EECS 498-007/598-005 Assignment 1-2: K-Nearest Neighbors (k-NN)

Before we start, please put your name and UMID in following format

: Firstname LASTNAME, #00000000 // e.g.) Justin JOHNSON, #12345678

Your Answer:
Artem KARPOV

K-Nearest Neighbors (k-NN)

In this notebook you will implement a K-Nearest Neighbors classifier on the CIFAR-10 dataset.

Recall that the K-Nearest Neighbor classifier does the following:

- During training, the classifier simply memorizes the training data
- During testing, test images are compared to each training image; the predicted label is the majority vote among the K nearest training examples.

After implementing the K-Nearest Neighbor classifier, you will use *cross-validation* to find the best value of K.

The goals of this exercise are to go through a simple example of the data-driven image classification pipeline, and also to practice writing efficient, vectorized code in PyTorch.

Install starter code

We have implemented some utility functions for this exercise in the `coutils` package. Run this cell to download and install it.

```
!pip install git+https://github.com/deepvision-class/starter-code
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting git+https://github.com/deepvision-class/starter-code
  Cloning https://github.com/deepvision-class/starter-code to /tmp/pip-req-build-j9e_202e
  Running command git clone -q https://github.com/deepvision-class/starter-code /tmp/pip-req-build-j9e_202e
Requirement already satisfied: pydrive in /usr/local/lib/python3.7/dist-packages (from Colab-Utils==0.1.dev0) (1.3.1)
Requirement already satisfied: oauth2client>=4.0.0 in /usr/local/lib/python3.7/dist-packages (from pydrive->Colab-Utils==0.1.dev0) (4.1.3)
```

1 сек. выполнено в 08:13

```
Requirement already satisfied: PyYAML>=3.0 in /usr/local/lib/python3.7/dist-packages (from pydrive->Colab-Utils==0.1.dev0) (3.13)
Requirement already satisfied: httplib2<1dev,>=0.15.0 in /usr/local/lib/python3.7/dist-packages (from google-api-python-client>=1.2->pydrive->Colab-
Requirement already satisfied: six<2dev,>=1.13.0 in /usr/local/lib/python3.7/dist-packages (from google-api-python-client>=1.2->pydrive->Colab-Utils
Requirement already satisfied: google-auth-httplib2>=0.0.3 in /usr/local/lib/python3.7/dist-packages (from google-api-python-client>=1.2->pydrive->C
Requirement already satisfied: google-auth<3dev,>=1.16.0 in /usr/local/lib/python3.7/dist-packages (from google-api-python-client>=1.2->pydrive->Col
Requirement already satisfied: uritemplate<4dev,>=3.0.0 in /usr/local/lib/python3.7/dist-packages (from google-api-python-client>=1.2->pydrive->Cola
Requirement already satisfied: google-api-core<3dev,>=1.21.0 in /usr/local/lib/python3.7/dist-packages (from google-api-python-client>=1.2->pydrive-
Requirement already satisfied: protobuf<4.0.0dev,>=3.12.0 in /usr/local/lib/python3.7/dist-packages (from google-api-core<3dev,>=1.21.0->google-api-
Requirement already satisfied: setuptools>=40.3.0 in /usr/local/lib/python3.7/dist-packages (from google-api-core<3dev,>=1.21.0->google-api-python-c
Requirement already satisfied: pytz in /usr/local/lib/python3.7/dist-packages (from google-api-core<3dev,>=1.21.0->google-api-python-client>=1.2->py
Requirement already satisfied: googleapis-common-protos<2.0dev,>=1.6.0 in /usr/local/lib/python3.7/dist-packages (from google-api-core<3dev,>=1.21.0
Requirement already satisfied: requests<3.0.0dev,>=2.18.0 in /usr/local/lib/python3.7/dist-packages (from google-api-core<3dev,>=1.21.0->google-api-
Requirement already satisfied: packaging>=14.3 in /usr/local/lib/python3.7/dist-packages (from google-api-core<3dev,>=1.21.0->google-api-python-clie
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-packages (from google-auth<3dev,>=1.16.0->google-api-python-client>=1.
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from google-auth<3dev,>=1.16.0->google-api-python-c
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.7/dist-packages (from google-auth<3dev,>=1.16.0->google-api-python-cl
Requirement already satisfied: pyasn1>=0.1.7 in /usr/local/lib/python3.7/dist-packages (from oauth2client>=4.0.0->pydrive->Colab-Utils==0.1.dev0) (0
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in /usr/local/lib/python3.7/dist-packages (from packaging>=14.3->google-api-core<3dev,>=1.21
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests<3.0.0dev,>=2.18.0->google-api-core<3dev,>=
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests<3.0.0dev,>=2.18.0->g
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests<3.0.0dev,>=2.18.0->google-api-core<3dev,>
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests<3.0.0dev,>=2.18.0->google-api-core<3dev,>=1.21.
```

Setup code

Run some setup code for this notebook: Import some useful packages and increase the default figure size.

```
import coutils
import torch
import torchvision
import matplotlib.pyplot as plt
import statistics

plt.rcParams['figure.figsize'] = (10.0, 8.0)
plt.rcParams['font.size'] = 16
```

Load the CIFAR-10 dataset

The utility function `coutils.data.cifar10()` returns the entire CIFAR-10 dataset as a set of four **Torch tensors**:

- `x_train` contains all training images (real numbers in the range $[0, 1]$)

- `y_train` contains all training labels (integers in the range $[0, 9]$)
- `x_test` contains all test images
- `y_test` contains all test labels

This function automatically downloads the CIFAR-10 dataset the first time you run it.

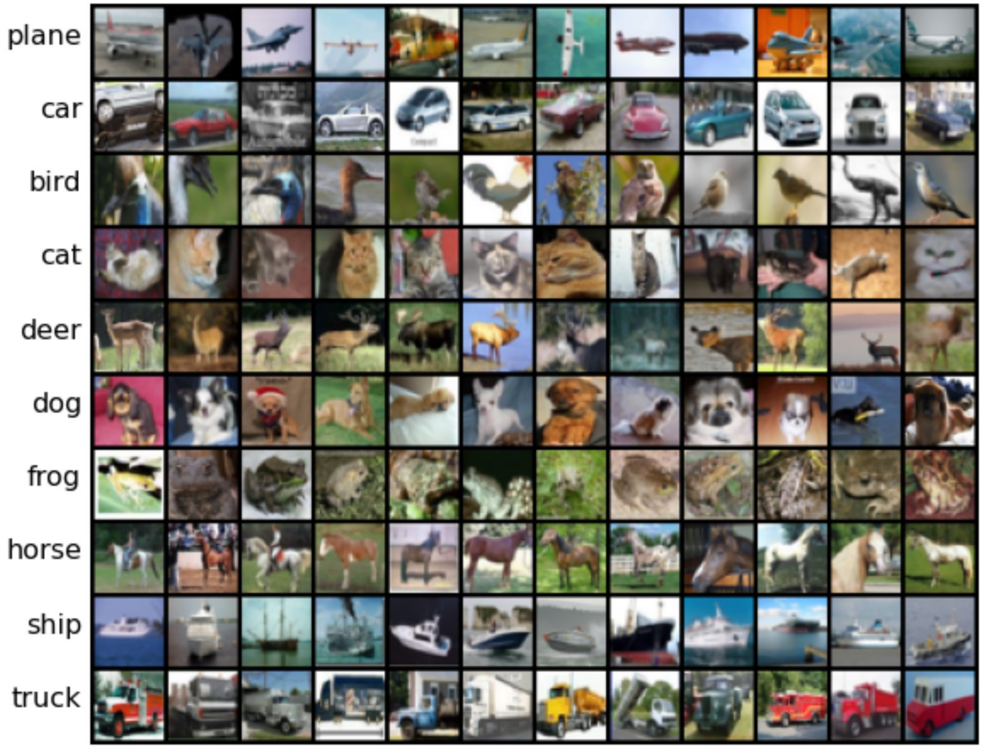
```
x_train, y_train, x_test, y_test = coutils.data.cifar10()

print('Training set:', )
print('  data shape:', x_train.shape)
print('  labels shape: ', y_train.shape)
print('Test set:')
print('  data shape: ', x_test.shape)
print('  labels shape', y_test.shape)
```

```
Training set:
  data shape: torch.Size([50000, 3, 32, 32])
  labels shape:  torch.Size([50000])
Test set:
  data shape:  torch.Size([10000, 3, 32, 32])
  labels shape torch.Size([10000])
```

Visualize the dataset

To give you a sense of the nature of the images in CIFAR-10, this cell visualizes some random examples from the training set.

```
import random
from torchvision.utils import make_grid

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
samples_per_class = 12
samples = []
for y, cls in enumerate(classes):
  plt.text(-4, 34 * y + 18, cls, ha='right')
  idxs = (y_train == y).nonzero().view(-1)
  for i in range(samples_per_class):
    idx = idxs[random.randrange(idxs.shape[0])].item()
    samples.append(x_train[idx])
img = torchvision.utils.make_grid(samples, nrow=samples_per_class)
plt.imshow(coutils.tensor_to_image(img))
plt.axis('off')
plt.show()
```



Subsample the dataset

When implementing machine learning algorithms, it's usually a good idea to use a small sample of the full dataset. This way your code will run much faster, allowing for more interactive and efficient development. Once you are satisfied that you have correctly implemented the algorithm, you can then rerun with the entire dataset.

The function `coutils.data.cifar10()` can automatically subsample the CIFAR10 dataset for us. To see how to use it, we can check the documentation using the built-in `help` command:

```
help(coutils.data.cifar10)
```

```
Help on function cifar10 in module coutils.data:

cifar10(num_train=None, num_test=None)
    Return the CIFAR10 dataset, automatically downloading it if necessary.
    This function can also subsample the dataset.
```

```
Inputs:
- num_train: [Optional] How many samples to keep from the training set.
  If not provided, then keep the entire training set.
- num_test: [Optional] How many samples to keep from the test set.
  If not provided, then keep the entire test set.

Returns:
- x_train: float32 tensor of shape (num_train, 3, 32, 32)
- y_train: int64 tensor of shape (num_train, 3, 32, 32)
- x_test: float32 tensor of shape (num_test, 3, 32, 32)
- y_test: int64 tensor of shape (num_test, 3, 32, 32)
```

We will subsample the data to use only 500 training examples and 100 test examples:

```
num_train = 500
num_test = 250

x_train, y_train, x_test, y_test = coutils.data.cifar10(num_train, num_test)

print('Training set:', )
print('  data shape:', x_train.shape)
print('  labels shape: ', y_train.shape)
print('Test set:')
print('  data shape: ', x_test.shape)
print('  labels shape', y_test.shape)
```

```
Training set:
  data shape: torch.Size([500, 3, 32, 32])
  labels shape:  torch.Size([500])
Test set:
  data shape:  torch.Size([250, 3, 32, 32])
  labels shape torch.Size([250])
```

Compute distances: Naive implementation

Now that we have examined and prepared our data, it is time to implement the kNN classifier. We can break the process down into two steps:

1. Compute the (squared Euclidean) distances between all training examples and all test examples
2. Given these distances, for each test example find its k nearest neighbors and have them vote for the label to output

Lets begin with computing the distance matrix between all training and test examples. First we will implement a naive version of the distance

Lets begin with computing the distance matrix between all training and test examples. First we will implement a naive version of the distance computation, using explicit loops over the training and test sets:

NOTE: When implementing distance functions in this notebook, you may not use the `torch.norm` function (or its instance method variant `x.norm`); you may not use any functions from `torch.nn` or `torch.nn.functional`.

```
def compute_distances_two_loops(x_train, x_test):
  """
  Computes the squared Euclidean distance between each element of the training
  set and each element of the test set. Images should be flattened and treated
  as vectors.

  This implementation uses a naive set of nested loops over the training and
  test data.

  Inputs:
  - x_train: Torch tensor of shape (num_train, C, H, W)
  - x_test: Torch tensor of shape (num_test, C, H, W)

  Returns:
  - dists: Torch tensor of shape (num_train, num_test) where dists[i, j] is the
    squared Euclidean distance between the ith training point and the jth test
    point.
  """
  # Initialize dists to be a tensor of shape (num_train, num_test) with the
  # same datatype and device as x_train
  num_train = x_train.shape[0]
  num_test = x_test.shape[0]
  dists = x_train.new_zeros(num_train, num_test)
  ##############################################################################
  # TODO: Implement this function using a pair of nested loops over the        #
  # training data and the test data.                                           #
  #                                                                            #
  # You may not use torch.norm (or its instance method variant), nor any       #
  # functions from torch.nn or torch.nn.functional.                            #
  ##############################################################################
  for i in range(num_train):
    for j in range(num_test):
      dists[i, j] = (x_train[i, :, :, :] -  x_test[j, :, :, :]).pow(2).sum().sqrt().item()
  ##############################################################################
  #                             END OF YOUR CODE                               #
  ##############################################################################
  return dists
```

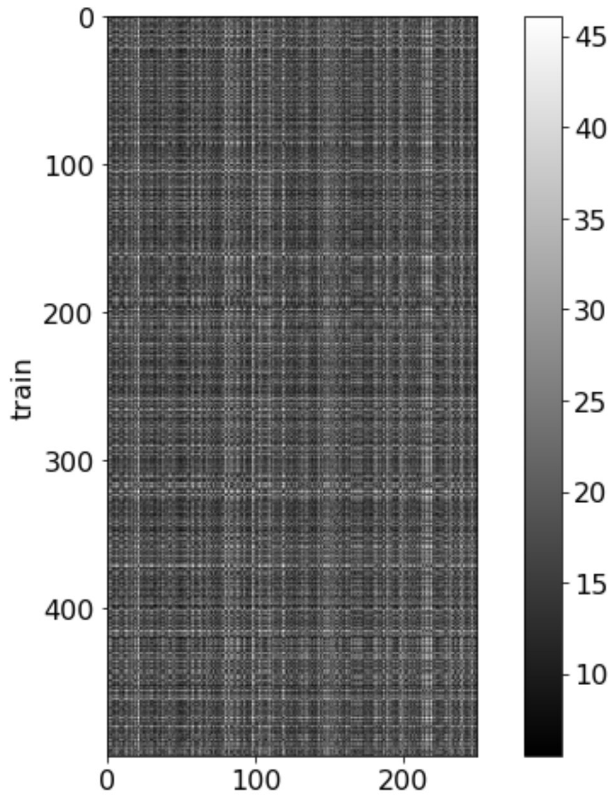
After implementing the function above, we can run it to check that it has the expected shape:

```
num_train = 500
num_test = 250
x_train, y_train, x_test, y_test = coutils.data.cifar10(num_train, num_test)

dists = compute_distances_two_loops(x_train, x_test)
print('dists has shape: ', dists.shape)
```

```
dists has shape:  torch.Size([500, 250])
```

As a visual debugging step, we can visualize the distance matrix, where each row is a test example and each column is a training example.

```
plt.imshow(dists.numpy(), cmap='gray', interpolation='none')
plt.colorbar()
plt.xlabel('test')
plt.ylabel('train')
plt.show()
```



test

Compute distances: Vectorization

Our implementation of the distance computation above is fairly inefficient since it uses nested Python loops over the training and test sets.

When implementing algorithms in PyTorch, it's best to avoid loops in Python if possible. Instead it is preferable to implement your computation so that all loops happen inside PyTorch functions. This will usually be much faster than writing your own loops in Python, since PyTorch functions can be internally optimized to iterate efficiently, possibly using multiple threads. This is especially important when using a GPU to accelerate your code.

The process of eliminating explict loops from your code is called **vectorization**. Sometimes it is straighforward to vectorize code originally written with loops; other times vectorizing requires thinking about the problem in a new way. We will use vectorization to improve the speed of our distance computation function.

As a first step toward vectorizing our distance computation, complete the following implementation which uses only a single Python loop over the training data:

```
def compute_distances_one_loop(x_train, x_test):
  """
  Computes the squared Euclidean distance between each element of the training
  set and each element of the test set. Images should be flattened and treated
  as vectors.

  This implementation uses only a single loop over the training data.

  Inputs:
  - x_train: Torch tensor of shape (num_train, C, H, W)
  - x_test: Torch tensor of shape (num_test, C, H, W)

  Returns:
  - dists: Torch tensor of shape (num_train, num_test) where dists[i, j] is the
    squared Euclidean distance between the ith training point and the jth test
    point.
  """
  # Initialize dists to be a tensor of shape (num_train, num_test) with the
  # same datatype and device as x_train
  num_train = x_train.shape[0]
  num_test = x_test.shape[0]
  dists = x_train.new_zeros(num_train, num_test)
  ##############################################################################
  # TODO: Implement this function using only a single loop over x_train.      #
```

```
#                                                                             #
# You may not use torch.norm (or its instance method variant), nor any       #
# functions from torch.nn or torch.nn.functional.                             #
###############################################################################
for i in range(num_train):
    dists[i] = (x_test[:, :, :, :] - x_train[i, :, :, :]).pow(2).sum(dim=(1,2,3)).sqrt()
###############################################################################
#                              END OF YOUR CODE                               #
###############################################################################
return dists
```

We can check the correctness of our one-loop implementation by comparing it with our two-loop implementation on some randomly generated data.

Note that we do the comparison with 64-bit floating points for increased numeric precision.

```
torch.manual_seed(0)
x_train_rand = torch.randn(100, 3, 16, 16, dtype=torch.float64)
x_test_rand = torch.randn(100, 3, 16, 16, dtype=torch.float64)

dists_one = compute_distances_one_loop(x_train_rand, x_test_rand)
dists_two = compute_distances_two_loops(x_train_rand, x_test_rand)
difference = (dists_one - dists_two).pow(2).sum().sqrt().item()
print('Difference: ', difference)
if difference < 1e-4:
  print('Good! The distance matrices match')
else:
  print('Uh-oh! The distance matrices are different')
```

```
Difference:  0.0
Good! The distance matrices match
```

Now implement a fully vectorized version of the distance computation function that does not use any python loops.

```
def compute_distances_no_loops(x_train, x_test):
  """
  Computes the squared Euclidean distance between each element of the training
  set and each element of the test set. Images should be flattened and treated
  as vectors.

  This implementation should not use any Python loops. For memory-efficiency,
  it also should not create any large intermediate tensors; in particular you
  should not create any intermediate tensors with O(num_train*num_test)
```

```
elements.

Inputs:
- x_train: Torch tensor of shape (num_train, C, H, W)
- x_test: Torch tensor of shape (num_test, C, H, W)

Returns:
- dists: Torch tensor of shape (num_train, num_test) where dists[i, j] is the
  squared Euclidean distance between the ith training point and the jth test
  point.
"""
# Initialize dists to be a tensor of shape (num_train, num_test) with the
# same datatype and device as x_train
num_train = x_train.shape[0]
num_test = x_test.shape[0]
dists = x_train.new_zeros(num_train, num_test)
##############################################################################
# TODO: Implement this function without using any explicit loops and without #
# creating any intermediate tensors with O(num_train * num_test) elements.   #
#                                                                            #
# You may not use torch.norm (or its instance method variant), nor any       #
# functions from torch.nn or torch.nn.functional.                            #
#                                                                            #
# HINT: Try to formulate the Euclidean distance using two broadcast sums     #
#       and a matrix multiply.                                               #
##############################################################################
# Replace "pass" statement with your code
x_train = x_train.view(num_train, -1)
x_test = x_test.view(num_test, -1)
dists = (
  -2 * x_train.mm(x_test.T) +
  x_test.pow(2).sum(dim=1).view(1, -1) +
  x_train.pow(2).sum(dim=1).view(-1, 1)
).sqrt()

"""
# Another version:
dists =  (
  (x_test.view(1, num_test, -1) - x_train.view(num_train, 1, -1))
  .pow(2)
  .sum(dim=(2))
  .sqrt()
)
# But it fails with:
# RuntimeError: CUDA out of memory. Tried to allocate 28.61 GiB (GPU 0; 15.90 GiB total capacity;
# 74.03 MiB already allocated; 15.20 GiB free; 82.00 MiB reserved in total by PyTorch)
"""
##############################################################################
#                             END OF YOUR CODE                               #
```

```
#                               END OF YOUR CODE                               #
################################################################################
return dists
```

As before, we can check the correctness of our implementation by comparing the fully vectorized version against the original naive version:

```
torch.manual_seed(0)
x_train_rand = torch.randn(100, 3, 16, 16, dtype=torch.float64)
x_test_rand = torch.randn(100, 3, 16, 16, dtype=torch.float64)

dists_two = compute_distances_two_loops(x_train_rand, x_test_rand)
dists_none = compute_distances_no_loops(x_train_rand, x_test_rand)
difference = (dists_two - dists_none).pow(2).sum().sqrt().item()
print('Difference: ', difference)
if difference < 1e-4:
  print('Good! The distance matrices match')
else:
  print('Uh-oh! The distance matrices are different')
```

```
Difference:  3.1760544968942397e-13
Good! The distance matrices match
```

We can now compare the speed of our three implementations. If you've implemented everything properly, the one-loop implementation should take less than 4 seconds to run, and the fully vectorized implementation should take less than 0.1 seconds to run.

```
import time

def timeit(f, *args):
  tic = time.time()
  f(*args)
  toc = time.time()
  return toc - tic

torch.manual_seed(0)
x_train_rand = torch.randn(500, 3, 32, 32)
x_test_rand = torch.randn(500, 3, 32, 32)

two_loop_time = timeit(compute_distances_two_loops, x_train_rand, x_test_rand)
print('Two loop version took %.2f seconds' % two_loop_time)

one_loop_time = timeit(compute_distances_one_loop, x_train_rand, x_test_rand)
speedup = two_loop_time / one_loop_time
print('One loop version took %.2f seconds (%.1fX speedup)'
      % (one_loop_time, speedup))
```

```

no_loop_time = timeit(compute_distances_no_loops, x_train_rand, x_test_rand)
speedup = two_loop_time / no_loop_time
print('No loop version took %.2f seconds (%.1fX speedup)'
      % (no_loop_time, speedup))
```

```
Two loop version took 9.26 seconds
One loop version took 0.66 seconds (13.9X speedup)
No loop version took 0.02 seconds (555.5X speedup)
```

Predict labels

Now that we have a method for computing distances between training and test examples, we need to implement a function that uses those distances together with the training labels to predict labels for test samples.

Complete the implementation of the function below:

```
def predict_labels(dists, y_train, k=1):
  """
  Given distances between all pairs of training and test samples, predict a
  label for each test sample by taking a majority vote among its k nearest
  neighbors in the training set.

  In the event of a tie, this function should return the smaller label. For
  example, if k=5 and the 5 nearest neighbors to a test example have labels
  [1, 2, 1, 2, 3] then there is a tie between 1 and 2 (each have 2 votes), so
  we should return 1 since it is the smaller label.

  Inputs:
  - dists: Torch tensor of shape (num_train, num_test) where dists[i, j] is the
    squared Euclidean distance between the ith training point and the jth test
    point.
  - y_train: Torch tensor of shape (y_train,) giving labels for all training
    samples. Each label is an integer in the range [0, num_classes - 1]
  - k: The number of nearest neighbors to use for classification.

  Returns:
  - y_pred: A torch int64 tensor of shape (num_test,) giving predicted labels
    for the test data, where y_pred[j] is the predicted label for the jth test
    example. Each label should be an integer in the range [0, num_classes - 1].
  """
  num_train, num_test = dists.shape
  y_pred = torch.zeros(num_test, dtype=torch.int64)
  ##############################################################################
  # TODO: Implement this function. You may use an explicit loop over the test  #
```

```
# TODO: Implement this function. You may use an explicit loop over the test #
# samples. Hint: Look up the function torch.topk                            #
##############################################################################
# Replace "pass" statement with your code
for j in range(num_test):
  y_pred[j] = y_train[
      torch.topk(dists[:,j], k=k, largest=False, sorted=True).indices
    ].bincount().argmax(dim=0)

##############################################################################
#                             END OF YOUR CODE                               #
##############################################################################
return y_pred
```

Now we have implemented all the required functionality for the K-Nearest Neighbor classifier. We can define a simple object to encapsulate the classifier:

```
class KnnClassifier:
  def __init__(self, x_train, y_train):
    """
    Create a new K-Nearest Neighbor classifier with the specified training data.
    In the initializer we simply memorize the provided training data.

    Inputs:
    - x_train: Torch tensor of shape (num_train, C, H, W) giving training data
    - y_train: int64 torch tensor of shape (num_train,) giving training labels
    """
    self.x_train = x_train.contiguous()
    self.y_train = y_train.contiguous()

  def predict(self, x_test, k=1):
    """
    Make predictions using the classifier.

    Inputs:
    - x_test: Torch tensor of shape (num_test, C, H, W) giving test samples
    - k: The number of neighbors to use for predictions

    Returns:
    - y_test_pred: Torch tensor of shape (num_test,) giving predicted labels
      for the test samples.
    """
    dists = compute_distances_no_loops(self.x_train, x_test.contiguous())
    y_test_pred = predict_labels(dists, self.y_train, k=k)
    return y_test_pred

```

```python
  def check_accuracy(self, x_test, y_test, k=1, quiet=False):
    """
    Utility method for checking the accuracy of this classifier on test data.
    Returns the accuracy of the classifier on the test data, and also prints a
    message giving the accuracy.

    Inputs:
    - x_test: Torch tensor of shape (num_test, C, H, W) giving test samples
    - y_test: int64 torch tensor of shape (num_test,) giving test labels
    - k: The number of neighbors to use for prediction
    - quiet: If True, don't print a message.

    Returns:
    - accuracy: Accuracy of this classifier on the test data, as a percent.
      Python float in the range [0, 100]
    """
    y_test_pred = self.predict(x_test, k=k)
    num_samples = x_test.shape[0]
    num_correct = (y_test == y_test_pred).sum().item()
    accuracy = 100.0 * num_correct / num_samples
    msg = (f'Got {num_correct} / {num_samples} correct; '
           f'accuracy is {accuracy:.2f}%')
    if not quiet:
      print(msg)
    return accuracy
```

Now lets put everything together and test our K-NN clasifier on a subset of CIFAR-10, using k=1:

If you've implemented everything correctly you should see an accuracy of about 27%.

```python
num_train = 5000
num_test = 500
x_train, y_train, x_test, y_test = coutils.data.cifar10(num_train, num_test)
classifier = KnnClassifier(x_train, y_train)
classifier.check_accuracy(x_test, y_test, k=1)
```

```
Got 137 / 500 correct; accuracy is 27.40%
27.4
```

Now lets increase to k=5. You should see a slightly higher accuracy than k=1:

```python
classifier.check_accuracy(x_test, y_test, k=5)
```

```
Got 139 / 500 correct; accuracy is 27.80%
```

27.8

Cross-validation

We have not implemented the full k-Nearest Neighbor classifier, but the choice of $k = 5$ was arbitrary. We will use **cross-validation** to set this hyperparameter in a more principled manner.

Implement the following function to run cross-validation:

```
def knn_cross_validate(x_train, y_train, num_folds=5, k_choices=None):
  """
  Perform cross-validation for KnnClassifier.

  Inputs:
  - x_train: Tensor of shape (num_train, C, H, W) giving all training data
  - y_train: int64 tensor of shape (num_train,) giving labels for training data
  - num_folds: Integer giving the number of folds to use
  - k_choices: List of integers giving the values of k to try

  Returns:
  - k_to_accuracies: Dictionary mapping values of k to lists, where
    k_to_accuracies[k][i] is the accuracy on the ith fold of a KnnClassifier
    that uses k nearest neighbors.
  """
  if k_choices is None:
    # Use default values
    k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100]

  # First we divide the training data into num_folds equally-sized folds.
  x_train_folds = []
  y_train_folds = []
  ##############################################################################
  # TODO: Split the training data and images into folds. After splitting,      #
  # x_train_folds and y_train_folds should be lists of length num_folds, where #
  # y_train_folds[i] is the label vector for images in x_train_folds[i].       #
  # Hint: torch.chunk                                                          #
  ##############################################################################
  # Replace "pass" statement with your code
  x_train_folds = x_train.chunk(num_folds)
  y_train_folds = y_train.chunk(num_folds)
  ##############################################################################
  #                             END OF YOUR CODE                               #
  ##############################################################################

  # 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
# accuracies we found when trying KnnClassifiers that use k neighbors.
k_to_accuracies = {}

############################################################################
# TODO: Perform cross-validation to find the best value of k. For each value #
# of k in k_choices, run the k-nearest-neighbor algorithm num_folds times;   #
# in each case you'll use all but one fold as training data, and use the     #
# last fold as a validation set. Store the accuracies for all folds and all  #
# values in k in k_to_accuracies.   HINT: torch.cat                          #
############################################################################
# Replace "pass" statement with your code
for k in k_choices:
  for fi in range(num_folds):
    f_x_train = torch.cat(x_train_folds[:fi] + x_train_folds[fi+1:], dim=0)
    f_y_train = torch.cat(y_train_folds[:fi] + y_train_folds[fi+1:], dim=0)
    classifier = KnnClassifier(f_x_train, f_y_train)
    acc = classifier.check_accuracy(x_train_folds[fi], y_train_folds[fi], k=k, quiet=True)
    if k not in k_to_accuracies:
      k_to_accuracies[k] = []
    k_to_accuracies[k] += [acc]
############################################################################
#                             END OF YOUR CODE                             #
############################################################################

return k_to_accuracies
```

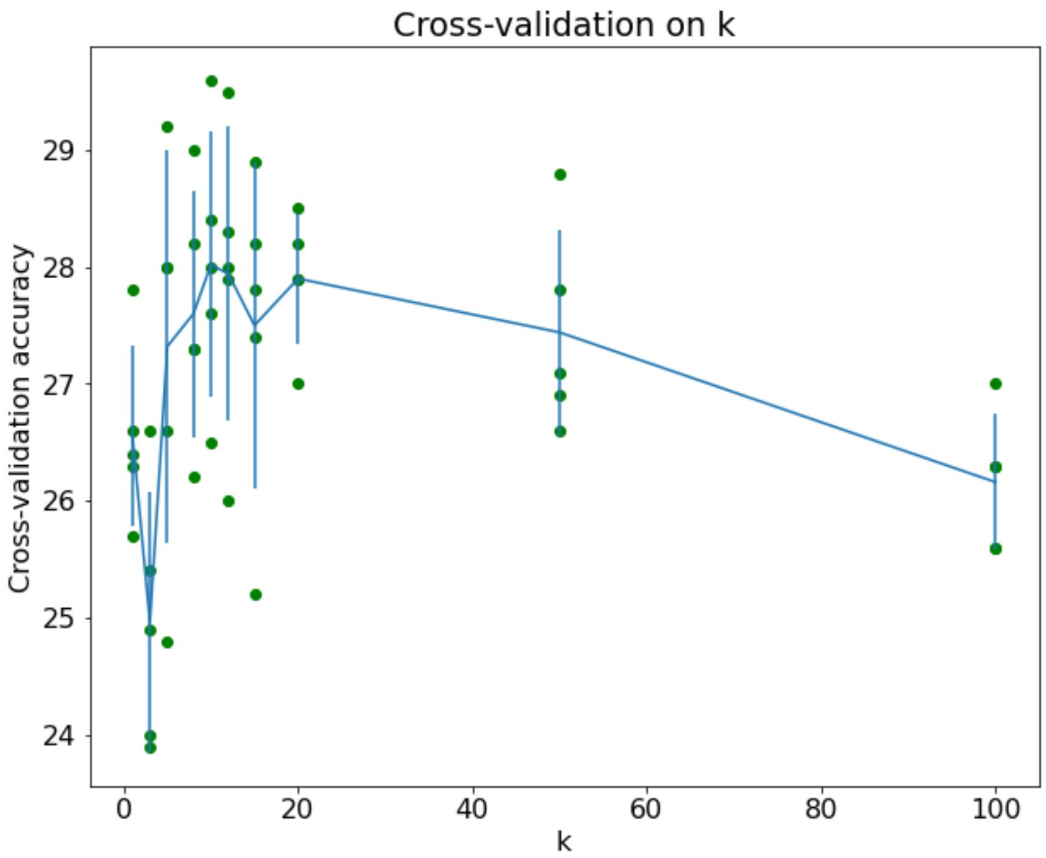
Now we'll run your cross-validation function:

```
num_train = 5000
num_test = 500
x_train, y_train, x_test, y_test = coutils.data.cifar10(num_train, num_test)

k_to_accuracies = knn_cross_validate(x_train, y_train, num_folds=5)

for k, accs in sorted(k_to_accuracies.items()):
  print('k = %d got accuracies: %r' % (k, accs))
```

```
k = 1 got accuracies: [26.3, 25.7, 26.4, 27.8, 26.6]
k = 3 got accuracies: [23.9, 24.9, 24.0, 26.6, 25.4]
k = 5 got accuracies: [24.8, 26.6, 28.0, 29.2, 28.0]
k = 8 got accuracies: [26.2, 28.2, 27.3, 29.0, 27.3]
k = 10 got accuracies: [26.5, 29.6, 27.6, 28.4, 28.0]
k = 12 got accuracies: [26.0, 29.5, 27.9, 28.3, 28.0]
k = 15 got accuracies: [25.2, 28.9, 27.8, 28.2, 27.4]
```

```
k = 20 got accuracies: [27.0, 27.9, 27.9, 28.2, 28.5]
k = 50 got accuracies: [27.1, 28.8, 27.8, 26.9, 26.6]
k = 100 got accuracies: [25.6, 27.0, 26.3, 25.6, 26.3]
```

```
ks, means, stds = [], [], []
for k, accs in sorted(k_to_accuracies.items()):
  plt.scatter([k] * len(accs), accs, color='g')
  ks.append(k)
  means.append(statistics.mean(accs))
  stds.append(statistics.stdev(accs))
plt.errorbar(ks, means, yerr=stds)
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.title('Cross-validation on k')
plt.show()
```



Now we can use the results of cross-validation to select the best value for k, and rerun the classifier on our full 5000 set of training examples.

You should get an accuracy above 28%.

```
best_k = 1
################################################################################
# TODO: Use the results of cross-validation stored in k_to_accuracies to     #
# choose the value of k, and store the result in best_k. You should choose   #
# the value of k that has the highest mean accuracy accross all folds.       #
################################################################################
# Replace "pass" statement with your code
best_k = max((statistics.mean(k_to_accuracies[k]), k) for k in k_to_accuracies.keys())[1]
################################################################################
#                             END OF YOUR CODE                               #
################################################################################

print('Best k is ', best_k)
classifier = KnnClassifier(x_train, y_train)
classifier.check_accuracy(x_test, y_test, k=best_k)
```

```
Best k is  10
Got 141 / 500 correct; accuracy is 28.20%
28.2
```

Finally, we can use our chosen value of k to run on the entire training and test sets.

This may take a while to run, since the full training and test sets have 50k and 10k examples respectively. You should get an accuracy above 33%.

Run this only once!

```
x_train_all, y_train_all, x_test_all, y_test_all = coutils.data.cifar10()
classifier = KnnClassifier(x_train_all, y_train_all)
classifier.check_accuracy(x_test_all, y_test_all, k=best_k)
```

```
Got 3386 / 10000 correct; accuracy is 33.86%
33.86
```