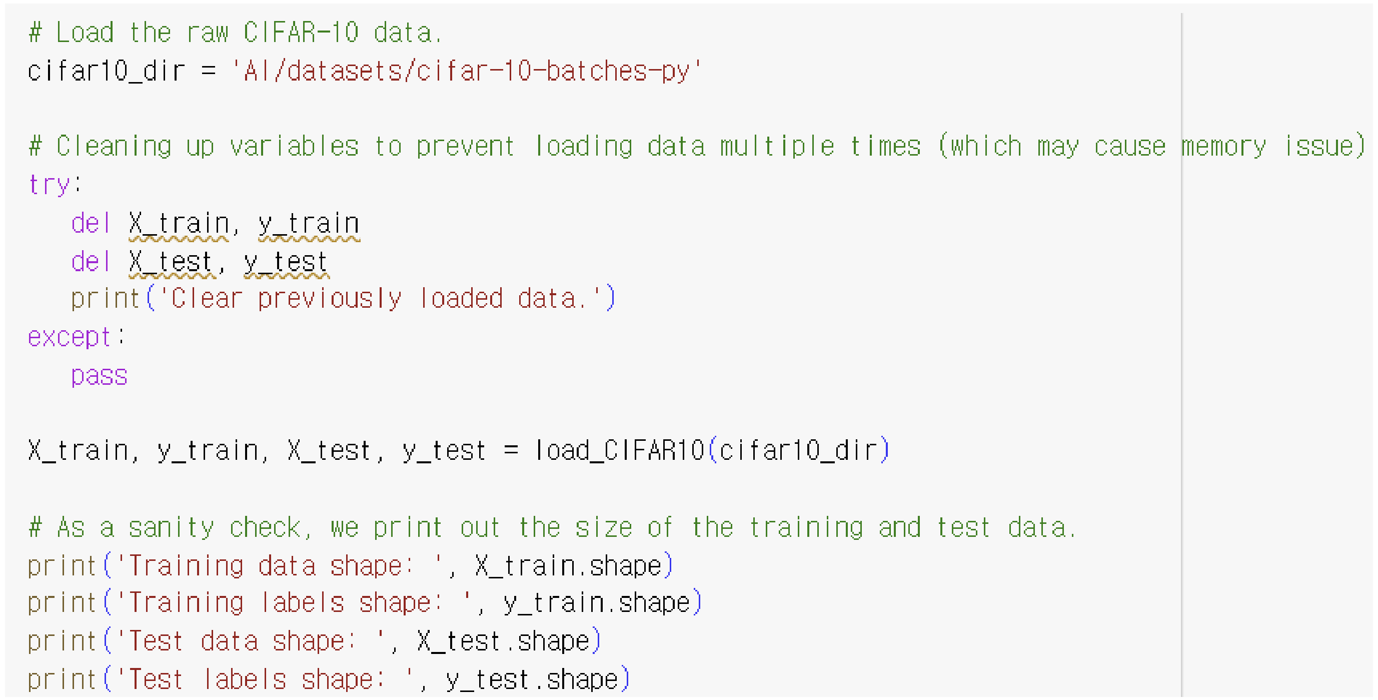
실습2: svm 전반부

설정

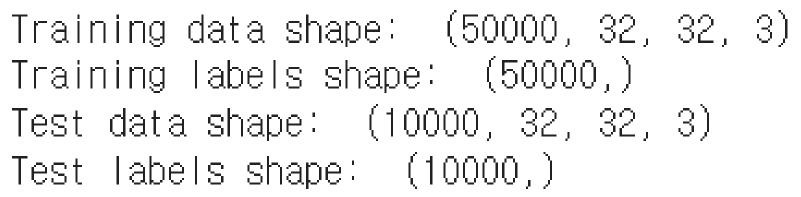
```python
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment1/'
FOLDERNAME = 'AI1my/'
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My₩ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My₩ Drive/$FOLDERNAME
```

```python
# Run some setup code for this notebook.
import random
import numpy as np
from AI.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
```

CIFAR-10 데이터셋 로드



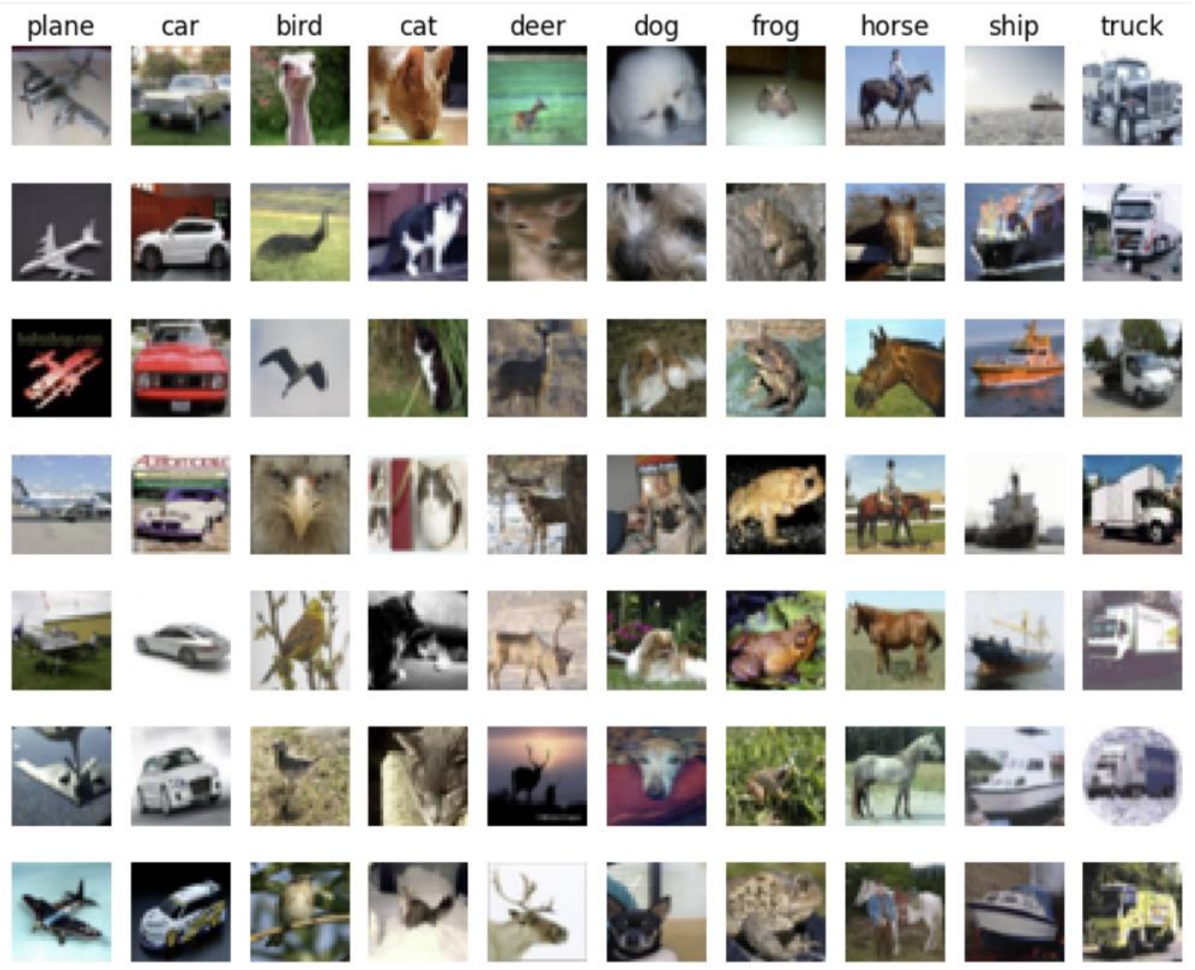
```python
# Load the raw CIFAR-10 data.
cifar10_dir = 'AI/datasets/cifar-10-batches-py'

# 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,)
```

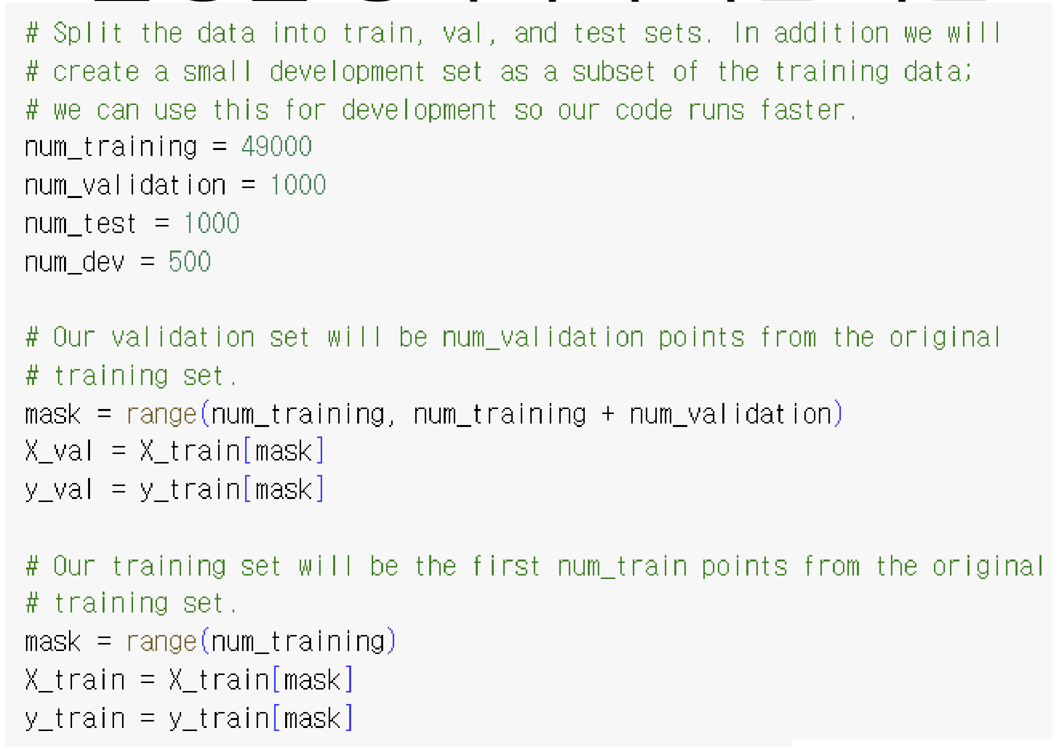
CIFAR-10 데이터셋 샘플 그리기

```
# 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()
```

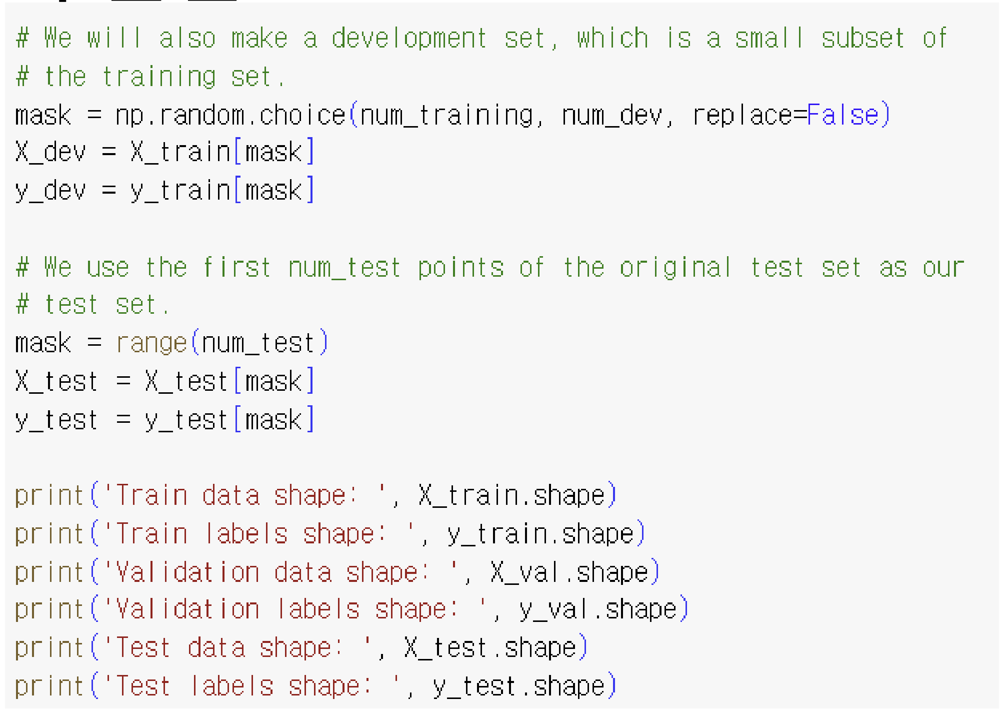


훈련세트/검증세트/테스트세트 생성

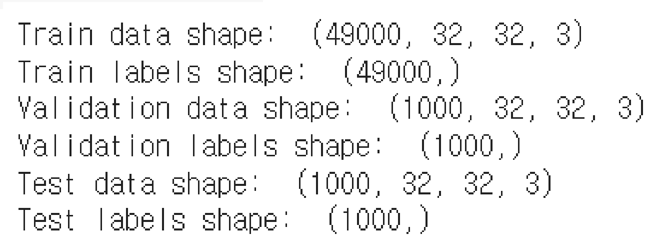
- 개발 세트(development set): 학습 중에 빠르게 코드의 정확성 및 효율성을 평가하기 위한 작은 데이터 집합



```
# Split the data into train, val, and test sets. In addition we will
# create a small development set as a subset of the training data;
# we can use this for development so our code runs faster.
num_training = 49000
num_validation = 1000
num_test = 1000
num_dev = 500

# Our validation set will be num_validation points from the original
# training set.
mask = range(num_training, num_training + num_validation)
X_val = X_train[mask]
y_val = y_train[mask]

# Our training set will be the first num_train points from the original
# training set.
mask = range(num_training)
X_train = X_train[mask]
y_train = y_train[mask]
```



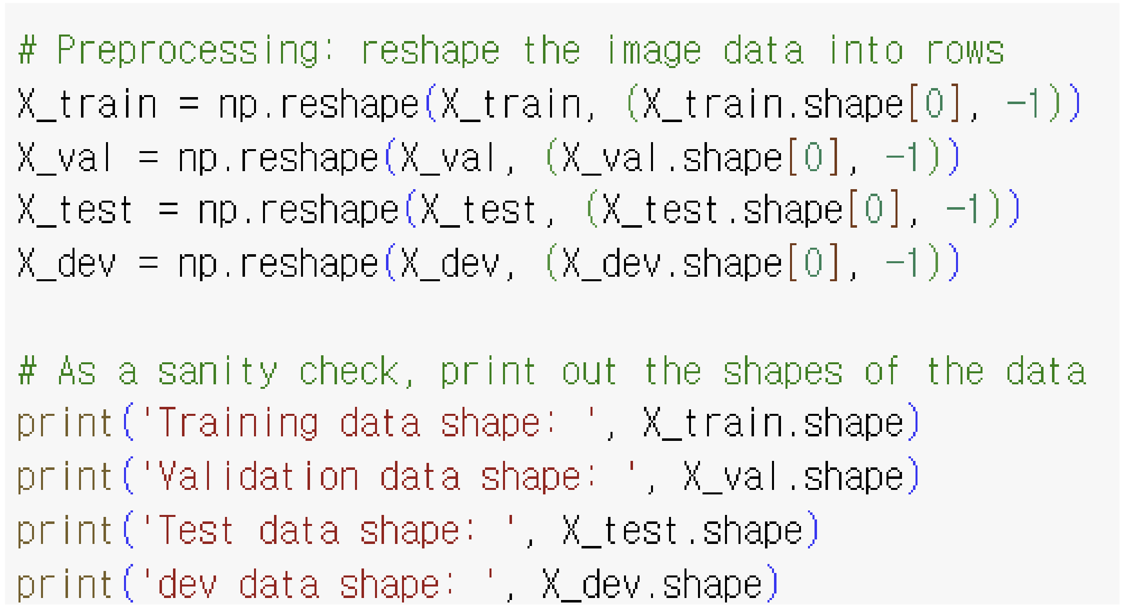
```
# We will also make a development set, which is a small subset of
# the training set.
mask = np.random.choice(num_training, num_dev, replace=False)
X_dev = X_train[mask]
y_dev = y_train[mask]

# We use the first num_test points of the original test set as our
# test set.
mask = range(num_test)
X_test = X_test[mask]
y_test = y_test[mask]

print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```



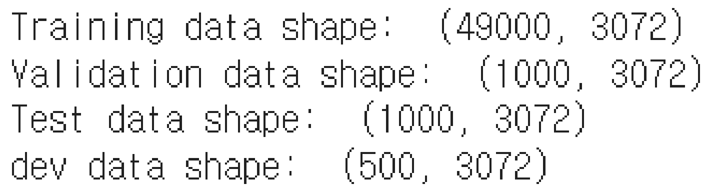
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
```

이미지 픽셀들을 1차원으로 나열

- reshape 함수를 사용하여 4차원 형태 배열을 2차원 형태 배열로 변환

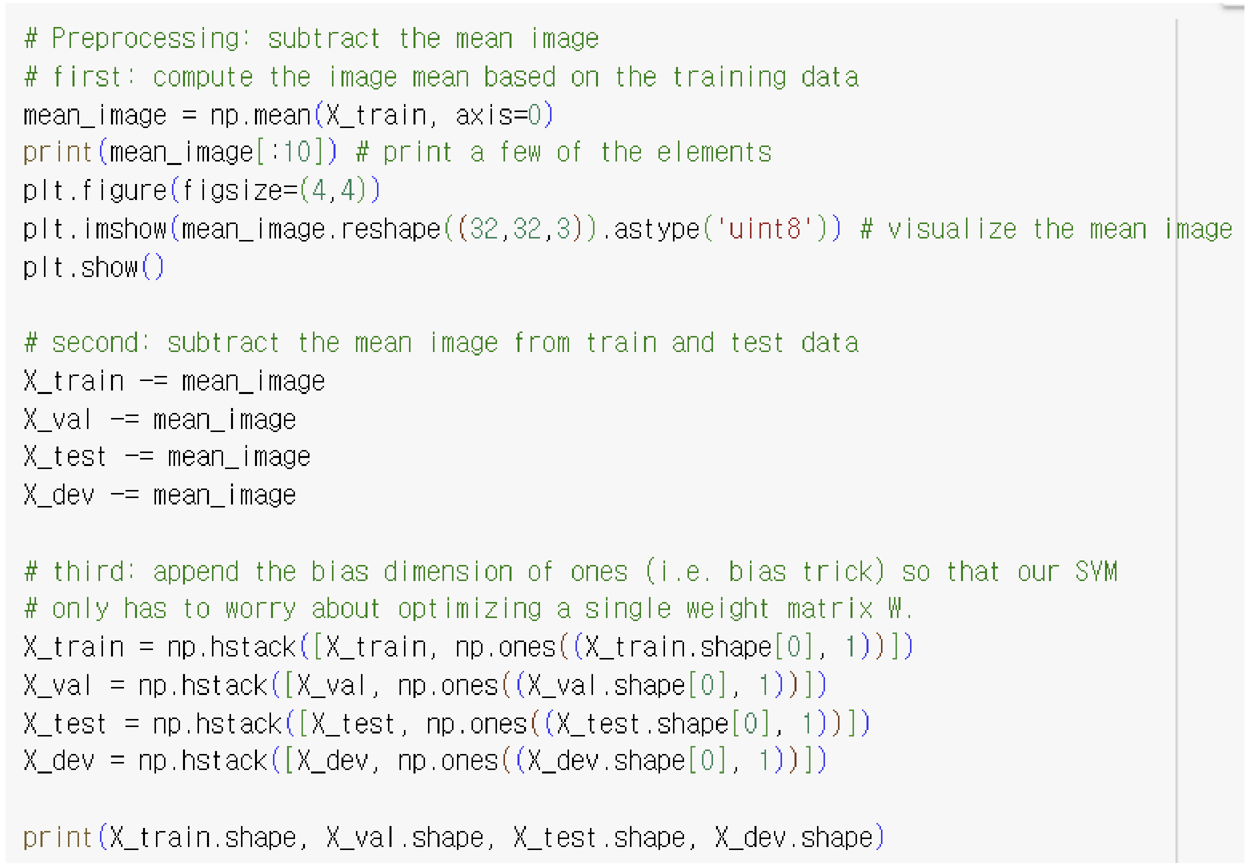


```python
# Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# As a sanity check, print out the shapes of the data
print('Training data shape: ', X_train.shape)
print('Validation data shape: ', X_val.shape)
print('Test data shape: ', X_test.shape)
print('dev data shape: ', X_dev.shape)
```



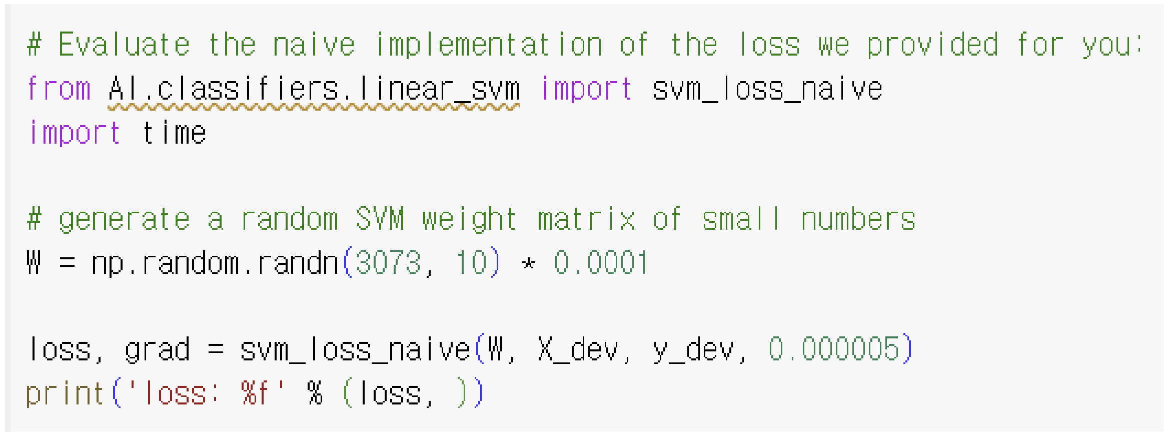
```
Training data shape:  (49000, 3072)
Validation data shape:  (1000, 3072)
Test data shape:  (1000, 3072)
dev data shape:  (500, 3072)
```

전처리

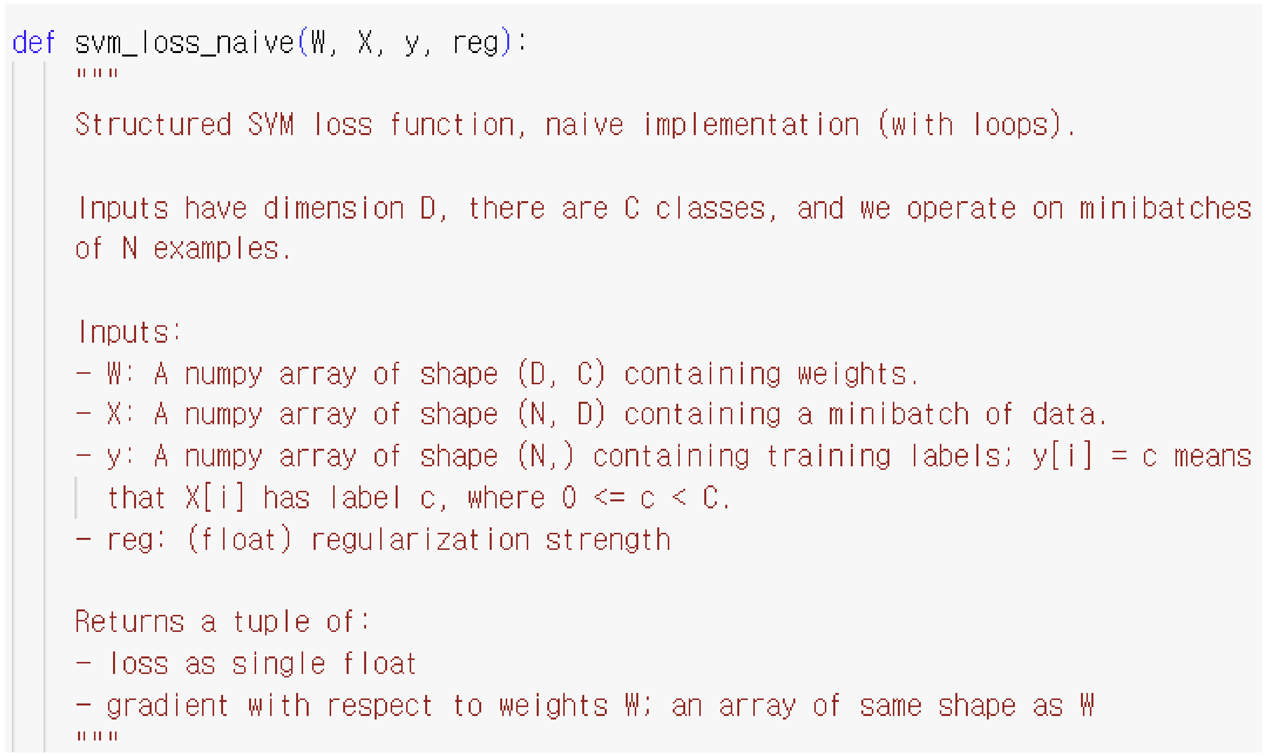


```
# Preprocessing: subtract the mean image
# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean image
plt.show()

# second: subtract the mean image from train and test data
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
X_dev -= mean_image

# third: append the bias dimension of ones (i.e. bias trick) so that our SVM
# only has to worry about optimizing a single weight matrix W.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

print(X_train.shape, X_val.shape, X_test.shape, X_dev.shape)
```

구현1: svm_loss_naive구현

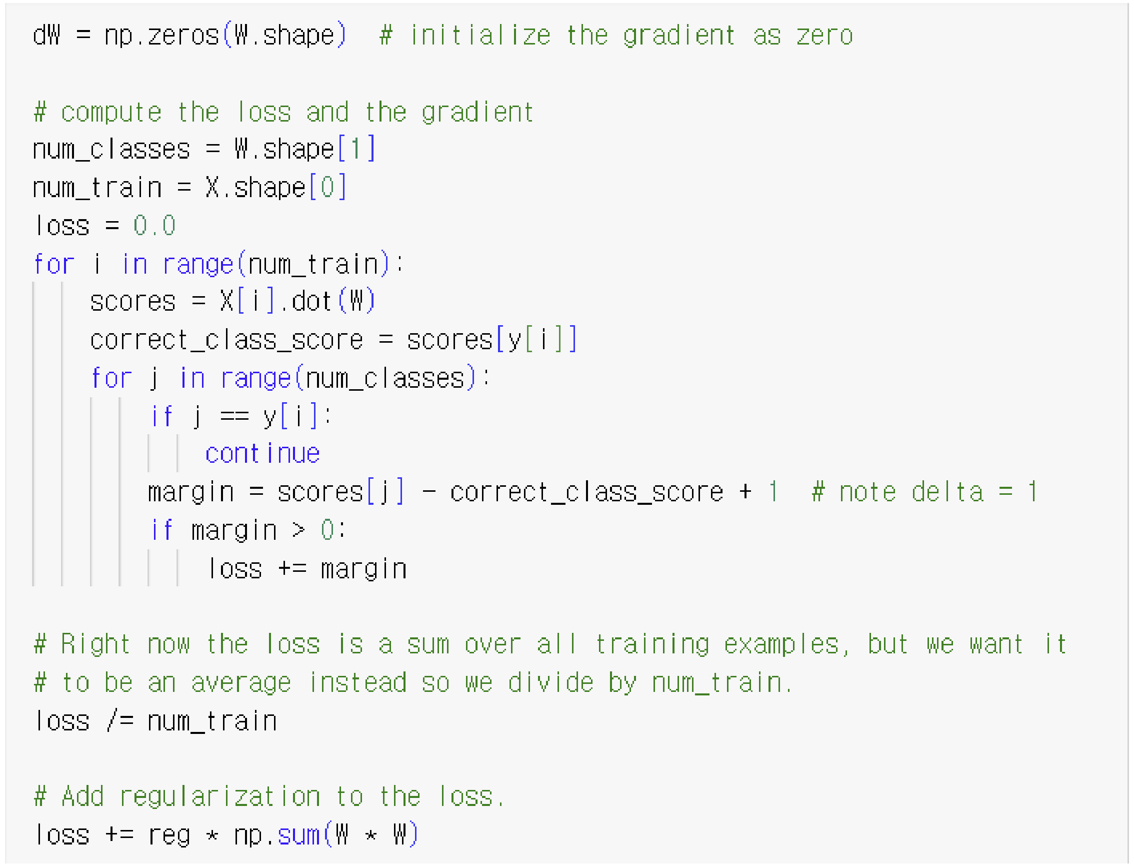


```python
# Evaluate the naive implementation of the loss we provided for you:
from AI.classifiers.linear_svm import svm_loss_naive
import time

# generate a random SVM weight matrix of small numbers
W = np.random.randn(3073, 10) * 0.0001

loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))
```

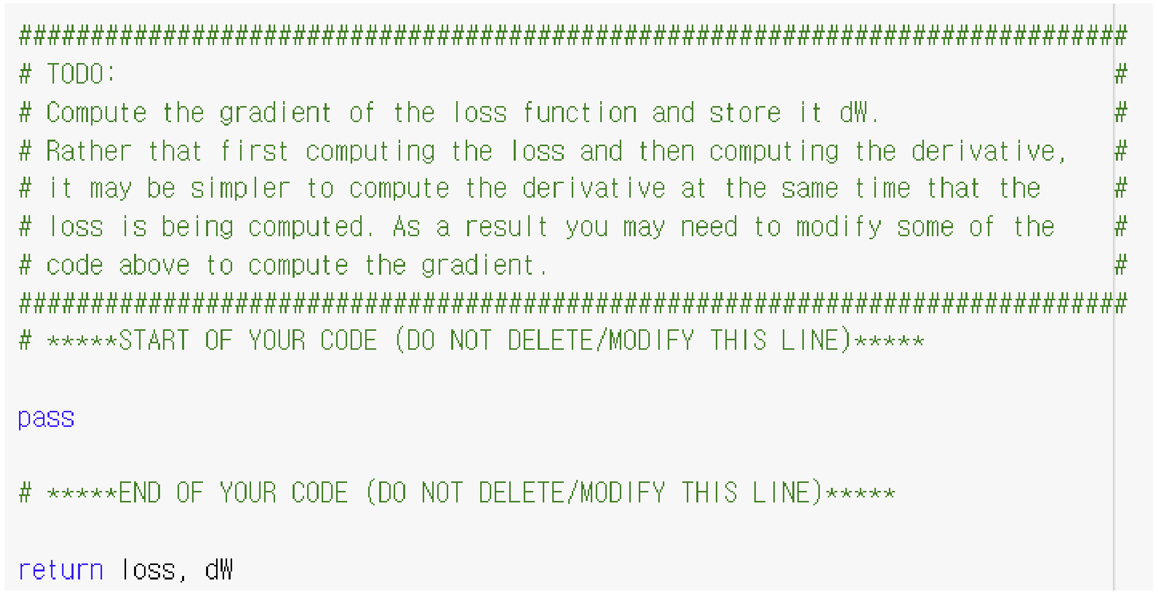


```python
def svm_loss_naive(W, X, y, reg):
    """
    Structured SVM loss function, naive implementation (with loops).

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - W: A numpy array of shape (D, C) containing weights.
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.
    - reg: (float) regularization strength

    Returns a tuple of:
    - loss as single float
    - gradient with respect to weights W; an array of same shape as W
    """
```

구현1: svm_loss_naive구현



```python
dW = np.zeros(W.shape)  # initialize the gradient as zero

# compute the loss and the gradient
num_classes = W.shape[1]
num_train = X.shape[0]
loss = 0.0
for i in range(num_train):
    scores = X[i].dot(W)
    correct_class_score = scores[y[i]]
    for j in range(num_classes):
        if j == y[i]:
            continue
        margin = scores[j] - correct_class_score + 1  # note delta = 1
        if margin > 0:
            loss += margin

# Right now the loss is a sum over all training examples, but we want it
# to be an average instead so we divide by num_train.
loss /= num_train

# Add regularization to the loss.
loss += reg * np.sum(W * W)
```

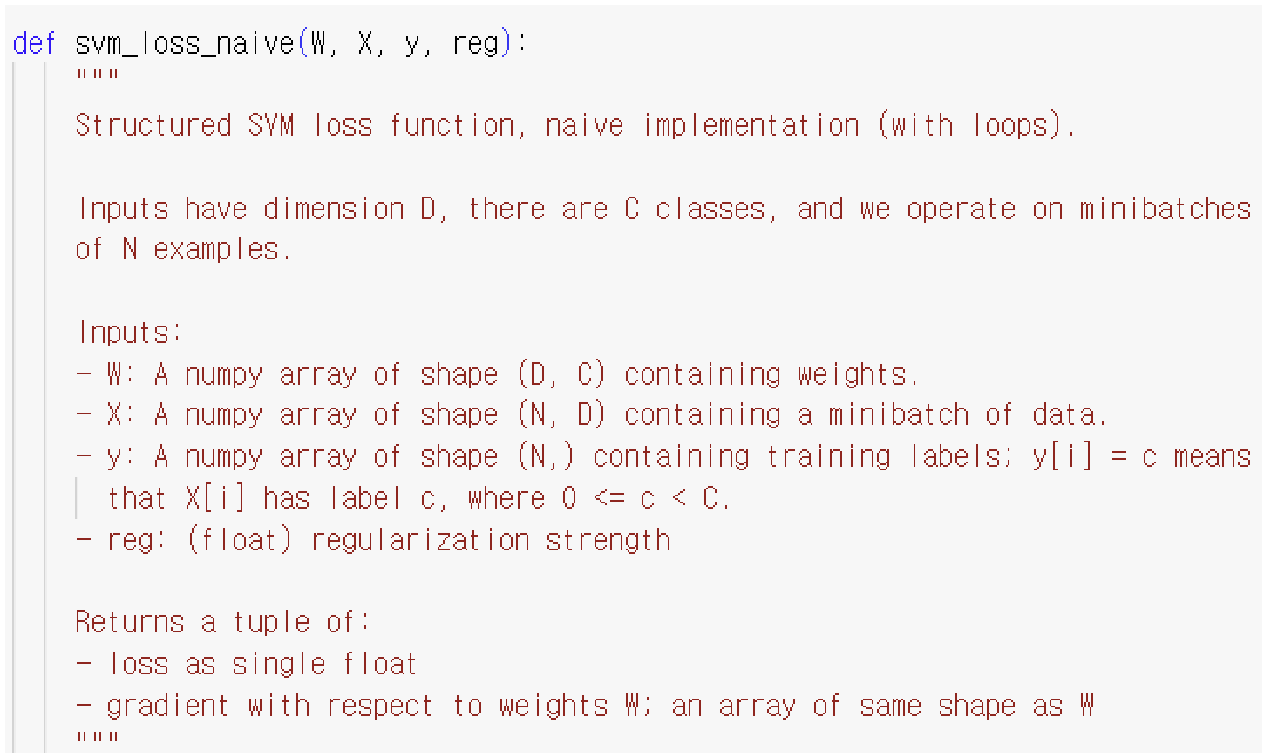


```python
#############################################################################
# TODO:                                                                     #
# Compute the gradient of the loss function and store it dW.                #
# Rather that first computing the loss and then computing the derivative,   #
# it may be simpler to compute the derivative at the same time that the     #
# loss is being computed. As a result you may need to modify some of the    #
# code above to compute the gradient.                                       #
#############################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

pass

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

return loss, dW
```

구현1: svm_loss_naive구현

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$$

$$L = \underbrace{\frac{1}{N} \sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}} \qquad R(W) = \sum_k \sum_l W_{k,l}^2$$

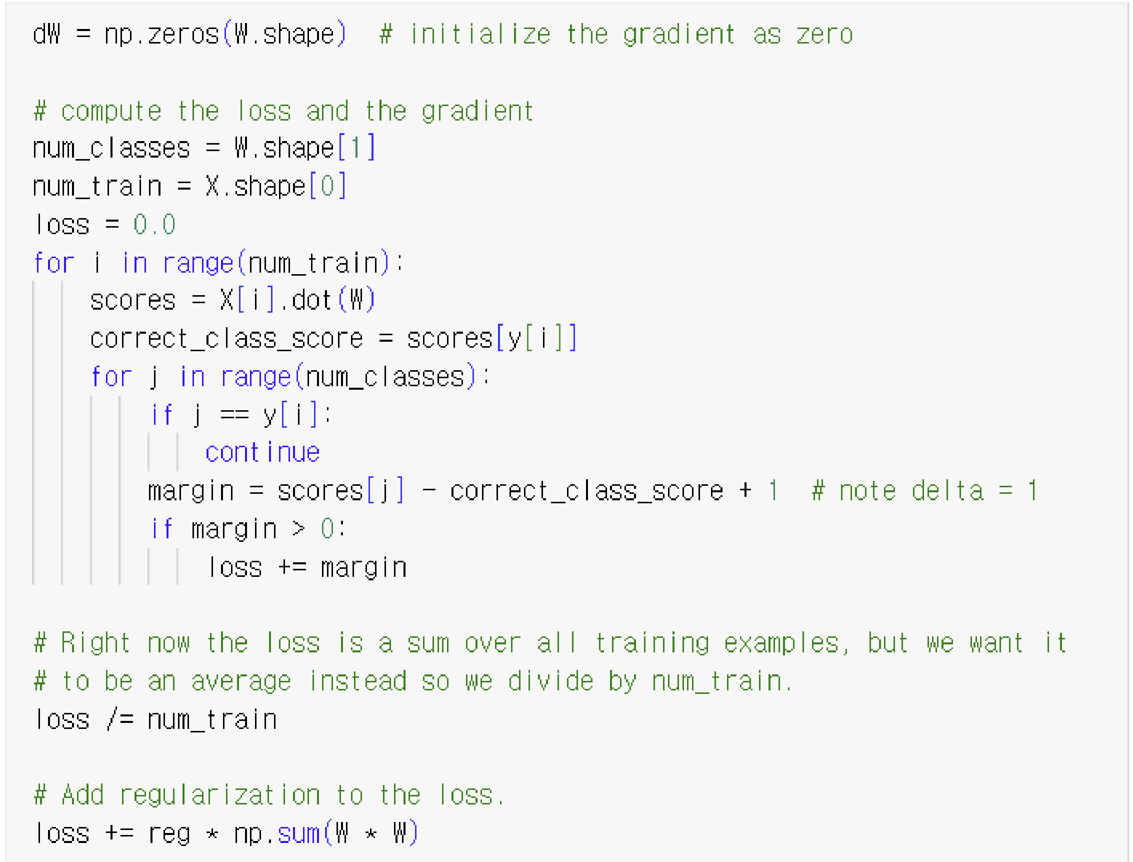


```
def svm_loss_naive(W, X, y, reg):
    """
    Structured SVM loss function, naive implementation (with loops).

    Inputs have dimension D, there are C classes, and we operate on minibatches
    of N examples.

    Inputs:
    - W: A numpy array of shape (D, C) containing weights.
    - X: A numpy array of shape (N, D) containing a minibatch of data.
    - y: A numpy array of shape (N,) containing training labels; y[i] = c means
      that X[i] has label c, where 0 <= c < C.
    - reg: (float) regularization strength

    Returns a tuple of:
    - loss as single float
    - gradient with respect to weights W; an array of same shape as W
    """
```

구현1: svm_loss_naive구현

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta) \qquad L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

$$L = \underbrace{\frac{1}{N} \sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}} \qquad R(W) = \sum_k \sum_l W_{k,l}^2$$



```
dW = np.zeros(W.shape)  # initialize the gradient as zero

# compute the loss and the gradient
num_classes = W.shape[1]
num_train = X.shape[0]
loss = 0.0
for i in range(num_train):
    scores = X[i].dot(W)
    correct_class_score = scores[y[i]]
    for j in range(num_classes):
        if j == y[i]:
            continue
        margin = scores[j] - correct_class_score + 1  # note delta = 1
        if margin > 0:
            loss += margin

# Right now the loss is a sum over all training examples, but we want it
# to be an average instead so we divide by num_train.
loss /= num_train

# Add regularization to the loss.
loss += reg * np.sum(W * W)
```

구현1: svm_loss_naive구현

- $\frac{dL_i}{dW}$ 의 y_i번째 열

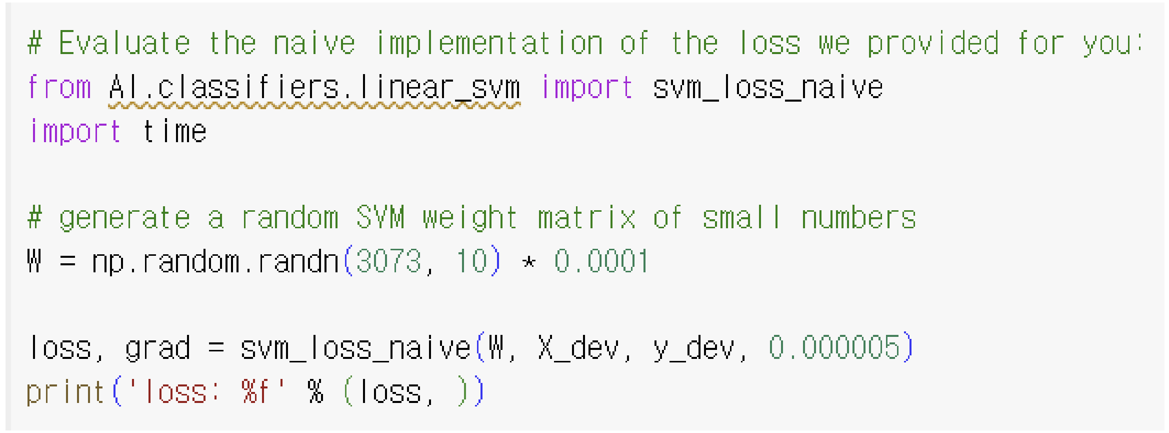
$$\frac{\partial L_i}{\partial w_{y_i}} = -\left(\sum_{\ell \neq y_i} 1(w_\ell^T x_i - w_{y_i}^T x_i + \Delta > 0)\right) x_i$$

- $\frac{dL_i}{dW}$ 의 $j(\neq y_i)$번째 열

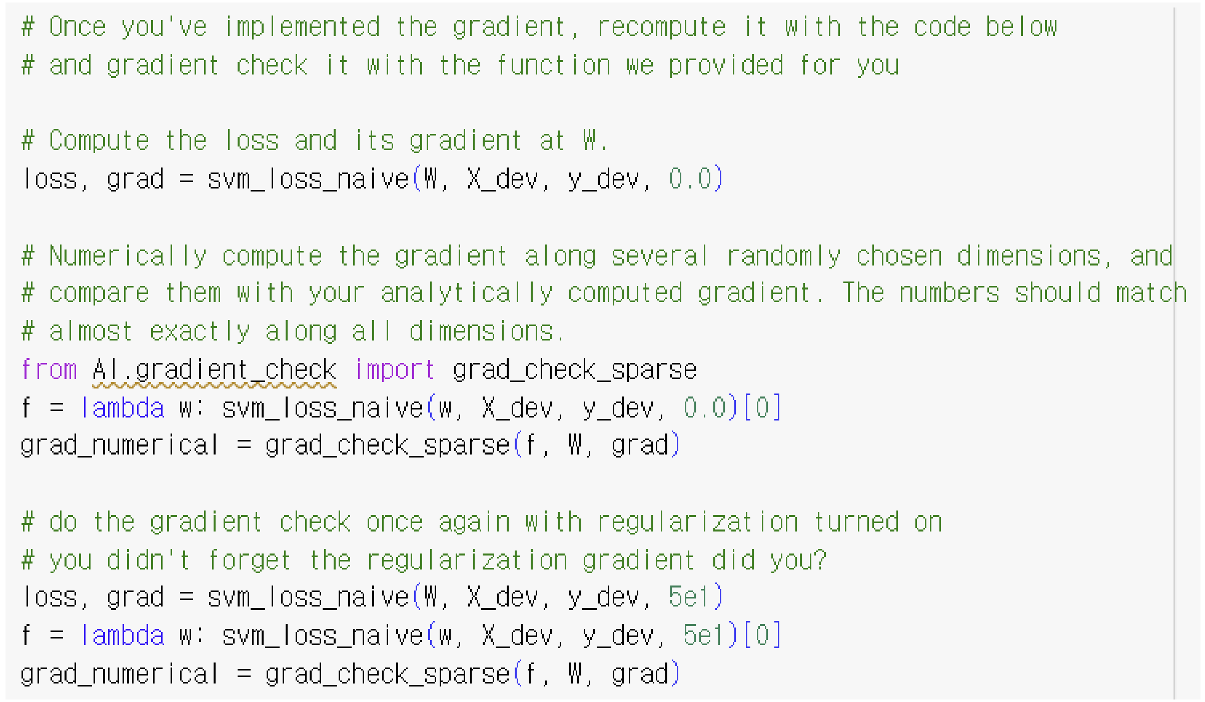
$$\frac{\partial L_i}{\partial w_j} = 1(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

$$\frac{\partial L}{\partial W} = \frac{1}{N} \sum_i \frac{\partial L_i}{\partial W} + 2\lambda W$$

구현1: svm_loss_naive구현



```python
# Evaluate the naive implementation of the loss we provided for you:
from AI.classifiers.linear_svm import svm_loss_naive
import time

# generate a random SVM weight matrix of small numbers
W = np.random.randn(3073, 10) * 0.0001

loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.000005)
print('loss: %f' % (loss, ))
```



```python
# Once you've implemented the gradient, recompute it with the code below
# and gradient check it with the function we provided for you

# Compute the loss and its gradient at W.
loss, grad = svm_loss_naive(W, X_dev, y_dev, 0.0)

# Numerically compute the gradient along several randomly chosen dimensions, and
# compare them with your analytically computed gradient. The numbers should match
# almost exactly along all dimensions.
from AI.gradient_check import grad_check_sparse
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad)

# do the gradient check once again with regularization turned on
# you didn't forget the regularization gradient did you?
loss, grad = svm_loss_naive(W, X_dev, y_dev, 5e1)
f = lambda w: svm_loss_naive(w, X_dev, y_dev, 5e1)[0]
grad_numerical = grad_check_sparse(f, W, grad)
```

구현2: svm_loss_vectorized구현

$$L_i = \sum_{j \neq y_i} \max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta) \qquad L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

$$L = \underbrace{\frac{1}{N} \sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}} \qquad R(W) = \sum_k \sum_l W_{k,l}^2$$

- i번째 이미지에 대한 j번째 클래스 점수를 s_{ij}라고 해서 다시 표현

$$L_i = \sum_{\ell \neq y_i} \max(0, s_{i\ell} - s_{iy_i} + \Delta) \qquad L = \underbrace{\frac{1}{N} \sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

- L에 대한 식을 풀어쓰면 다음과 같이 됨

$$L = \frac{1}{N} \sum_i \sum_{\ell \neq y_i} \max(0, s_{i\ell} - s_{iy_i} + \Delta) + \lambda R(W)$$

구현2: svm_loss_vectorized구현

$$L = \frac{1}{N} \sum_i \sum_{\ell \neq y_i} \max(0, s_{i\ell} - s_{iy_i} + \Delta) + \lambda R(W)$$

- 행렬 M을 정의

$$M_{ij} = \max(0, s_{ij} - s_{iy_i} + 1)$$

- 행렬 M에서 모든 i 각각에 대해 (i, y_i) 요소를 0로 만든 후, 모든 요소들을 더하는 방식을 사용하면 위 이중 sigma 식 계산가능
- 행렬 M을 계산하기 위해 점수 행렬 S를 계산 필요 $(S_{ij} = s_{ij})$

구현2: svm_loss_vectorized구현

$$L = \frac{1}{N}\sum_i \sum_{\ell \neq y_i} \max(0, s_{i\ell} - s_{iy_i} + \Delta) + 0.5\lambda R(W)$$

- 정규화 항에 0.5를 곱한 것을 사용
- 그래디언트 계산 시에는 2가 곱해짐

구현2: svm_loss_vectorized구현

- 먼저, L_i에 대한 그래디언트 $\frac{\partial L_i}{\partial W}$를 계산

- $\frac{\partial L_i}{\partial W}$의 y_i번째 열 ($\frac{\partial L_i}{\partial w_{y_i}}$)

$$\frac{\partial L_i}{\partial w_{y_i}} = -\left(\sum_{\ell \neq y_i} 1(s_{i\ell} - s_{iy_i} + \Delta > 0)\right)x_i$$

- $\frac{\partial L_i}{\partial W}$의 j번째 열($j \neq y_i$) ($\frac{\partial L_i}{\partial w_j}$)

$$\frac{\partial L_i}{\partial w_j} = 1(s_{ij} - s_{iy_i} + \Delta > 0)x_i$$

구현2: svm_loss_vectorized구현

- 다음과 같이 $N \times C$크기의 행렬 C를 정의

$$C_{ij} = 1(s_{ij} - s_{iy_i} + \Delta > 0)$$

- 그래디언트를 행렬 C를 사용하여 표현

$$\frac{\partial L_i}{\partial w_{y_i}} = -\Big(\sum_{\ell \neq y_i} C_{i\ell}\Big) x_i$$

$$\frac{\partial L_i}{\partial w_j} = C_{ij} x_i \qquad j \neq y_i$$

- j번째 열에 대한 식 정리

$$\frac{\partial L_i}{\partial w_j} = -1(y_i = j)\Big(\sum_{\ell \neq y_i} C_{i\ell}\Big) x_i + 1(y_i \neq j) C_{ij} x_i$$

구현2: svm_loss_vectorized구현

- $\frac{\partial L}{\partial w_j}$ 식 정리

$$\frac{\partial L}{\partial w_j} = \frac{1}{N}\sum_i [1(y_i \neq j)C_{ij} - 1(y_i = j)(\sum_{\ell \neq y_i} C_{i\ell})]x_i$$

- 다음 행렬 D를 정의

$$D_{ij} = 1(y_i \neq j)C_{ij} - 1(y_i = j)(\sum_{\ell \neq y_i} C_{i\ell})$$

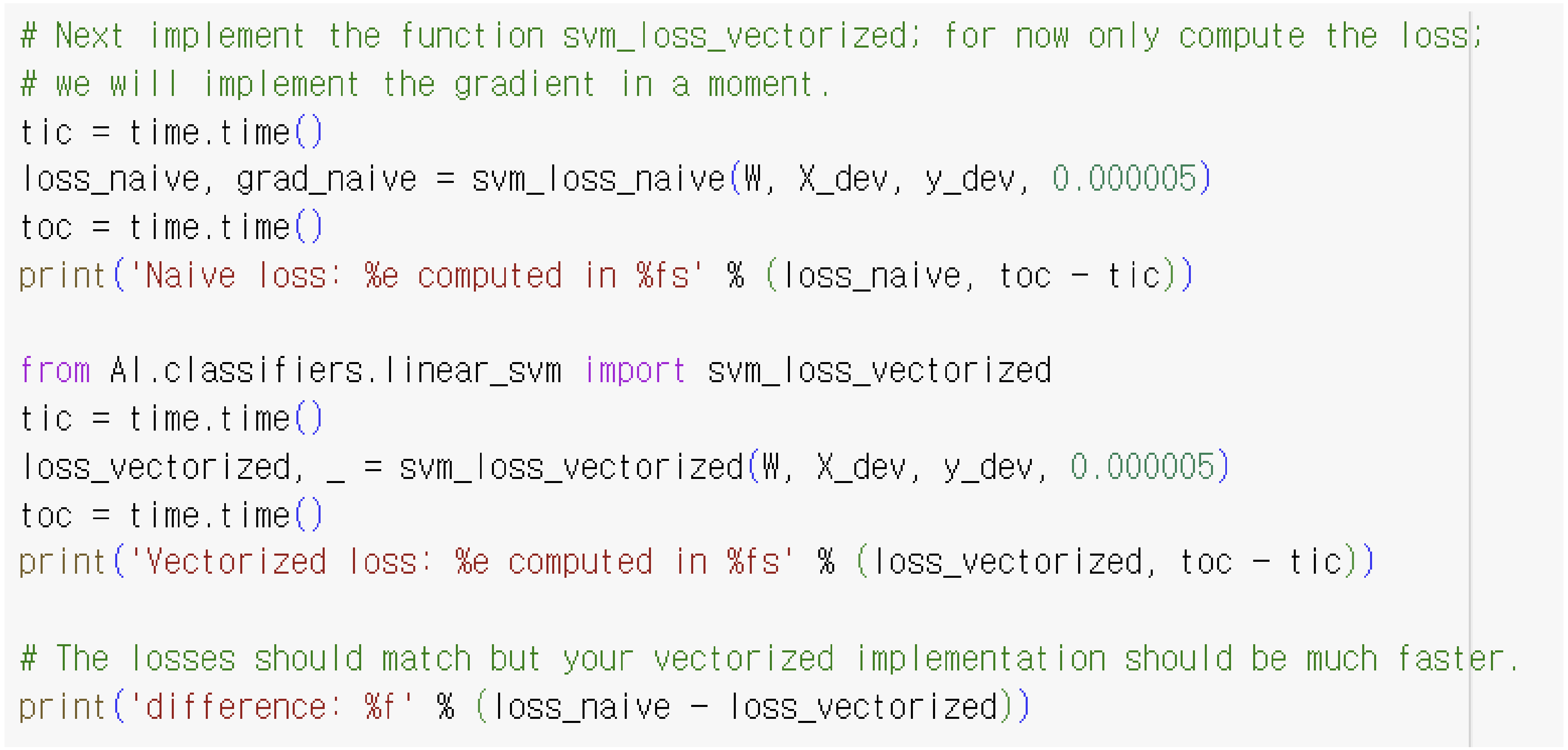
- $\frac{\partial L}{\partial w_i}$ 식을 행렬 D로 표현

$$\frac{\partial L}{\partial w_j} = \frac{1}{N}\sum_i D_{ij}x_i \qquad (\frac{\partial L}{\partial w_j})_\ell = (\frac{\partial L}{\partial W})_{\ell j} = (\frac{1}{N}X^T \cdot D)_{\ell j}$$

구현2: svm_loss_vectorized구현

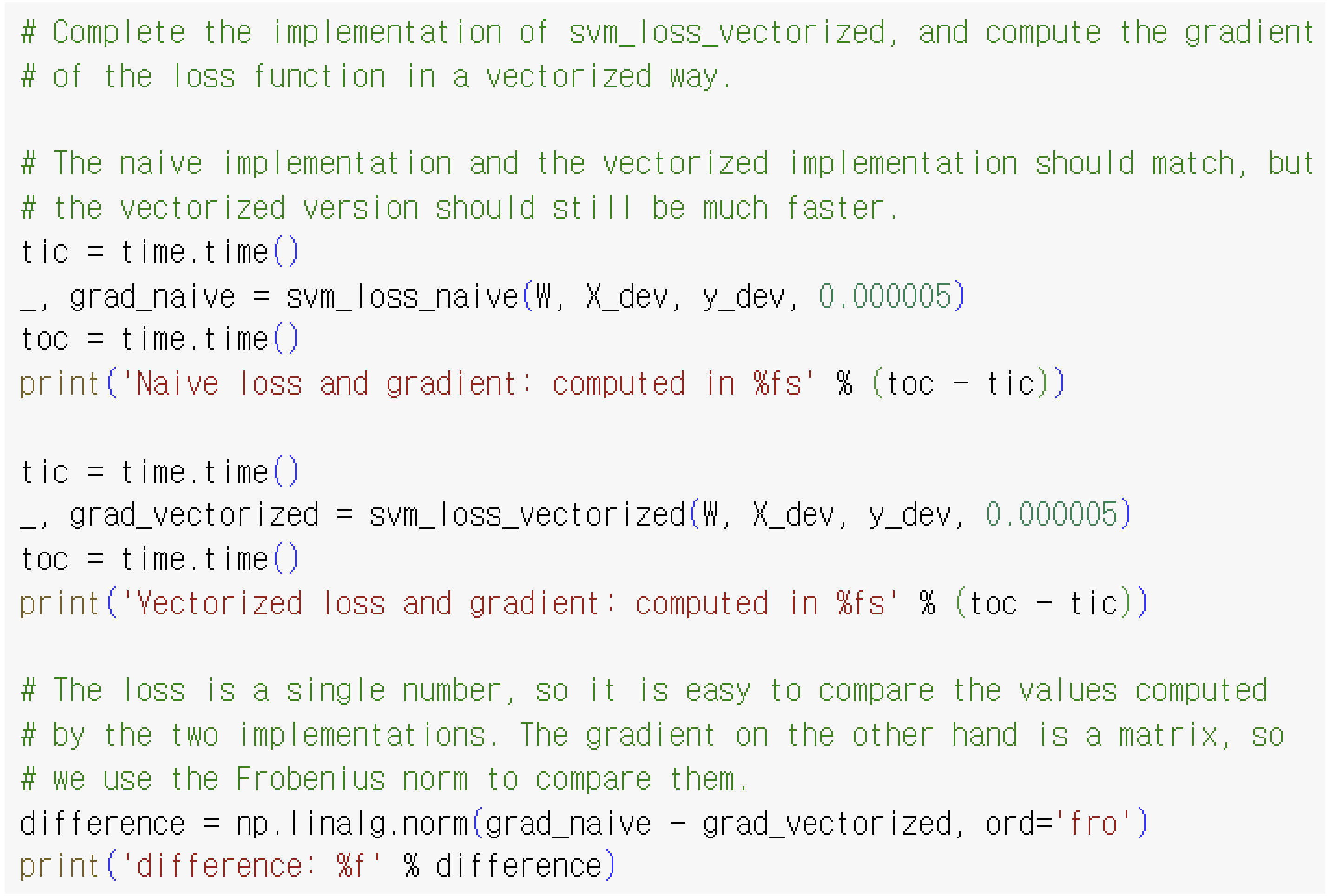
- 행렬 D 계산
  - $D = D^{(1)} + D^{(2)}$
  - $D_{ij}^{(1)} = 1(y_i \neq j)C_{ij}$
  - $D_{ij}^{(2)} = 1(y_i = j)(\sum_{\ell \neq y_i} C_{i\ell})$
- 정규화 항 추가
  - 지금까지 구한 $\frac{\partial L}{\partial W}$ 식에 λW를 더함

svm_loss_naive v.s. svm_loss_vectorized



```python
# Next implement the function svm_loss_vectorized; for now only compute the loss;
# we will implement the gradient in a moment.
tic = time.time()
loss_naive, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss: %e computed in %fs' % (loss_naive, toc - tic))

from AI.classifiers.linear_svm import svm_loss_vectorized
tic = time.time()
loss_vectorized, _ = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss: %e computed in %fs' % (loss_vectorized, toc - tic))

# The losses should match but your vectorized implementation should be much faster.
print('difference: %f' % (loss_naive - loss_vectorized))
```

svm_loss_naive v.s. svm_loss_vectorized



```
# Complete the implementation of svm_loss_vectorized, and compute the gradient
# of the loss function in a vectorized way.

# The naive implementation and the vectorized implementation should match, but
# the vectorized version should still be much faster.
tic = time.time()
_, grad_naive = svm_loss_naive(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Naive loss and gradient: computed in %fs' % (toc - tic))

tic = time.time()
_, grad_vectorized = svm_loss_vectorized(W, X_dev, y_dev, 0.000005)
toc = time.time()
print('Vectorized loss and gradient: computed in %fs' % (toc - tic))

# The loss is a single number, so it is easy to compare the values computed
# by the two implementations. The gradient on the other hand is a matrix, so
# we use the Frobenius norm to compare them.
difference = np.linalg.norm(grad_naive - grad_vectorized, ord='fro')
print('difference: %f' % difference)
```