실습1: knn

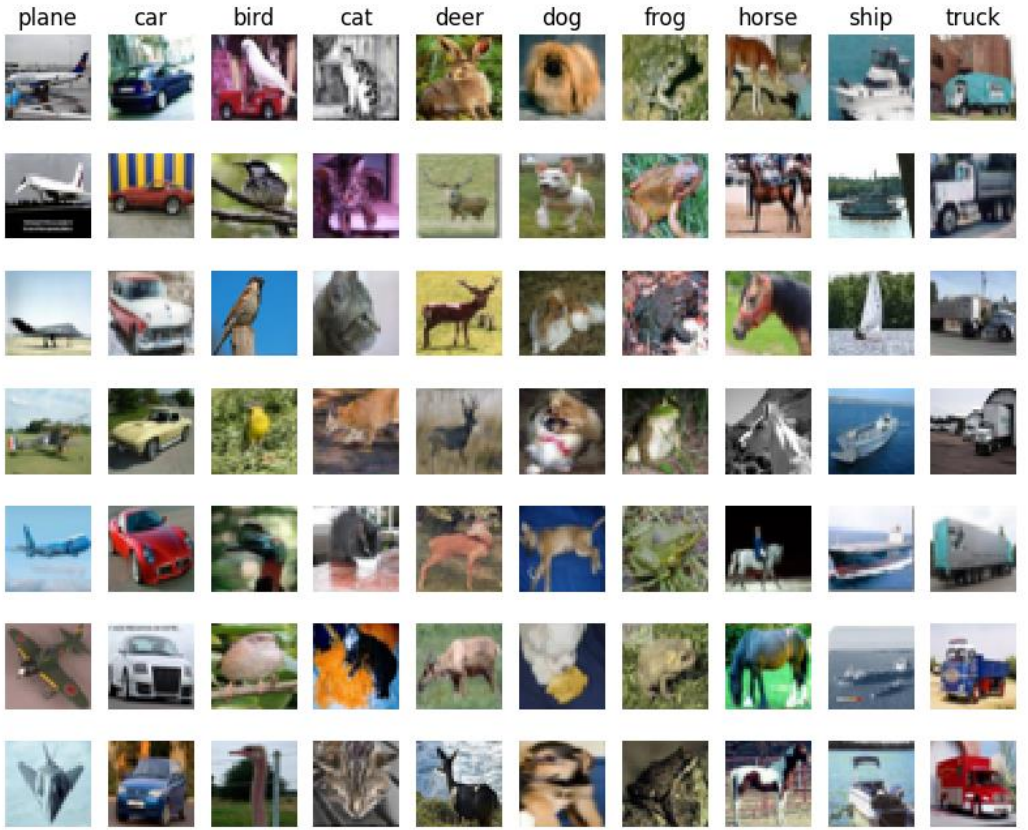
CIFAR-10 데이터셋 로드

```
# Load the raw CIFAR-10 data.
cifar10_dir = 'AI/datasets/cifar-10-batches-py'

# Cleaning up variables to prevent loading data multiple times (which may cause memory issue)
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,)
```

CIFAR-10 데이터셋 샘플 그리기

```
# 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()
```



CIFAR-10 데이터셋 샘플 그리기

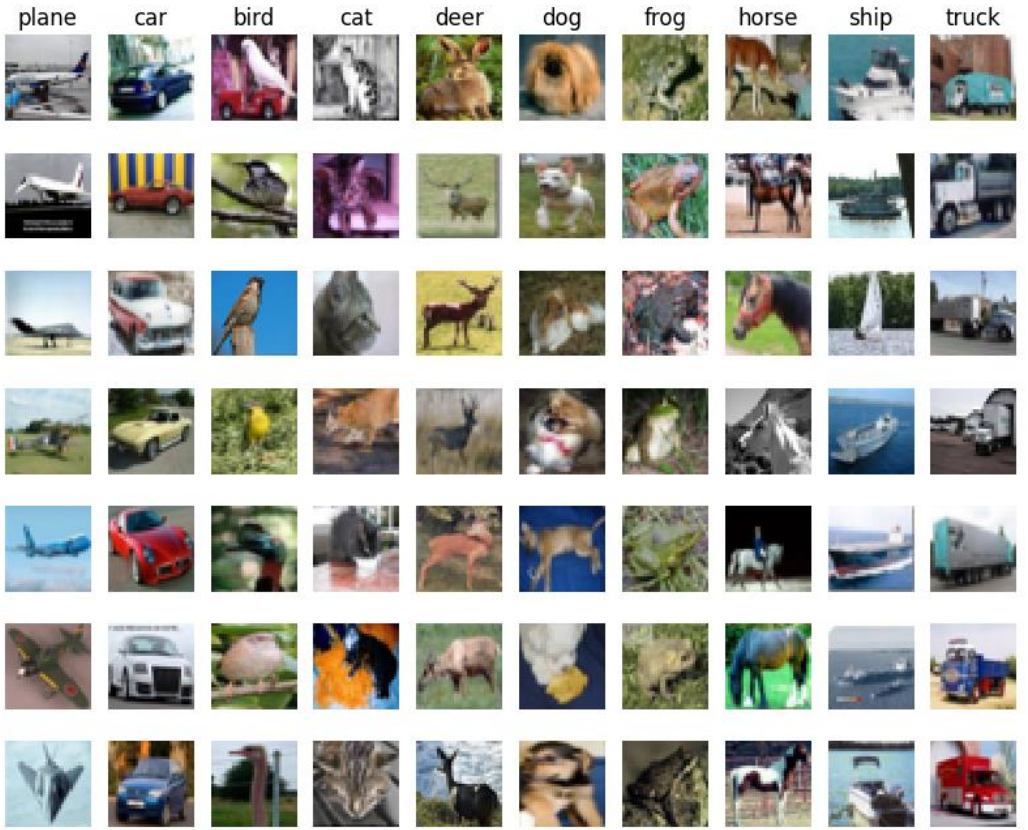
- np.flatnonzero
  - 주어진 배열에서 0이 아닌 원소의 인덱스를 반환
  - 1차원 형태(즉, flat된)로 인덱스를 반환
  - y_train==y는 두 배열의 각 원소가 서로 같은지 여부를 나타내는 논리형 (Boolean) 배열

```
# 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()
```

CIFAR-10 데이터셋 샘플 그리기

- np.random_choice
  - 해당 인덱스들 중 몇 개를 무작위로 선택

```
# 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()
```



훈련세트, 테스트세트 선택

```
# 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)
```

KNearestNeighbor 객체 생성

```
from AI.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)
```

구현1: compute_distances_two_loops 구현

- AI1/AI/classifiers/k_nearest_neighbor.py 파일 안의 compute_distances_two_loops 구현 필요

```
# Open AI/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)
```

```
def compute_distances_two_loops(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using a nested loop over both the training data and the
    test data.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
        for j in range(num_train):
            #####################################################################
            # TODO:                                                             #
            # Compute the l2 distance between the ith test point and the jth    #
            # training point, and store the result in dists[i, j]. You should   #
            # not use a loop over dimension, nor use np.linalg.norm().          #
            #####################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            pass

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

구현1: compute_distances_two_loops 구현

```
def compute_distances_two_loops(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using a nested loop over both the training data and the
    test data.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
        for j in range(num_train):
            #####################################################################
            # TODO:                                                             #
            # Compute the l2 distance between the ith test point and the jth    #
            # training point, and store the result in dists[i, j]. You should   #
            # not use a loop over dimension, nor use np.linalg.norm().          #
            #####################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            pass

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

- 테스트 이미지들 X의 각 이미지와 학습세트 self.X_train의 각 이미지 간의 거리를 모두 구해야함
- X: (500, 3072)
- X_train: (5000, 3072)
- dists: (500, 5000)
- dists[i,j]는 i번째 테스트 이미지와 j번째 훈련세트 이미지 사이의 (L2) 거리를 저장

구현1: compute_distances_two_loops 구현

```
def compute_distances_two_loops(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using a nested loop over both the training data and the
    test data.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
        for j in range(num_train):
            #####################################################################
            # TODO:                                                             #
            # Compute the l2 distance between the ith test point and the jth    #
            # training point, and store the result in dists[i, j]. You should   #
            # not use a loop over dimension, nor use np.linalg.norm().          #
            #####################################################################
            # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

            pass

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

- 즉, X[i]와 self.X_train[j] 사이의 L2 거리를 계산하여 dists[i, j]에 저장

$$I_1, I_2 \in \mathbb{R}^{3072}$$

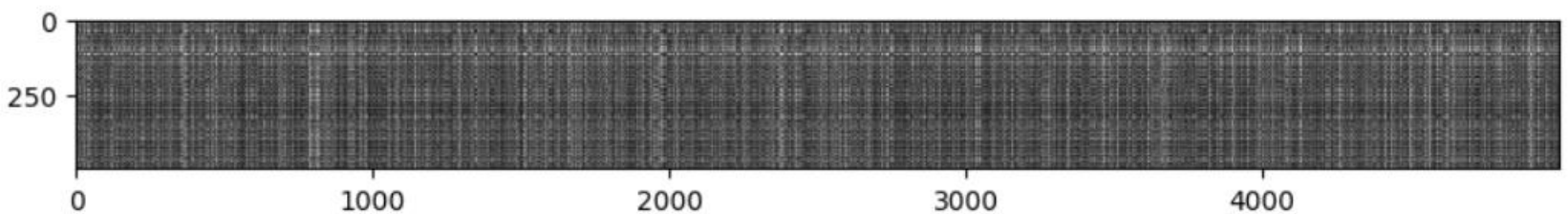
$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

거리 계산

```
# Open AI/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)
```

```
# 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()
```



구현2: predict_labels 구현

- AI1/AI/classifiers/k_nearest_neighbor.py 파일 안의 predict_labels 함수 구현 필요

```
# 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))
```

구현2: predict_labels 구현

- 목표: 각 테스트 이미지에 대한 정답 레이블 예측
- 각 테스트 이미지에 대해, 가장 거리가 짧은 k개의 훈련세트 이미지를 찾음
- 각각의 i에 대해 dists[i,j]가 가장 작은 k개의 훈련세트 이미지의 정답 레이블들을 저장하는 리스트 closest_y 생성

```
def predict_labels(self, dists, k=1):
    """
    Given a matrix of distances between test points and training points,
    predict a label for each test point.

    Inputs:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      gives the distance betwen the ith test point and the jth training point.

    Returns:
    - y: A numpy array of shape (num_test,) containing predicted labels for the
      test data, where y[i] is the predicted label for the test point X[i].
    """
```

```
num_test = dists.shape[0]
y_pred = np.zeros(num_test)
for i in range(num_test):
    # A list of length k storing the labels of the k nearest neighbors to
    # the ith test point.
    closest_y = []
    ###########################################################################
    # TODO:                                                                   #
    # Use the distance matrix to find the k nearest neighbors of the ith      #
    # testing point, and use self.y_train to find the labels of these         #
    # neighbors. Store these labels in closest_y.                             #
    # Hint: Look up the function numpy.argsort.                               #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    pass

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    ###########################################################################
    # TODO:                                                                   #
    # Now that you have found the labels of the k nearest neighbors, you      #
    # need to find the most common label in the list closest_y of labels.     #
    # Store this label in y_pred[i]. Break ties by choosing the smaller       #
    # label.                                                                  #
    ###########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    pass

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

return y_pred
```

구현2: predict_labels 구현

- np.argsort
  - 주어진 배열의 요소를 정렬하는 대신 정렬된 값들이 원래 어떤 순서의 인덱스를 가지는지 반환

```python
import numpy as np

arr = np.array([4, 2, 1, 3])
sorted_indices = np.argsort(arr)
print(sorted_indices)  # [2, 1, 3, 0]
```

- np.bincount
  - 정수 배열에 나타난 각 값의 발생 횟수를 계산

```python
import numpy as np

x = np.array([0, 1, 1, 3, 2, 1, 7])
count = np.bincount(x)

print(count)  # 출력: [1 3 1 1 0 0 0 1]
```

구현2: predict_labels 구현

- 정확도가 27% 정도 나오면 제대로 구현된 것

```
# 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))
```

- k=5를 사용하면 조금 더 나은 정확도를 얻음

```
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))
```

구현3: classifier.compute_distances_one_loop

```
# 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')
```

```
def compute_distances_one_loop(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using a single loop over the test data.

    Input / Output: Same as compute_distances_two_loops
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
        #######################################################################
        # TODO:                                                               #
        # Compute the l2 distance between the ith test point and all training #
        # points, and store the result in dists[i, :].                        #
        # Do not use np.linalg.norm().                                        #
        #######################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        pass

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

구현3: classifier.compute_distances_one_loop

- 힌트: X[i]-self.X_train 계산 (브로드캐스팅)
- 그 후, 각 행에 대해 L2 norm을 계산
- 이렇게 얻은 5000 크기 배열이 바로 dists의 i번째 행

```
def compute_distances_one_loop(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using a single loop over the test data.

    Input / Output: Same as compute_distances_two_loops
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    for i in range(num_test):
        #######################################################################
        # TODO:                                                               #
        # Compute the l2 distance between the ith test point and all training #
        # points, and store the result in dists[i, :].                        #
        # Do not use np.linalg.norm().                                        #
        #######################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        pass

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

구현3: classifier.compute_distances_one_loop

```
# 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')
```

구현4: classifier.compute_distances_no_loop

```
# 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')
```

```
def compute_distances_no_loops(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train using no explicit loops.

    Input / Output: Same as compute_distances_two_loops
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))
    #########################################################################
    # TODO:                                                                 #
    # Compute the l2 distance between all test points and all training      #
    # points without using any explicit loops, and store the result in      #
    # dists.                                                                #
    #                                                                       #
    # You should implement this function using only basic array operations; #
    # in particular you should not use functions from scipy,                #
    # nor use np.linalg.norm().                                             #
    #                                                                       #
    # HINT: Try to formulate the l2 distance using matrix multiplication    #
    #       and two broadcast sums.                                         #
    #########################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    pass

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

구현4: classifier.compute_distances_no_loop

- 단순히 브로드캐스팅으로 no loop 구현은 메모리 부족으로 어려움
- X를 (500,1,3072)로, X_train을 (1,5000,3072) 크기로 바꾼 후 뺄셈한 후 제곱들의 합을 구하는건 중간에 500x5000x3072 배열을 필요로 함
- 아래 식을 사용 가능하여 메모리 부족 없이 수행 가능
- 내적 부분은 X와 X_train의 행렬곱으로 구현

$$dists(i,j) = ||x^{(i)} - x^{(j)}_{train}||$$

$$= \sqrt{(x^{(i)} - x^{(j)}_{train}) \cdot (x^{(i)} - x^{(j)}_{train})} = \sqrt{||x^{(i)}||^2 + ||x^{(j)}_{train}||^2 - 2x^{(i)} \cdot x^{(j)}_{train}}$$

시간 비교

```
# 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 execute.
    """
    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 implementation!

# 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.
```

구현5: 교차검증

```
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)*****

pass

# *****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)*****

pass

# *****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))
```

교차검증 결과

- k 최근접 이웃 분류기 정확도 그래프
- 최적의 k 선택

```
# 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()
```

```
# 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))
```

Inline Question

- Inline Question는 할 필요x
- 과제 제출 시 한글로 된 Question에 대한 답은 해야함

과제 1

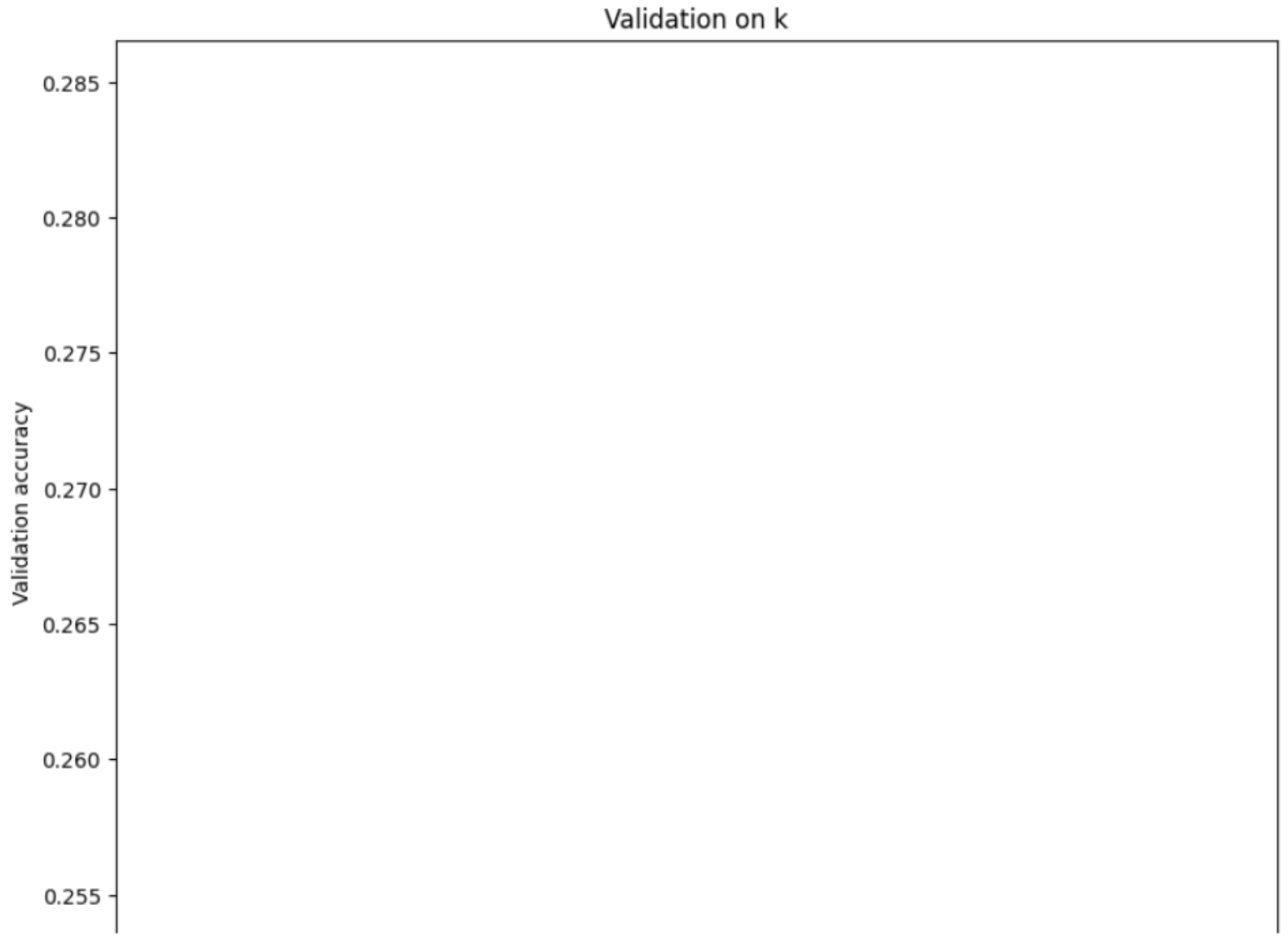
인공지능 수업 과제1

위에서는 교차검증을 수행했었는데 이번에는 교차검증이 아닌 일반 검증을 수행한다. 5000개의 훈련세트 이미지 중 1000개는 검증세트로 사용하고 나머지 4000개는 작아진 훈련세트로 설정한다. 다양한 k값에 대해 검증을 수행한다.

아래 코드의 pass 부분에 내용을 채워야한다. 먼저, k에 대한 검증 정확도 그래프를 그려야한다. 그리고 가장 좋은 검증 정확도를 가진 k를 best_k 값으로 설정한 후 그 k값에 대해 최종적으로 테스트 세트에 대해 분류를 수행하여 정확도를 계산한다.

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

X_train_new = X_train[:4000]
y_train_new = y_train[:4000]
X_val_new = X_train[4000:]
y_val_new = y_train[4000:]

num_train_new = 4000
num_val_new = 1000

k_to_accuracy = {}

# k_choices에 있는 각 k값에 대해 검증정확도를 계산한 후 k_to_accuracy[k]에 저장
pass

for k in k_to_accuracy:
    print('k = %d, accuracy = %f' % (k, k_to_accuracy[k]))

# 각 k에 대한 검증 정확도를 꺾은선 그래프로 그림
# 제목: Validation on k, x축 제목: k, y축 제목: Validation accuracy
pass
```

과제 1


Validation on k
Validation accuracy
0.285
0.280
0.275
0.270
0.265
0.260
0.255

과제 1

```
# 위에서 얻은 그래프를 바탕으로 가장 좋은 성능을 가진 k를 구한 후 아래 best_k에 대입
best_k = 1

classifier = KNearestNeighbor()
classifier.train(X_train_new, y_train_new)
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))
```

과제 1

Question 1

교차 검증이 아닌 일반 검증을 사용하였을 때의 그래프(k - 정확도)를 교차 검증을 사용하였을 때의 그래프와 비교하면 어떠한지 변동성 측면에서 쓰시오. 그 이유는 무엇인가?

답:

Question 2

교차 검증이 아닌 일반 검증을 사용하였을 때의 테스트 정확도는 교차 검증 사용한 경우의 테스트 정확도에 비해 어떠한가?

답: