X_train_fold, y_train_fold)

                # predict with 1 folds(validation set)
                y_test_pred = classifier.predict(X_train_folds[i], k=k, num_loops=0)

                # Compute and print the fraction of correctly predicted eamples
                num_correct = np.sum(y_test_pred == y_train_folds[i])
                accuracy = float(num_correct) / X_train_folds[i].shape[0]
                k_to_accuracies[k].append(accuracy)


        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # 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 = 0.263000
k = 1, accuracy = 0.257000
```

```
k = 1, accuracy = 0.264000
k = 1, accuracy = 0.278000
k = 1, accuracy = 0.266000
k = 3, accuracy = 0.239000
k = 3, accuracy = 0.249000
k = 3, accuracy = 0.240000
k = 3, accuracy = 0.266000
k = 3, accuracy = 0.254000
k = 5, accuracy = 0.248000
k = 5, accuracy = 0.266000
k = 5, accuracy = 0.280000
k = 5, accuracy = 0.292000
k = 5, accuracy = 0.280000
k = 8, accuracy = 0.262000
k = 8, accuracy = 0.282000
k = 8, accuracy = 0.273000
k = 8, accuracy = 0.290000
k = 8, accuracy = 0.273000
k = 10, accuracy = 0.265000
k = 10, accuracy = 0.296000
k = 10, accuracy = 0.276000
k = 10, accuracy = 0.284000
k = 10, accuracy = 0.280000
k = 12, accuracy = 0.260000
k = 12, accuracy = 0.295000
k = 12, accuracy = 0.279000
k = 12, accuracy = 0.283000
k = 12, accuracy = 0.280000
k = 15, accuracy = 0.252000
k = 15, accuracy = 0.289000
k = 15, accuracy = 0.278000
k = 15, accuracy = 0.282000
k = 15, accuracy = 0.274000
k = 20, accuracy = 0.270000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.279000
k = 20, accuracy = 0.282000
k = 20, accuracy = 0.285000
k = 50, accuracy = 0.271000
k = 50, accuracy = 0.288000
k = 50, accuracy = 0.278000
k = 50, accuracy = 0.269000
k = 50, accuracy = 0.266000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.270000
k = 100, accuracy = 0.263000
k = 100, accuracy = 0.256000
k = 100, accuracy = 0.263000
```

```
In [14]: # 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 [15]: # 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 28% accuracy on the test data.
         best_k = 1

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

```
Got 137 / 500 correct => accuracy: 0.274000
```

Inline Question 3

Which of the following statements about k-Nearest Neighbor (k-NN) are true in a classification setting, and for all k? Select all that apply. 1. The decision boundary of the k-NN classifier is linear. 2. The training error of a 1-NN will always be lower than that of 5-NN. 3. The test error of a 1-NN will always be lower than that of a 5-NN. 4. The time needed to classify a test example with the k-NN classifier grows with the size of the training set. 5. None of the above.

YourAnswer : 1 and 4

YourExplanation : k-nn classifier는 non-linear한 경우도 존재한다. 그리고 traing과 test의 error는 같은 K여도 가지고 있는 데이터 분포에 따라 달라질수 있다. 하지만, training error에서 K가 1이면 nearest neighbor는 자기 자신이기 때문에 K가 1일때 trainig error는 0이다. 그리고 training set이 증가하면 하나의 test example은 모든 training examples들과 비교해야 하는 시간이 증가하기 때문에 분류 시간이 늘어난다.