

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 transferring 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.

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]:

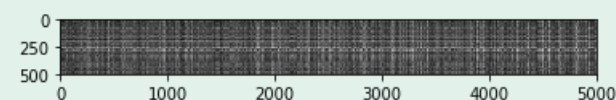
```
# Open cs231n/classifiers/k_nearest_neighbor.py and implement
# compute_distances_two_loops.
```

```
# 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?

Your Answer: fill this in.

1. This test point is very different from most samples in the training data set. It is probably from another class.
2. This training data point is very different from all test points.

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 let's 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.

Your Answer:

1,2,3

Your Explanation:

For L1 distance, the difference between two values will not change when both of the values subtract the same number, no matter which kind of mean that number is, so 1, 2 are correct. $\|x^{(i)} - x^{(j)}\| = \|(x^{(i)} - \mu) - (x^{(j)} - \mu)\|$

3 is just normalizing the data, the ordering of distance will not change after normalizing. So 3 is also correct.

4 is wrong because the pixel-wise standard deviation is different for each pixel and the relative magnitude of the sum of difference will change. The performance of L1 distance will change.

5 is wrong because L1 distance will change when rotating the coordinate axes of the data. For example, there are 3 points $x(1,0)$, $y(0,1)$, $z(0,0)$. The L1 distance between x and y is 2, the L1 distance between x and z is 1, the L1 distance between y and z is 1. If we rotate the coordinate axes, the L1 distances will change.

$y(0,1)$, $z(3,0)$. The L1 distances between x , y is 2, the L1 distances between x , z is 2. They are the same.

Now rotate the axes clockwise by 45 degrees, the L1 distances between x , y becomes $\sqrt{2}$, the L1 distances between x , z is $2\sqrt{2}$. So 5 is wrong.

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

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

Two loop version took 39.371335 seconds
One loop version took 93.376822 seconds
No loop version took 0.643046 seconds

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 [14]:

```
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)*****

X_train_folds = np.array_split(X_train, num_folds)
y_train_folds = np.array_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):
        # All elements except the i-th one should be concatenated to be the train set
        X_train_set = np.concatenate(X_train_folds[0: i] + X_train_folds[i + 1: num_folds])
        y_train_set = np.concatenate(y_train_folds[0: i] + y_train_folds[i + 1: num_folds])

        # The i-th element is chosen to be the validation set
        X_validation_set = X_train_folds[i]
        y_validation_set = y_train_folds[i]

        classifier = KNearestNeighbor()
        classifier.train(X_train_set, y_train_set)
        y_validation_pred = classifier.predict(X_validation_set, k = k)
```

```

# num_test = X_validation_set.shape[0]
# num_correct = np.sum(y_validation_pred == y_validation_set)
# accuracy = float(num_correct) / float(num_test)

# The concise way to calculate accuracy
accuracy = np.mean(y_validation_pred == y_validation_set)

k_to_accuracies[k].append(accuracy)

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

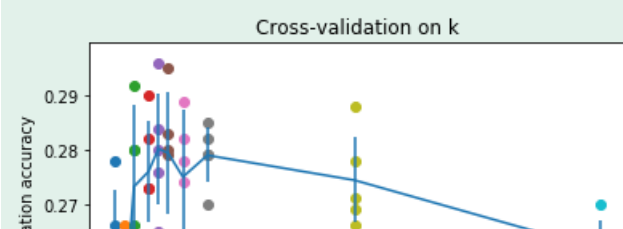
# Print out the computed accuracies
for k in sorted(k_to_accuracies):
    for accuracy in k_to_accuracies[k]:
        print('k = %d, accuracy = %f' % (k, accuracy))

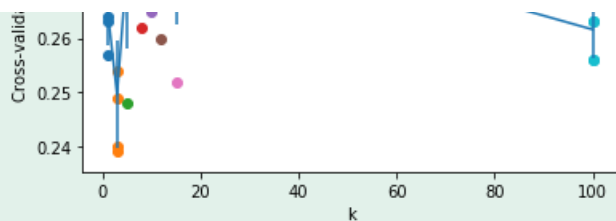
```

```

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 [16]:

```
# 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 = 10

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 141 / 500 correct => accuracy: 0.282000
```

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.

Your Answer: 2,4

Your Explanation:

1. False. When k is not 1 the decision boundary can be curved instead of linear.
2. True. For 1-NN, the training error is always 0 since the data point has been used to train the model and the one nearest neighbor is just itself. But the training error is no less than 0 for 5-NN.
3. False. From the cross-validation plot above we can see the test result of 5-NN can be more accurate than that of 1-NN.
4. True. Because in order to classify a test sample we need to explore the entire data set and sort the difference. When the size of training set gets larger, more time will be needed.