This is the k-nearest neighbors workbook for ECE 239AS Assignment #2

Please follow the notebook linearly to implement k-nearest neighbors.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with the data, training and evaluating a simple classifier, k-fold cross validation, and as a Python refresher.

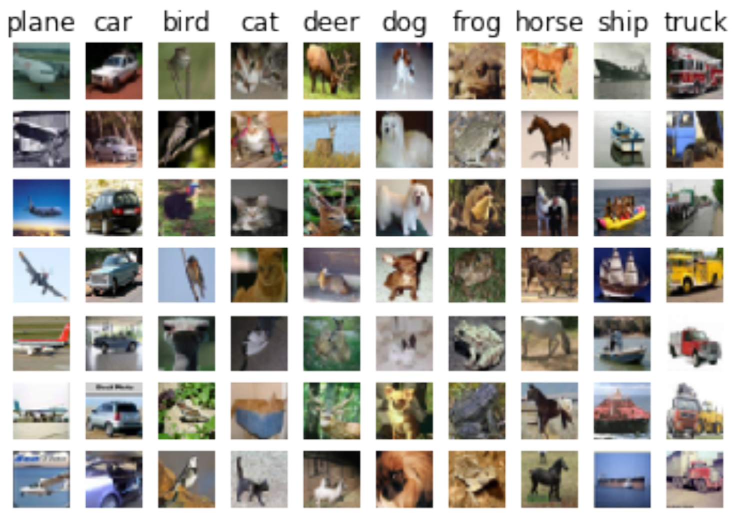
Import the appropriate libraries

```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.data_utils import load_CIFAR10 # function to load the CIFAR-10
dataset.

# Load matplotlib images inline
%matplotlib inline

# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-
ipython
%load_ext autoreload
%autoreload 2
```

```
# Set the path to the CIFAR-10 data
cifar10_dir = 'cifar-10-batches-py'
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,)
```

```
# 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()
```



```
# Subsample the data for more efficient code execution in this exercise
num_training = 5000
mask = list(range(num_training))
X_train = X_train[mask]
y_train = y_train[mask]

num_test = 500
mask = list(range(num_test))
X_test = X_test[mask]
y_test = y_test[mask]

# Reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print(X_train.shape, X_test.shape)
```

```
(5000, 3072) (500, 3072)
```

K-nearest neighbors

In the following cells, you will build a KNN classifier and choose hyperparameters via k-fold cross-validation.

```
# Import the KNN class

from nndl import KNN
```

```
# Declare an instance of the knn class.
knn = KNN()

# Train the classifier.
#   We have implemented the training of the KNN classifier.
#   Look at the train function in the KNN class to see what this does.
knn.train(X=X_train, y=y_train)
```

Questions

(1) Describe what is going on in the function knn.train().

(2) What are the pros and cons of this training step?

Answers

(1) We just memorize the data, i.e. store the data in class variables.

(2) The pros are that training is fast and simple, because it's just a simple variable assignment. The cons are that it requires a large memory overhead, i.e. the bigger the dataset the more memory we use. For datasets that don't fit in memory this won't really work

KNN prediction

In the following sections, you will implement the functions to calculate the distances of test points to training points, and from this information, predict the class of the KNN.

```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition of the norm
#   in the code, which is the 2-norm.
# You should only have to fill out the clearly marked sections.

import time
time_start =time.time()

dists_L2 = knn.compute_distances(X=X_test)

print('Time to run code: {}'.format(time.time()-time_start))
print('Frobenius norm of L2 distances: {}'.format(np.linalg.norm(dists_L2, 'fro')))
```

```
Time to run code: 132.85851097106934
Frobenius norm of L2 distances: 7906696.077040902
```

Really slow code

Note: This probably took a while. This is because we use two for loops. We could increase the speed via vectorization, removing the for loops.

If you implemented this correctly, evaluating np.linalg.norm(dists_L2, 'fro') should return: ~7906696

KNN vectorization

The above code took far too long to run. If we wanted to optimize hyperparameters, it would be time-expensive. Thus, we will speed up the code by vectorizing it, removing the for loops.

```
# Implement the function compute_L2_distances_vectorized() in the KNN class.
# In this function, you ought to achieve the same L2 distance but WITHOUT any
for loops.
# Note, this is SPECIFIC for the L2 norm.

time_start =time.time()
dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=X_test)
print('Time to run code: {}'.format(time.time()-time_start))
print('Difference in L2 distances between your KNN implementations (should be
0): {}'.format(np.linalg.norm(dists_L2 - dists_L2_vectorized, 'fro')))
```

```
Time to run code: 1.4170951843261719
Difference in L2 distances between your KNN implementations (should be 0): 0.0
```

Speedup

Depending on your computer speed, you should see a 10-100x speed up from vectorization. On our computer, the vectorized form took 0.36 seconds while the naive implementation took 38.3 seconds.

Implementing the prediction

Now that we have functions to calculate the distances from a test point to given training points, we now implement the function that will predict the test point labels.

```
# Implement the function predict_labels in the KNN class.
# Calculate the training error (num_incorrect / total_samples)
#   from running knn.predict_labels with k=1

error = 1

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the error rate by calling predict_labels on the test
#   data with k = 1.  Store the error rate in the variable error.
# ================================================================ #
def get_error(distances, y,k):
    labels = knn.predict_labels(distances,k=k)
    diffs = [1 if predicted != actual else 0 for predicted, actual in
zip(labels, y)]
    return sum(diffs)/len(diffs)

error = get_error(dists_L2_vectorized, y_test,k=1)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print(error)
```

```
0.726
```

If you implemented this correctly, the error should be: 0.726.

This means that the k-nearest neighbors classifier is right 27.4% of the time, which is not great, considering that chance levels are 10%.

Optimizing KNN hyperparameters

In this section, we'll take the KNN classifier that you have constructed and perform cross-validation to choose a best value of k, as well as a best choice of norm.

Create training and validation folds

First, we will create the training and validation folds for use in k-fold cross validation.

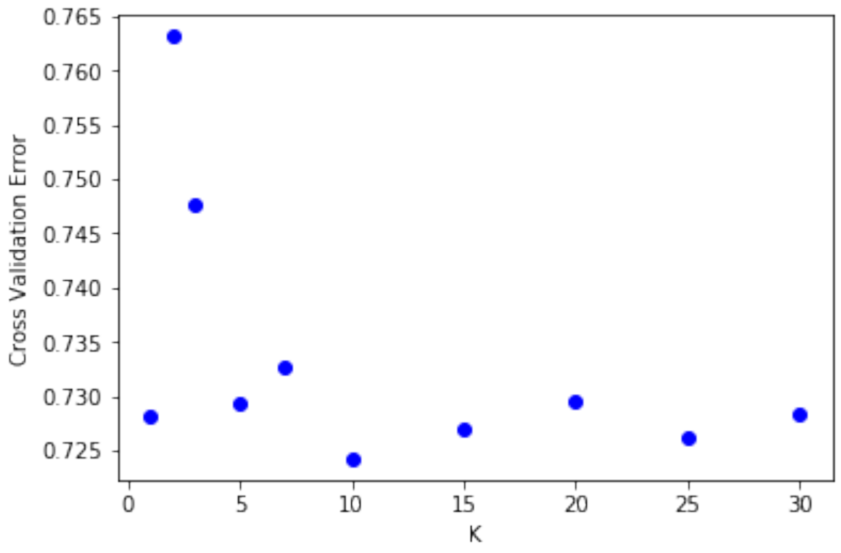
```
# Create the dataset folds for cross-valdiation.
num_folds = 5

X_train_folds = []
y_train_folds =  []

# ================================================================ #
# YOUR CODE HERE:
#   Split the training data into num_folds (i.e., 5) folds.
#   X_train_folds is a list, where X_train_folds[i] contains the
#      data points in fold i.
#   y_train_folds is also a list, where y_train_folds[i] contains
#      the corresponding labels for the data in X_train_folds[i]
# ================================================================ #
combined = np.zeros((X_train.shape[0], X_train.shape[1] + 1))
combined[:, :-1] = X_train
combined[:, -1] = y_train
np.random.shuffle(combined)
examples_per_fold = X_train.shape[0]//num_folds
start = 0
for fold in range(num_folds):
    X_fold, y_fold = combined[start:start + examples_per_fold, :-1],
combined[start:start + examples_per_fold, -1]
    X_train_folds.append(X_fold)
    y_train_folds.append(y_fold)
    start+=examples_per_fold

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

Optimizing the number of nearest neighbors hyperparameter.

In this section, we select different numbers of nearest neighbors and classes which one has the lowest k-fold cross validation error.

```
time_start =time.time()

ks = [1, 2, 3, 5, 7, 10, 15, 20, 25, 30]

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each k in ks, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of k vs. cross-validation error. Since
#   we are assuming L2 distance here, please use the vectorized code!
#   Otherwise, you might be waiting a long time.
# ================================================================ #

k_to_errs = {}
for k in ks:
    overall_error = 0
    for i in range(num_folds):
        # leave out i for testing, and train on all of the others
        training_X = np.vstack([fold for idx, fold in enumerate(X_train_folds)
if idx != i])
        training_Y = np.hstack([fold for idx, fold in enumerate(y_train_folds)
if idx !=i])
        knn.train(training_X, training_Y)
        cv_X, cv_Y = np.array(X_train_folds[i]), np.array(y_train_folds[i])
        cv_distances = knn.compute_L2_distances_vectorized(cv_X)
        cv_error = get_error(cv_distances, cv_Y, k = k)
        overall_error+=cv_error
    avg_err = overall_error/num_folds
    print("got average error {} for k = {}".format(avg_err, k))
    k_to_errs[k] = avg_err

min_key = min(k_to_errs, key = k_to_errs.get)
print('lowest CV error was {} with k = {}'.format(min_key,
k_to_errs[min_key]))
plt.plot(list(k_to_errs.keys()), list(k_to_errs.values()), 'bo')
plt.xlabel('K')
plt.ylabel('Cross Validation Error')
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print('Computation time: %.2f'%(time.time()-time_start))
```

```
got average error 0.7282 for k = 1
got average error 0.7632 for k = 2
got average error 0.7476 for k = 3
got average error 0.7294 for k = 5
got average error 0.7325999999999999 for k = 7
got average error 0.7242 for k = 10
got average error 0.727 for k = 15
got average error 0.7296000000000001 for k = 20
got average error 0.7262 for k = 25
got average error 0.7284 for k = 30
lowest CV error was 10 with k = 0.7242
Computation time: 73.82
```



Questions:

(1) What value of k is best amongst the tested k's?

(2) What is the cross-validation error for this value of k?

Answers:

(1) $k = 10$ was the best k by cross validation error.

(2) The error was 0.7242.

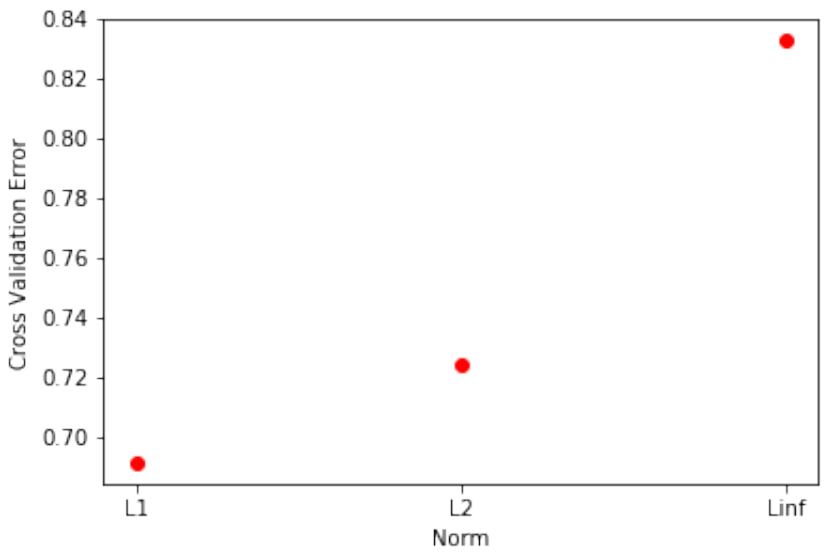
Optimizing the norm

Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

```
time_start =time.time()

L1_norm = lambda x: np.linalg.norm(x, ord=1)
L2_norm = lambda x: np.linalg.norm(x, ord=2)
Linf_norm = lambda x: np.linalg.norm(x, ord= np.inf)
norms = [L1_norm, L2_norm, Linf_norm]
norm_map = {L1_norm: 'l1', L2_norm: 'l2', Linf_norm: 'linf'}
norm_vals = []
# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each norm in norms, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of the norm used vs the cross-validation error
#   Use the best cross-validation k from the previous part.
#
#   Feel free to use the compute_distances function.  We're testing just
#   three norms, but be advised that this could still take some time.
#   You're welcome to write a vectorized form of the L1- and Linf- norms
#   to speed this up, but it is not necessary.
# ================================================================ #
norm_to_errs = {}
for norm in norms:
    overall_error = 0
    for i in range(num_folds):
        # leave out i for testing, and train on all of the others
        training_X = np.vstack([fold for idx, fold in enumerate(X_train_folds)
if idx != i])
        training_Y = np.hstack([fold for idx, fold in enumerate(y_train_folds)
if idx !=i])
        knn.train(training_X, training_Y)
        cv_X, cv_Y = np.array(X_train_folds[i]), np.array(y_train_folds[i])
        cv_distances = knn.compute_distances(cv_X, norm = norm)
        cv_error = get_error(cv_distances, cv_Y, k = 10)
        overall_error+=cv_error
    avg_err = overall_error/num_folds
    print("got average error {} for norm = {}".format(avg_err,
norm_map[norm]))
    norm_to_errs[norm] = avg_err
    norm_vals.append(avg_err)




min_key = min(norm_to_errs, key = norm_to_errs.get)
print("{} norm had lowest error of {}".format(norm_map[min_key],
norm_to_errs[min_key]))
```

```
x_axis = np.array([1,2,3])
x_ticks = ['L1', 'L2', 'Linf']
plt.xticks(x_axis, x_ticks)
plt.plot(x_axis, norm_vals, 'ro') # norm vals should be ordered as l1, l2 linf
plt.xlabel('Norm')
plt.ylabel('Cross Validation Error')

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
print('Computation time: %.2f'%(time.time()-time_start))
```

```
got average error 0.6916 for norm = l1
got average error 0.7242 for norm = l2
got average error 0.8326 for norm = linf
l1 norm had lowest error of 0.6916
Computation time: 1188.94
```



Questions:

(1) What norm has the best cross-validation error?

(2) What is the cross-validation error for your given norm and k?

Answers:

(1) The L1 norm had the best CV error.

(2) Using k = 10 and the L1 norm, we get a cross validation error of 0.6916.

Evaluating the model on the testing dataset.

Now, given the optimal k and norm you found in earlier parts, evaluate the testing error of the k-nearest neighbors model.

```
error = 1

# ================================================================ #
# YOUR CODE HERE:
#   Evaluate the testing error of the k-nearest neighbors classifier
#   for your optimal hyperparameters found by 5-fold cross-validation.
# ================================================================ #

knn = KNN()
knn.train(X_train, y_train)
distances = knn.compute_distances(X_test, L1_norm)
error = get_error(distances, y_test, k = 10)

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

print('Error rate achieved: {}'.format(error))
```

```
Error rate achieved: 0.722
```

Question:

How much did your error improve by cross-validation over naively choosing $k = 1$ and using the L2-norm?

Answer:

It improved by 0.004

This is the svm workbook for ECE 239AS Assignment #2

Please follow the notebook linearly to implement a linear support vector machine.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and includes code to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training an SVM classifier via gradient descent.

Importing libraries and data setup

```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from cs231n.data_utils import load_CIFAR10 # function to load the CIFAR-10
dataset.
import pdb

# Load matplotlib images inline
%matplotlib inline

# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-
ipython
%load_ext autoreload
%autoreload 2
```

```
# Set the path to the CIFAR-10 data
cifar10_dir = 'cifar-10-batches-py'
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,)
```

```
# 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()
```

```python
# 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)
print('Dev data shape: ', X_dev.shape)
print('Dev labels shape: ', y_dev.shape)
```

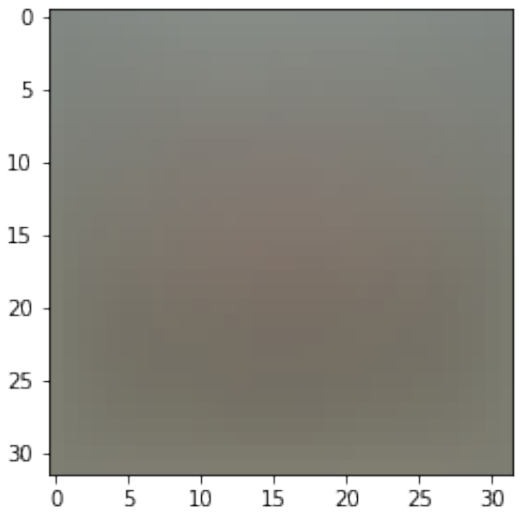
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

```
# 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)
```

```
# 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()
```

```
[ 130.64189796  135.98173469  132.47391837  130.05569388  135.34804082
  131.75402041  130.96055102  136.14328571  132.47636735  131.48467347]
```



```
# 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)
```

```
(49000, 3073) (1000, 3073) (1000, 3073) (500, 3073)
```

Question:

(1) For the SVM, we perform mean-subtraction on the data. However, for the KNN notebook, we did not. Why?

Answer:

(1) In SVM, our scores, and therefore our predictions for our classes, are going to determined by linear combinations of our features with weights, such as $w_i x_i$ for a specific weight/feature combination. This means that if one feature has very large values, then it will influence the classification a lot more than the other features, but if all the features are on the same scale, then this will not occur. We did not need to do this for KNN since KNN makes predicitions based on the

neighbors with closest distances, and the distances themselves will have the same relative ordering regardless of if we scale features up or down uniformly across the dataset, so mean subtraction would not change what the KNN predicts.

Training an SVM

The following cells will take you through building an SVM. You will implement its loss function, then subsequently train it with gradient descent. Finally, you will choose the learning rate of gradient descent to optimize its classification performance.

```
from nndl.svm import SVM
```

```
# Declare an instance of the SVM class.
# Weights are initialized to a random value.
# Note, to keep people's initial solutions consistent, we are going to use a
random seed.

np.random.seed(1)

num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]

svm = SVM(dims=[num_classes, num_features])
```

SVM loss

```
## Implement the loss function for in the SVM class(nndl/svm.py), svm.loss()

loss = svm.loss(X_train, y_train)
print('The training set loss is {}.'.format(loss))

# If you implemented the loss correctly, it should be 15569.98
```

```
The training set loss is 15569.97791541019.
```

SVM gradient

```python
## Calculate the gradient of the SVM class.
# For convenience, we'll write one function that computes the loss
#   and gradient together. Please modify svm.loss_and_grad(X, y).
# You may copy and paste your loss code from svm.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = svm.loss_and_grad(X_dev,y_dev)
print(loss)
# Compare your gradient to a numerical gradient check.
# You should see relative gradient errors on the order of 1e-07 or less if you
implemented the gradient correctly.
svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
15431.6117988
numerical: -10.071826 analytic: -10.071826, relative error: 2.275427e-08
numerical: 6.509800 analytic: 6.509800, relative error: 2.755785e-08
numerical: -0.886501 analytic: -0.886501, relative error: 2.587231e-07
numerical: 3.802422 analytic: 3.802422, relative error: 3.724579e-08
numerical: -9.323489 analytic: -9.323489, relative error: 2.509547e-08
numerical: 5.602462 analytic: 5.602462, relative error: 2.775571e-08
numerical: 4.339594 analytic: 4.339595, relative error: 7.390497e-08
numerical: -12.396901 analytic: -12.396901, relative error: 3.065738e-09
numerical: -9.335607 analytic: -9.335607, relative error: 4.470406e-09
numerical: -17.778925 analytic: -17.778926, relative error: 2.487988e-08
```

A vectorized version of SVM

To speed things up, we will vectorize the loss and gradient calculations. This will be helpful for stochastic gradient descent.

```python
import time
```

```
## Implement svm.fast_loss_and_grad which calculates the loss and gradient
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = svm.loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss,
np.linalg.norm(grad, 'fro'), toc - tic))

tic = time.time()
loss_vectorized, grad_vectorized = svm.fast_loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in
{}s'.format(loss_vectorized, np.linalg.norm(grad_vectorized, 'fro'), toc -
tic))

# The losses should match but your vectorized implementation should be much
faster.
print('difference in loss / grad: {} / {}'.format(loss - loss_vectorized,
np.linalg.norm(grad - grad_vectorized)))

# You should notice a speedup with the same output, i.e., differences on the
order of 1e-12
```

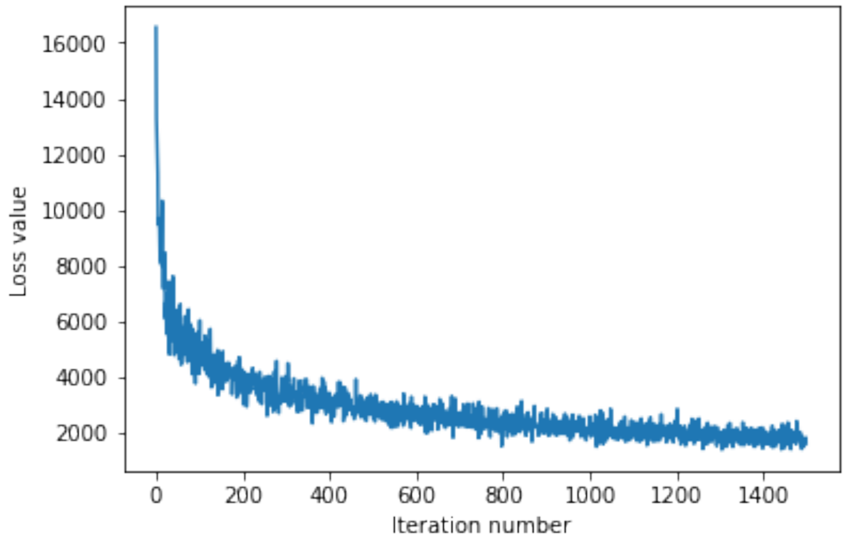
```
Normal loss / grad_norm: 15431.611798772026 / 2272.736768867464 computed in
0.05608487129211426s
Vectorized loss / grad: 15431.611798772023 / 2272.7367688674635 computed in
0.019289016723632812s
difference in loss / grad: 3.637978807091713e-12 / 9.028510448517017e-12
```

Stochastic gradient descent

We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

```python
# Implement svm.train() by filling in the code to extract a batch of data
# and perform the gradient step.

tic = time.time()
loss_hist = svm.train(X_train, y_train, learning_rate=5e-4,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took {}s'.format(toc - tic))

plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 16557.38000190916
iteration 100 / 1500: loss 4701.089451272714
iteration 200 / 1500: loss 4017.333137942788
iteration 300 / 1500: loss 3681.9226471953625
iteration 400 / 1500: loss 2732.6164373988986
iteration 500 / 1500: loss 2786.637842464506
iteration 600 / 1500: loss 2837.0357842782664
iteration 700 / 1500: loss 2206.2348687399317
iteration 800 / 1500: loss 2269.03882411698
iteration 900 / 1500: loss 2543.23781538592
iteration 1000 / 1500: loss 2566.692135726826
iteration 1100 / 1500: loss 2182.068905905164
iteration 1200 / 1500: loss 1861.1182244250447
iteration 1300 / 1500: loss 1982.9013858528256
iteration 1400 / 1500: loss 1927.5204158582114
That took 11.144428014755249s
```



Evaluate the performance of the trained SVM on the validation data.

```
## Implement svm.predict() and use it to compute the training and testing
error.

y_train_pred = svm.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pred),
)))
y_val_pred = svm.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pred)),
))
```

```
training accuracy: 0.28530612244897957
validation accuracy: 0.3
```

Optimize the SVM

Note, to make things faster and simpler, we won't do k-fold cross-validation, but will only optimize the hyperparameters on the validation dataset (X_val, y_val).

```
# ================================================================ #
# YOUR CODE HERE:
#   Train the SVM with different learning rates and evaluate on the
#     validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best VALIDATION accuracy corresponding to the best VALIDATION
error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
#   Note: You do not need to modify SVM class for this section
# ================================================================ #
chosen_rates = [5e-8, 5e-7, 5e-6, 5e-5, 5e-4, 5e-3, 5e-2, 5e-1]
rate_to_err = {}
rate_to_acc = {}
for rate in chosen_rates:
    # train on training data
    svm.train(X_train, y_train, learning_rate=rate,
                  num_iters=1500, verbose=False)
    # predict on validation dataset
    y_val_pred = svm.predict(X_val)
    acc = np.mean(np.equal(y_val, y_val_pred))
    err = 1 - acc
    rate_to_err[rate] = err
    rate_to_acc[rate] = acc
    print('using learning rate = {} validation accuracy: {}, error =
{}'.format(rate, acc, err))

best_rate = min(rate_to_err, key = rate_to_err.get)
print('best learning rate: {} and validation accuracy: {} (corresponding to
validation error {})'.format(best_rate, rate_to_acc[best_rate],
rate_to_err[best_rate]))

# train the best SVM on the training set now
svm.train(X_train, y_train, learning_rate = best_rate, num_iters = 1500,
verbose = False)
# predict on the test set now
y_train_pred = svm.predict(X_test)
acc = np.mean(np.equal(y_test, y_train_pred))
err = 1 - acc
print('Testing error for best SVM using learning rate {} is
{}'.format(best_rate, err))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
using learning rate = 5e-08 validation accuracy: 0.097, error = 0.903
using learning rate = 5e-07 validation accuracy: 0.105, error = 0.895
using learning rate = 5e-06 validation accuracy: 0.173, error = 0.827
using learning rate = 5e-05 validation accuracy: 0.254, error = 0.746
using learning rate = 0.0005 validation accuracy: 0.298, error = 0.702
using learning rate = 0.005 validation accuracy: 0.267, error = 0.733
using learning rate = 0.05 validation accuracy: 0.281, error = 0.719
using learning rate = 0.5 validation accuracy: 0.303, error =
0.6970000000000001
best learning rate: 0.5 and validation accuracy: 0.303 (corresponding to
validation error 0.6970000000000001)
Testing error for best SVM using learning rate 0.5 is 0.728
```

The best learning rate I found was 0.5 with a validation accuracy of 0.303, corresponding to a validation error of 0.697.

Predicting on the test set with that learning rate, I got a testing error of 0.728.

This is the softmax workbook for ECE 239AS Assignment #2

Please follow the notebook linearly to implement a softmax classifier.

Please print out the workbook entirely when completed.

We thank Serena Yeung & Justin Johnson for permission to use code written for the CS 231n class (cs231n.stanford.edu). These are the functions in the cs231n folders and code in the jupyer notebook to preprocess and show the images. The classifiers used are based off of code prepared for CS 231n as well.

The goal of this workbook is to give you experience with training a softmax classifier.

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

%matplotlib inline
%load_ext autoreload
%autoreload 2
```

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,
num_dev=500):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the linear classifier. These are the same steps as we used for the
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # subsample the data
    mask = list(range(num_training, num_training + num_validation))
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = list(range(num_training))
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = list(range(num_test))
    X_test = X_test[mask]
    y_test = y_test[mask]
    mask = np.random.choice(num_training, num_dev, replace=False)
    X_dev = X_train[mask]
    y_dev = y_train[mask]

    # 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))

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis = 0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image
    X_dev -= mean_image

    # add bias dimension and transform into columns
    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))])

    return X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test, X_dev, y_dev =
get_CIFAR10_data()
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)
print('dev data shape: ', X_dev.shape)
print('dev labels shape: ', y_dev.shape)
```

```
Train data shape:  (49000, 3073)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3073)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3073)
Test labels shape:  (1000,)
dev data shape:  (500, 3073)
dev labels shape:  (500,)
```

Training a softmax classifier.

The following cells will take you through building a softmax classifier. You will implement its loss function, then subsequently train it with gradient descent. Finally, you will choose the learning rate of gradient descent to optimize its classification performance.

```
from nndl import Softmax
```

```
# Declare an instance of the Softmax class.
# Weights are initialized to a random value.
# Note, to keep people's first solutions consistent, we are going to use a
random seed.

np.random.seed(1)

num_classes = len(np.unique(y_train))
num_features = X_train.shape[1]

softmax = Softmax(dims=[num_classes, num_features])
print("{} classes, {} features".format(num_classes, num_features))
```

```
10 classes, 3073 features
```

Softmax loss

```
## Implement the loss function of the softmax using a for loop over
#  the number of examples

loss = softmax.loss(X_train, y_train)
```

```
print(loss)
```

```
2.3277607028
```

Question:

You'll notice the loss returned by the softmax is about 2.3 (if implemented correctly). Why does this value make sense?

Answer:

Since we initialize our variables to be random values around 0, we have $E[W] = 0$ (the scaling doesn't affect the expectation). Then we compute the scores with $s = WX$, and invoking the assumption that the features are independent of the weights, we have the expected values of the scores as 0 also.

Now, since $L_i = \log(\sum_{j=1}^{C} \exp(s_j)) - s_{y_i}$ and we expect $s_{y_i} = 0$ and $\exp(s_j) = \exp(0) = 1$, we have $L_i = \log C = \log(10) = 2.3$. So each score is about 2.3, and we add up n of these and divide by n at the end, so the overall loss makes sense to be around 2.3.

Softmax gradient

```
## Calculate the gradient of the softmax loss in the Softmax class.
# For convenience, we'll write one function that computes the loss
#   and gradient together, softmax.loss_and_grad(X, y)
# You may copy and paste your loss code from softmax.loss() here, and then
#   use the appropriate intermediate values to calculate the gradient.

loss, grad = softmax.loss_and_grad(X_dev,y_dev)

# Compare your gradient to a gradient check we wrote.
# You should see relative gradient errors on the order of 1e-07 or less if you
implemented the gradient correctly.
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 0.439395 analytic: 0.439395, relative error: 1.577763e-08
numerical: -0.677415 analytic: -0.677415, relative error: 8.076569e-09
numerical: -0.319042 analytic: -0.319042, relative error: 4.680549e-08
numerical: 1.878243 analytic: 1.878243, relative error: 3.282033e-09
numerical: 2.318809 analytic: 2.318809, relative error: 2.518504e-08
numerical: -0.350131 analytic: -0.350131, relative error: 9.114575e-08
numerical: -1.177781 analytic: -1.177781, relative error: 1.593415e-08
numerical: -0.837689 analytic: -0.837689, relative error: 2.163034e-08
numerical: 1.497596 analytic: 1.497596, relative error: 1.750898e-08
numerical: -2.341154 analytic: -2.341154, relative error: 1.448406e-08
```

A vectorized version of Softmax

To speed things up, we will vectorize the loss and gradient calculations. This will be helpful for stochastic gradient descent.

```
import time
```

```
## Implement softmax.fast_loss_and_grad which calculates the loss and gradient
#    WITHOUT using any for loops.

# Standard loss and gradient
tic = time.time()
loss, grad = softmax.loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss,
np.linalg.norm(grad, 'fro'), toc - tic))

tic = time.time()
loss_vectorized, grad_vectorized = softmax.fast_loss_and_grad(X_dev, y_dev)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in
{}s'.format(loss_vectorized, np.linalg.norm(grad_vectorized, 'fro'), toc -
tic))

# The losses should match but your vectorized implementation should be much
faster.
print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized,
np.linalg.norm(grad - grad_vectorized)))

# You should notice a speedup with the same output.
```

```
Normal loss / grad_norm: 2.331864986233698 / 289.3894464649084 computed in
0.2866799831390381s
Vectorized loss / grad: 2.3318649862336973 / 289.3894464649084 computed in
0.016371965408325195s
difference in loss / grad: 8.881784197001252e-16 /2.4484493005078967e-13
```

Stochastic gradient descent

We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

Question:

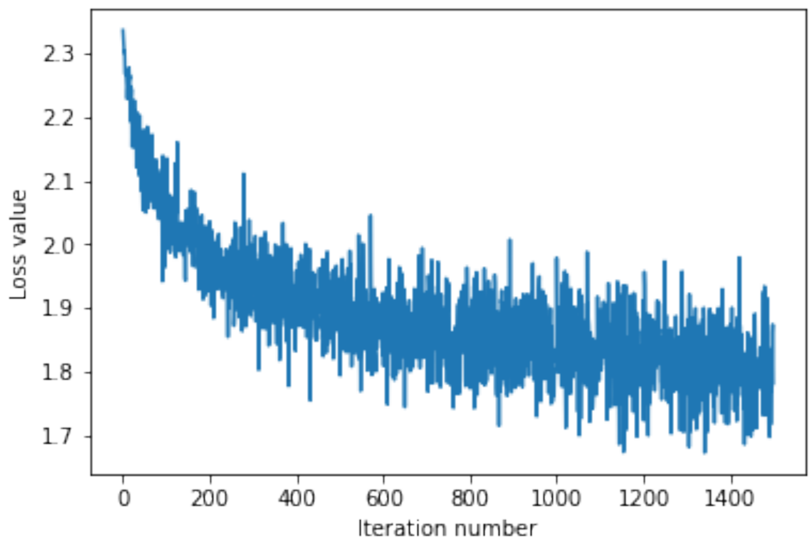
How should the softmax gradient descent training step differ from the svm training step, if at all?

Answer:

It shouldn't differ, given that the general process of gradient descent is to forward pass through the model, compute a loss and gradient (which we call functions for), and then have the same gradient update rule. (The functions for computing the loss and grads are different since the model is different obviously, but the overall gradient descent step is the same).

```
# Implement softmax.train() by filling in the code to extract a batch of data
# and perform the gradient step.
import time


tic = time.time()
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-7,
                      num_iters=1500, verbose=True)
toc = time.time()
print('That took {}s'.format(toc - tic))

plt.plot(loss_hist)
plt.xlabel('Iteration number')
plt.ylabel('Loss value')
plt.show()
```

```
iteration 0 / 1500: loss 2.3365926606637544
iteration 100 / 1500: loss 2.0557222613850827
iteration 200 / 1500: loss 2.0357745120662813
iteration 300 / 1500: loss 1.9813348165609888
iteration 400 / 1500: loss 1.9583142443981614
iteration 500 / 1500: loss 1.862265307354135
iteration 600 / 1500: loss 1.8532611454359382
iteration 700 / 1500: loss 1.835306222372583
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.8992158530357484
iteration 1000 / 1500: loss 1.97835035402523
iteration 1100 / 1500: loss 1.8470797913532633
iteration 1200 / 1500: loss 1.8411450268664082
iteration 1300 / 1500: loss 1.7910402495792102
iteration 1400 / 1500: loss 1.8705803029382257
That took 21.11606502532959s
```



Evaluate the performance of the trained softmax classifier on the validation data.

```
## Implement softmax.predict() and use it to compute the training and testing
error.

y_train_pred = softmax.predict(X_train)
print('training accuracy: {}'.format(np.mean(np.equal(y_train,y_train_pred),
)))
y_val_pred = softmax.predict(X_val)
print('validation accuracy: {}'.format(np.mean(np.equal(y_val, y_val_pred)),
))
```

```
training accuracy: 0.3811428571428571
validation accuracy: 0.398
```

Optimize the softmax classifier

You may copy and paste your optimization code from the SVM here.

```
np.finfo(float).eps
```

```
2.220446049250313e-16
```

```
# ================================================================ #
# YOUR CODE HERE:
#   Train the Softmax classifier with different learning rates and
#     evaluate on the validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best validation accuracy corresponding to the best validation
error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
chosen_rates = [5e-8, 5e-7, 5e-6, 5e-5, 5e-4, 5e-3, 5e-2, 5e-1]
rate_to_err = {}
rate_to_acc = {}
for rate in chosen_rates:
    # train on training data
    softmax.train(X_train, y_train, learning_rate=rate,
                  num_iters=1500, verbose=False)
    # predict on validation dataset
    y_val_pred = softmax.predict(X_val)
    acc = np.mean(np.equal(y_val, y_val_pred))
    err = 1 - acc
    rate_to_err[rate] = err
    rate_to_acc[rate] = acc
    print('Using learning rate = {} validation accuracy: {}, error is
{}'.format(rate, acc, err))

best_rate = min(rate_to_err, key = rate_to_err.get)
print('best learning rate: {} and validation accuracy: {} (corresponding to
validation error {})'.format(best_rate, rate_to_acc[best_rate],
rate_to_err[best_rate]))
# train the softmax with the best learning rate on the training set
softmax.train(X_train, y_train, learning_rate = best_rate, num_iters = 1500,
verbose = False)
# run the softmax prediction on the test set now
y_train_pred = softmax.predict(X_test)
acc = np.mean(np.equal(y_test, y_train_pred))
err = 1 - acc
print('Testing error for best softmax using learning rate {} is
{}'.format(best_rate, err))
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

```
Using learning rate = 5e-08 validation accuracy: 0.37, error is 0.63
Using learning rate = 5e-07 validation accuracy: 0.407, error is 0.593
Using learning rate = 5e-06 validation accuracy: 0.38, error is 0.62
Using learning rate = 5e-05 validation accuracy: 0.272, error is 0.728
Using learning rate = 0.0005 validation accuracy: 0.308, error is 0.692
Using learning rate = 0.005 validation accuracy: 0.262, error is 0.738
Using learning rate = 0.05 validation accuracy: 0.295, error is
0.7050000000000001
Using learning rate = 0.5 validation accuracy: 0.293, error is
0.7070000000000001
best learning rate: 5e-07 and validation accuracy: 0.407 (corresponding to
validation error 0.593)
Testing error for best softmax using learning rate 5e-07 is 0.618
```

The best learnign rate I found was $5e-7$ that had a validation accuracy of 0.407 corresponding to a validation error of 0.407.

Using this learning rate and running predictions on the testing set, I got a testing error of 0.618.

import numpy as np import pdb

""" This code was based off of code from cs231n at Stanford University, and modified for ece239as at UCLA. """

class KNN(object):

def **init**(self):

```
pass
```

def train(self, X, y):

```
"""
Inputs:
- X is a numpy array of size (num_examples, D)
- y is a numpy array of size (num_examples, )
"""
self.X_train = X
self.y_train = y
```

def compute_distances(self, X, norm=None):

```
"""
Compute the distance between each test point in X and each training point
in self.X_train.

Inputs:
- X: A numpy array of shape (num_test, D) containing test data.
- norm: the function with which the norm is taken.

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.
"""
if norm is None:
  norm = lambda x: np.sqrt(np.sum(x**2))
  #norm = 2

num_test = X.shape[0]
num_train = self.X_train.shape[0]
dists = np.zeros((num_test, num_train))
for i in np.arange(num_test):

  for j in np.arange(num_train):
    # ================================================================ #
    # YOUR CODE HERE:
    #   Compute the distance between the ith test point and the jth
    #   training point using norm(), and store the result in dists[i, j].
    # ================================================================ #

    dists[i][j] = norm(X[i] - self.X_train[j])

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

return dists
```

def compute_L2_distances_vectorized(self, X):

```
"""
Compute the distance between each test point in X and each training point
in self.X_train WITHOUT using any for loops.

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))

# ================================================================ #
# YOUR CODE HERE:
#   Compute the L2 distance between the ith test point and the jth
#   training point and store the result in dists[i, j].  You may
#    NOT use a for loop (or list comprehension).  You may only use
#     numpy operations.
#
#     HINT: use broadcasting.  If you have a shape (N,1) array and
#   a shape (M,) array, adding them together produces a shape (N, M)
#   array.
# ================================================================ #

vector_norms_X_train = np.array(np.sum(self.X_train**2, axis = 1)) # shape is
5000, column vector where each element is the norm of that feature vec
vector_norms_X_train =
vector_norms_X_train.reshape((vector_norms_X_train.shape[0], 1)) # reshape to
broadcast
vector_norms_X = np.array(np.sum(X**2, axis = 1)) # do the same thing for the
input examples
#broadcast operation: we have a (5000, 1) and a (500,)
sums = vector_norms_X_train + vector_norms_X
# basically, sums[i] will be a vector that is equal to vector_norms_X +
vector_norms_X_train[i] (where the latter is a single element, and the sum is
taken element wise on the vector)
# so the below assert should pass
#assert sums[0].all() == (vector_norms_X + vector_norms_X_train[0]).all()
dists = sums.T - 2 * X.dot(self.X_train.T)
dists = np.sqrt(dists)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return dists
```

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 np.arange(num_test):
  # A list of length k storing the labels of the k nearest neighbors to
  # the ith test point.
  closest_y = []
  # ================================================================ #
  # YOUR CODE HERE:
  #   Use the distances to calculate and then store the labels of
  #   the k-nearest neighbors to the ith test point.  The function
  #   numpy.argsort may be useful.
  #
  #   After doing this, find the most common label of the k-nearest
  #   neighbors.  Store the predicted label of the ith training example
  #   as y_pred[i].  Break ties by choosing the smaller label.
  # ================================================================ #

  sorted_indices = list(np.argsort(dists[i]))
  k_closest_idx = sorted_indices[:k]
  label_to_occ = {}
  for idx in k_closest_idx:
    label = self.y_train[idx]
    closest_y.append(label)
    label_to_occ[label] = label_to_occ.get(label, 0) + 1
  # vote: get the most common label, break ties by index
  best_label, best_occ = None, 0
  for key, val in label_to_occ.items():
    if val > best_occ:
      best_label, best_occ = key, val
    elif val == best_occ:
      best_label = min(best_label, key)
  y_pred[i] = best_label
```

```
    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #

  return y_pred
```

import numpy as np

class Softmax(object):

def **init**(self, dims=[10, 3073]):

```
self.init_weights(dims=dims)
```

def init_weights(self, dims):

```
"""
Initializes the weight matrix of the Softmax classifier.
Note that it has shape (C, D) where C is the number of
classes and D is the feature size.
"""
self.W = np.random.normal(size=dims) * 0.0001
```

def loss(self, X, y):

```
"""
Calculates the softmax loss.

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

Inputs:
- 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.

Returns a tuple of:
- loss as single float
"""

# Initialize the loss to zero.
loss = 0.0

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the normalized softmax loss.  Store it as the variable loss.
#   (That is, calculate the sum of the losses of all the training
#   set margins, and then normalize the loss by the number of
#   training examples.)
# ================================================================ #

scores = X.dot(self.W.T)
for i in range(scores.shape[0]):
  score_vec = scores[i]
  score_vec-=np.max(score_vec)
  correct_label = y[i]
  corresponding_score = score_vec[correct_label]
  loss+= -corresponding_score
  loss+=np.log(np.sum(np.exp(score_vec)))
loss/=X.shape[0]
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return loss
```

def loss_and_grad(self, X, y):

```
"""
Same as self.loss(X, y), except that it also returns the gradient.

Output: grad -- a matrix of the same dimensions as W containing
    the gradient of the loss with respect to W.
"""

# Initialize the loss and gradient to zero.
loss = 0.0
grad = np.zeros_like(self.W)

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the softmax loss and the gradient. Store the gradient
#   as the variable grad.
# ================================================================ #
scores = X.dot(self.W.T)
num_train = X.shape[0]
for i in range(scores.shape[0]):
  score_vec = scores[i]
  score_vec-=np.max(score_vec)
  correct_label = y[i]
  corresponding_score = score_vec[correct_label]
  loss+= -corresponding_score
  exp = np.exp(score_vec)
  loss+=np.log(np.sum(exp))
  for j in range(scores.shape[1]):
    grad[j] += (np.exp(score_vec[j])/np.sum(exp)) * X[i]
  grad[correct_label]+=-X[i]
loss/=X.shape[0]
grad/=X.shape[0]

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return loss, grad
```

def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):

```
"""
sample a few random elements and only return numerical
in these dimensions.
"""

for i in np.arange(num_checks):
  ix = tuple([np.random.randint(m) for m in self.W.shape])

  oldval = self.W[ix]
  self.W[ix] = oldval + h # increment by h
  fxph = self.loss(X, y)
  self.W[ix] = oldval - h # decrement by h
  fxmh = self.loss(X,y) # evaluate f(x - h)
  self.W[ix] = oldval # reset

  grad_numerical = (fxph - fxmh) / (2 * h)
  grad_analytic = your_grad[ix]
  rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) +
abs(grad_analytic))
  print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical,
grad_analytic, rel_error))
```

def fast_loss_and_grad(self, X, y):

```
"""
A vectorized implementation of loss_and_grad. It shares the same
inputs and ouptuts as loss_and_grad.
"""
loss = 0.0
grad = np.zeros(self.W.shape) # initialize the gradient as zero

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the softmax loss and gradient WITHOUT any for loops.
# ================================================================ #
scores = X.dot(self.W.T)
scores-=np.max(scores, axis = 1, keepdims = True) # NORMALIZE HERE
# run exp
expscores = np.exp(scores)
n = expscores/np.sum(expscores, axis = 1, keepdims = True)
# pick out the correct labels
correct_label_expscores = n[range(X.shape[0]),y]
# divide by the softmax denominator
#correct_label_expscores/=np.sum(expscores,axis=1)
# take the log
l = -np.log(correct_label_expscores.clip(min=np.finfo(float).eps))
# return the loss
loss = np.sum(l)/X.shape[0]

# compute all of the normalized softmax scores
# the gradient for a specific w_j will be (1/sum(expscores)) * e^(wjx^i), so
store that
p = expscores/np.sum(expscores,axis=1,keepdims=True)
# to account for the case where w_j = w_{y_i} (i.e. our class corresponds to
the correct class label)
# in the unvectorized version we multiplied by -x[i], so here we -1 and dot
with X
p[range(X.shape[0]),y]-=1
# now we dot with x (because of the chain rule)
grad = X.T.dot(p).T
grad/=X.shape[0]
```

```
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return loss, grad
```

def train(self, X, y, learning_rate=1e-3, num_iters=100,

```
        batch_size=200, verbose=False):
"""
Train this linear classifier using stochastic gradient descent.

Inputs:
- X: A numpy array of shape (N, D) containing training data; there are N
  training samples each of dimension D.
- y: A numpy array of shape (N,) containing training labels; y[i] = c
  means that X[i] has label 0 <= c < C for C classes.
- learning_rate: (float) learning rate for optimization.
- num_iters: (integer) number of steps to take when optimizing
- batch_size: (integer) number of training examples to use at each step.
- verbose: (boolean) If true, print progress during optimization.

Outputs:
A list containing the value of the loss function at each training iteration.
"""
num_train, dim = X.shape
num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number
of classes

self.init_weights(dims=[np.max(y) + 1, X.shape[1]]) # initializes the weights
of self.W

# Run stochastic gradient descent to optimize W
loss_history = []

for it in np.arange(num_iters):
  X_batch = None
  y_batch = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Sample batch_size elements from the training data for use in
  #      gradient descent.  After sampling,
  #     - X_batch should have shape: (dim, batch_size)
  #     - y_batch should have shape: (batch_size,)
  #   The indices should be randomly generated to reduce correlations
  #   in the dataset.  Use np.random.choice.  It's okay to sample with
  #   replacement.
  # ================================================================ #
  indices = np.random.choice(X.shape[0], batch_size)
  X_batch = X[indices]
  y_batch = y[indices]
  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  # evaluate loss and gradient
  loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
  loss_history.append(loss)

  # ================================================================ #
  # YOUR CODE HERE:
  #   Update the parameters, self.W, with a gradient step
  # ================================================================ #
  self.W+=-learning_rate*grad

  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  if verbose and it % 100 == 0:
    print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

return loss_history
```

def predict(self, X):

```
"""
Inputs:
- X: N x D array of training data. Each row is a D-dimensional point.

Returns:
- y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
  array of length N, and each element is an integer giving the predicted
  class.
"""
y_pred = np.zeros(X.shape[1])
# ================================================================ #
# YOUR CODE HERE:
#   Predict the labels given the training data.
# ================================================================ #
scores = np.dot(X, self.W.T)
y_pred = np.argmax(scores, axis = 1)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return y_pred
```

import numpy as np import pdb

""" This code was based off of code from cs231n at Stanford University, and modified for ece239as at UCLA. """ class SVM(object):

def **init**(self, dims=[10, 3073]):

```
self.init_weights(dims=dims)
```

def init_weights(self, dims):

```
"""
Initializes the weight matrix of the SVM.  Note that it has shape (C, D)
where C is the number of classes and D is the feature size.
"""
self.W = np.random.normal(size=dims)
```

def loss(self, X, y):

```
"""
Calculates the SVM loss.

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

Inputs:
- 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.

Returns a tuple of:
- loss as single float
"""

# compute the loss and the gradient
num_classes = self.W.shape[0]
num_train = X.shape[0]
loss = 0.0

# X is N * D, y is N weights are C * D
# ================================================================ #
# YOUR CODE HERE:
#   Calculate the normalized SVM loss, and store it as 'loss'.
#   (That is, calculate the sum of the losses of all the training
#   set margins, and then normalize the loss by the number of
#   training examples.)
# ================================================================ #
for i in np.arange(num_train):
  cur_x, label = X[i], y[i]
  scores = cur_x.dot(self.W.T)
  for j in range(num_classes):
    loss+=max(0, 1 + scores[j] - scores[label]) if label != j else 0
loss/=num_train
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return loss
```

def loss_and_grad(self, X, y):

```
"""
Same as self.loss(X, y), except that it also returns the gradient.

Output: grad -- a matrix of the same dimensions as W containing
    the gradient of the loss with respect to W.
"""

# compute the loss and the gradient
num_classes = self.W.shape[0]
num_train = X.shape[0]
loss = 0.0
grad = np.zeros_like(self.W) # W is C * D
for i in np.arange(num_train):
# ================================================================ #
# YOUR CODE HERE:
#   Calculate the SVM loss and the gradient.  Store the gradient in
#   the variable grad.
# ================================================================ #
# forward prop x and compute the loss.
  cur_x, label = X[i], y[i]
  scores = cur_x.dot(self.W.T)
  num_active_classes = 0
  for j in range(num_classes):
    inner_score_diff = 1 + scores[j] - scores[label]
    active = inner_score_diff > 0
    if active:
      num_active_classes+=1
    if label != j and active:
      loss+=inner_score_diff
    grad[j]+=cur_x if active else 0
  grad[label]-=num_active_classes*cur_x
```

```
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #f
loss /= num_train
grad /= num_train

return loss, grad
```

def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):

```
"""
sample a few random elements and only return numerical
in these dimensions.
"""

for i in np.arange(num_checks):
  ix = tuple([np.random.randint(m) for m in self.W.shape])

  oldval = self.W[ix]
  self.W[ix] = oldval + h # increment by h
  fxph = self.loss(X, y)
  self.W[ix] = oldval - h # decrement by h
  fxmh = self.loss(X,y) # evaluate f(x - h)
  self.W[ix] = oldval # reset

  grad_numerical = (fxph - fxmh) / (2 * h)
  grad_analytic = your_grad[ix]
  rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) +
abs(grad_analytic))
  print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical,
grad_analytic, rel_error))
```

def fast_loss_and_grad(self, X, y):

```
"""
A vectorized implementation of loss_and_grad. It shares the same
inputs and ouptuts as loss_and_grad.
"""
loss = 0.0
grad = np.zeros(self.W.shape) # initialize the gradient as zero

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the SVM loss WITHOUT any for loops.
# ================================================================ #
scores = np.dot(X, self.W.T).T
correct_label_picker = y, range(scores.shape[1]) # a tuple that allows us to
index the correct class scores.
# pick out the correct predictions
# this is done by indexing the np array scores by a tuple, namely a list of
all the labels
# going along the cols of x.
correct = scores[correct_label_picker] # pick off scores at that index
# now compute a vectorized difference of the class scores and actual scores,
adding the ones as a margin
# note that this is not completely correct right now, as we're summing across
teh correct labels too.
vectorized_loss = scores - correct + np.ones(scores.shape) # add the margin
# due to the above, we don't sum across the correct labels, so set those to 0
vectorized_loss[correct_label_picker] = 0
vectorized_loss[vectorized_loss< 0] = 0 # threshold at 0, because we take all
negative values as 0
loss = np.sum(vectorized_loss)/X.shape[0]
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
# ================================================================ #
# YOUR CODE HERE:
#   Calculate the SVM grad WITHOUT any for loops.
# ================================================================ #

# for the losses > 0, we threshold them at 1 so that when we dot with x, it
contributes x_i to the gradient
vectorized_loss[vectorized_loss > 0] = 1
# now, we calculate the gradients pertaining to y_i.
# this is done by taking x_i and scaling by the number of times the margin was
> 0 (i.e. active)
vectorized_loss[correct_label_picker] = -1 * np.sum(vectorized_loss, axis=0)
# now we include the x_i's for the gradient by dotting, and then normalize.
grad = np.dot(vectorized_loss, X)/X.shape[0]

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return loss, grad
```

def train(self, X, y, learning_rate=1e-3, num_iters=100,

```
        batch_size=200, verbose=False):
"""
Train this linear classifier using stochastic gradient descent.

Inputs:
- X: A numpy array of shape (N, D) containing training data; there are N
  training samples each of dimension D.
- y: A numpy array of shape (N,) containing training labels; y[i] = c
  means that X[i] has label 0 <= c < C for C classes.
- learning_rate: (float) learning rate for optimization.
- num_iters: (integer) number of steps to take when optimizing
- batch_size: (integer) number of training examples to use at each step.
- verbose: (boolean) If true, print progress during optimization.

Outputs:
A list containing the value of the loss function at each training iteration.
"""
num_train, dim = X.shape
num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number
of classes
```

```
self.init_weights(dims=[np.max(y) + 1, X.shape[1]]) # initializes the weights
of self.W

# Run stochastic gradient descent to optimize W
loss_history = []

for it in np.arange(num_iters):
  X_batch = None
  y_batch = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Sample batch_size elements from the training data for use in
  #   gradient descent.  After sampling,
  #     - X_batch should have shape: (dim, batch_size)
  #     - y_batch should have shape: (batch_size,)
  #   The indices should be randomly generated to reduce correlations
  #   in the dataset.  Use np.random.choice.  It's okay to sample with
  #   replacement.
  # ================================================================ #

  indices = np.random.choice(X.shape[0], batch_size)
  X_batch = X[indices]
  y_batch = y[indices]
  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  # evaluate loss and gradient
  loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
  loss_history.append(loss)

  # ================================================================ #
  # YOUR CODE HERE:
  #   Update the parameters, self.W, with a gradient step
  # ================================================================ #
  self.W+=-learning_rate*grad
  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

  if verbose and it % 100 == 0:
    print('iteration {} / {}: loss {}'.format(it, num_iters, loss))

return loss_history
```

def predict(self, X):

```
"""
Inputs:
- X: N x D array of training data. Each row is a D-dimensional point.

Returns:
- y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
  array of length N, and each element is an integer giving the predicted
  class.
"""
y_pred = np.zeros(X.shape[1])
```

```
# ================================================================ #
# YOUR CODE HERE:
#   Predict the labels given the training data with the parameter self.W.
# ================================================================ #
scores = np.dot(X, self.W.T)
y_pred = np.argmax(scores, axis = 1)

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return y_pred
```