knn

January 31, 2018

0.1 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.

0.2 Import the appropriate libraries

```
In [17]: 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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
In [18]: # 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,)
```

```
In [19]: # 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', 'tru
         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()
```

```
In [20]: # 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)
```

1 K-nearest neighbors

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

```
In [21]: # Import the KNN class

         from nndl import KNN
```

```
In [22]: # 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)
```

1.1 Questions

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

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

1.2 Answers

(1) In the function knn.train(), we store the whole data consisting of features and labels which will be used to train the knn classifier. This data will also be used for prediction step as well.

(2) The cons is that we need to store the whole data and keep it for both training and prediction. This means that a lot of memory is needed for knn classifier. On the other hand, since it is based on calculation of norms, it is a simple algorithm, and upon good implementation (e.g. vectorization instead of for loops), it is fast too.

1.3 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.

```
In [23]: # 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: 45.79017782211304
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

1.3.1 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.

```
In [24]: # 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 loo
         # 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): {}'
```

```
Time to run code: 0.39802050590515137
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.

1.3.2 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.

```
In [26]: # 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.
         # ================================================================ #

         y_est = knn.predict_labels(dists_L2_vectorized,1)

         y_diff = (y_test - y_est)
         num_incorrect = np.count_nonzero(y_diff)

         error = num_incorrect / num_test


         pass

         # ================================================================ #
         # 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%.

2 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.

2.0.1 Create training and validation folds

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

```
In [27]: # 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]
         # ================================================================ #


         print(X_train.shape)

         for i in range(num_folds):
             X_train_folds.append(X_train[i*1000:(i+1)*1000,:])
             y_train_folds.append(y_train[i*1000:(i+1)*1000])

         pass

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

```
(5000, 3072)
```

2.0.2 Optimizing the number of nearest neighbors hyperparameter.

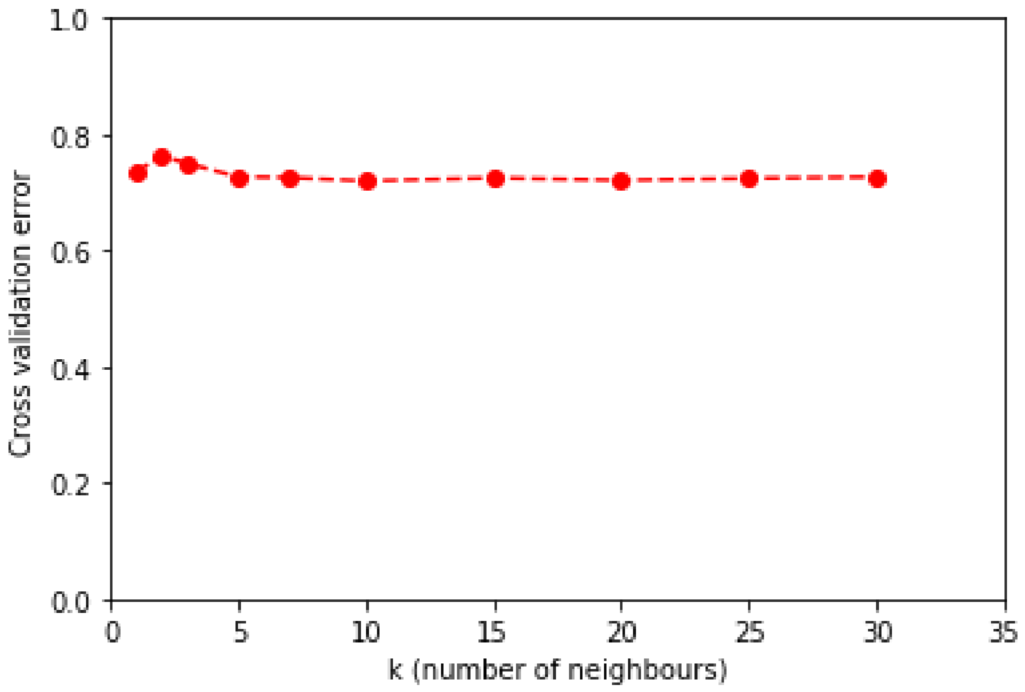
In this section, we select different numbers of nearest neighbors and assess which one has the lowest k-fold cross validation error.

```
In [28]: time_start =time.time()

         ks = [1, 2, 3, 5, 7, 10, 15, 20, 25, 30]
         error = []
         # ================================================================ #
         # 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.
         # ================================================================ #
```

```
num_test_fold = 1000

for k in ks:
    error2 = 0
    error1 = 0
    for i in range(num_folds):
        knn = KNN()

        X_test_kFold = X_train_folds[i]
        y_test_kFold = y_train_folds[i]

        X_train_kFold = []
        y_train_kFold = []
        #mask = list(range(num_training))
        #del mask[i*1000:(i+1)*1000]

        for j in range(num_folds):
            if i != j:
                X_train_kFold.extend(X_train_folds[j])
                y_train_kFold.extend(y_train_folds[j])

        X_train_kFold = np.array(X_train_kFold)
        y_train_kFold = np.array(y_train_kFold)

        #X_train_kFold = X_train[mask]
        #y_train_kFold = y_train[mask]

        knn.train(X=X_train_kFold, y=y_train_kFold)

        dists_fold = knn.compute_L2_distances_vectorized(X_test_kFold)
        y_est_fold = knn.predict_labels(dists_fold,k)
        y_diff_fold = (y_test_kFold - y_est_fold)

        num_correct = np.sum(y_test_kFold == y_est_fold)
        #num_incorrect_fold = np.count_nonzero(y_diff_fold)
        #error1 = num_incorrect_fold / num_test_fold

        error1 = (num_test_fold - num_correct)/num_test_fold

        error2 += error1

    error.append(error2/num_folds)


pass
```

```
for j in np.arange(len(error)):
    print(error[j],ks[j])

x_index = ks
y_value = error
plt.plot(x_index, y_value, 'ro--')
plt.axis([0, 35, 0, 1])
plt.xlabel('k (number of neighbours)')
plt.ylabel('Cross validation error')
plt.show()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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

```
0.7344 1
0.7626000000000002 2
0.7504000000000001 3
0.7267999999999999 5
0.7256 7
0.7198 10
0.725 15
0.721 20
0.7242 25
0.7266 30
```



```
Computation time: 34.17
```

2.1 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?

2.2 Answers:

(1) k = 10 has the lowest error value and hence the best option. In should be noted that the error values after 5 are close to each other and k = 10 is the best one.

(2) 0.7198.

2.2.1 Optimizing the norm

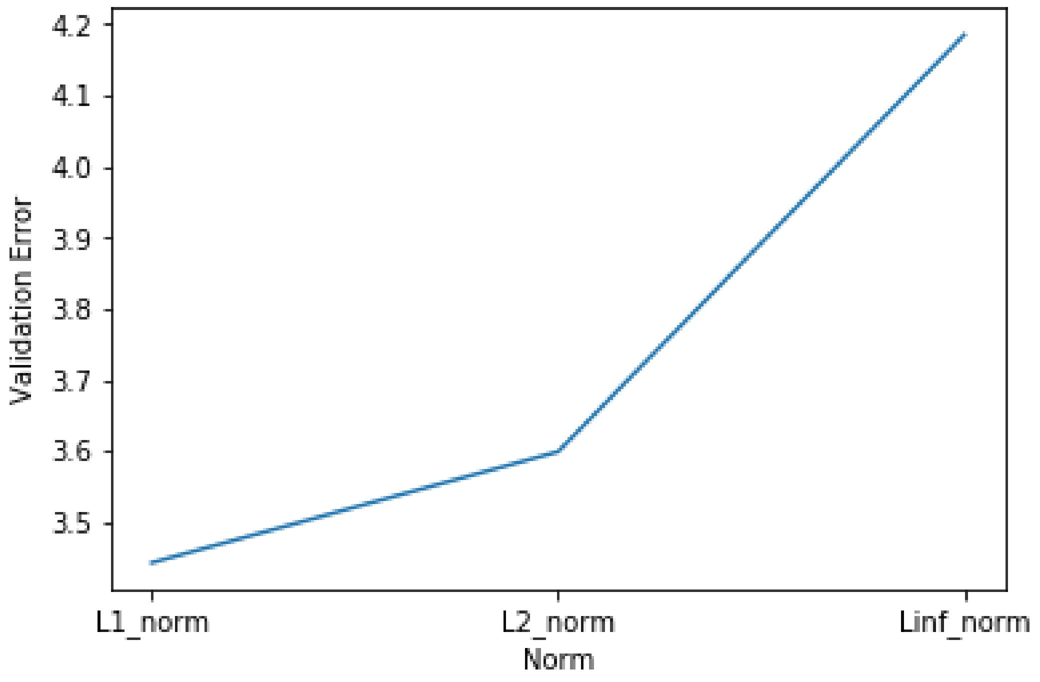
Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

```
In [29]: 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]

         # ================================================================ #
         # 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.
         # ================================================================ #

         error = []

         for L in norms:
             error2 = 0
             error1 = 0
```

```
    for i in range(num_folds):
        print(str(L),i)
        knn = KNN()

        X_test_kFold = X_train_folds[i]
        y_test_kFold = y_train_folds[i]

        X_train_kFold = []
        y_train_kFold = []
        #mask = list(range(num_training))
        #del mask[i*1000:(i+1)*1000]

        for j in range(num_folds):
            if i != j:
                X_train_kFold.extend(X_train_folds[j])
                y_train_kFold.extend(y_train_folds[j])

        X_train_kFold = np.array(X_train_kFold)
        y_train_kFold = np.array(y_train_kFold)

        #X_train_kFold = X_train[mask]
        #y_train_kFold = y_train[mask]

        knn.train(X=X_train_kFold, y=y_train_kFold)

        dists_fold = knn.compute_distances(X_test_kFold,L)
        y_est_fold = knn.predict_labels(dists_fold,10)
        y_diff_fold = (y_test_kFold - y_est_fold)

        num_correct = np.sum(y_test_kFold == y_est_fold)
        #num_incorrect_fold = np.count_nonzero(y_diff_fold)
        #error1 = num_incorrect_fold / num_test_fold

        error1 = (num_test_fold - num_correct)/num_test_fold

        error2 += error1

    error.append(error2/5)



print(error)
pass

plt.figure()
plt.plot(error)
plt.xlabel('Norm')
plt.ylabel('Validation Error')
```

```
    plt.xticks(np.arange(3), ['L1_norm', 'L2_norm', 'Linf_norm'])
    # ============================================================= #
    # END YOUR CODE HERE
    # ============================================================= #
    print('Computation time: %.2f'%(time.time()-time_start))
```

```
<function <lambda> at 0x000000D0041B8B70> 0
<function <lambda> at 0x000000D0041B8B70> 1
<function <lambda> at 0x000000D0041B8B70> 2
<function <lambda> at 0x000000D0041B8B70> 3
<function <lambda> at 0x000000D0041B8B70> 4
<function <lambda> at 0x000000D002D9F0D0> 0
<function <lambda> at 0x000000D002D9F0D0> 1
<function <lambda> at 0x000000D002D9F0D0> 2
<function <lambda> at 0x000000D002D9F0D0> 3
<function <lambda> at 0x000000D002D9F0D0> 4
<function <lambda> at 0x000000D002D9F8C8> 0
<function <lambda> at 0x000000D002D9F8C8> 1
<function <lambda> at 0x000000D002D9F8C8> 2
<function <lambda> at 0x000000D002D9F8C8> 3
<function <lambda> at 0x000000D002D9F8C8> 4
[3.4430000000000005, 3.599, 4.1850000000000005]
Computation time: 931.30
```



2.3 Questions:

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

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

2.4 Answers:

(1) L1 norm is the best one.

(2) 3.44/5 = 0.688. Note that in the above plot the y axis should be divided by for in order to average.

3 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.

```
In [30]: 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.
         # ================================================================ #

         L1_norm = lambda x: np.linalg.norm(x, ord=1)
         k_opt = 10
         L_opt = L1_norm

         knn.train(X=X_train, y=y_train)
         dists_final = knn.compute_distances(X_test,L_opt)
         y_est_final = knn.predict_labels(dists_final,k_opt)

         error = np.mean(y_est_final != y_test)

         pass

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

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

```
Error rate achieved: 0.722
```

3.1 Question:

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

3.2 Answer:

From L2 and k = 1 we obtained 0.726 amd from L1 and k=10 we obtained 0.722. Hence, we got 100*((0.726-0.722)/0.726) = 0.55% improvement.

```
import numpy as np
import pdb

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

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,:])  #ith test point and jth training point

        pass

        # ================================================================ #
        # 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.
    # ======================================================================= #

    X_SumSquare = np.sum(np.square(X),axis=1);
    X_train_SumSquare = np.sum(np.square(self.X_train),axis=1);
    mul = np.dot(X, self.X_train.T);
    dists = np.sqrt(X_SumSquare[:,np.newaxis]+X_train_SumSquare-2*mul)
    pass

    # ======================================================================= #
    # 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.
      # ======================================================================= #

      #indices = range(k);
      #closest_y.append(np.take(np.argsort(dists[i,:]), indices))   # k indices of smallest L2

      #class_numbers = np.zeros(10)    # list


      #for j in closest_y:
       #     class_numbers[self.y_train[j]] += 1
           # print(self.y_train[j])


       #y_pred[i] = np.argmax(class_numbers)


    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 = []

      y_indicies = np.argsort(dists[i, :], axis = 0)
      closest_y = self.y_train[y_indicies[:k]]

      y_pred[i] = np.argmax(np.bincount(closest_y))



  pass

    # ================================================================ #
    # 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
    num_classes = self.W.shape[0]     # C = num_classes
    num_train = X.shape[0]

    exp_a = np.zeros((num_classes,num_train))
    # ================================================================ #
    # 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.)
    # ================================================================ #


    for i in np.arange(num_train):

        Loss = 0.0

        class_scores = np.dot(self.W,X[i,:].T)        # calculating class scores (C x 1 vector
        class_scores -= np.max(class_scores)          # considering the possible issue for nur

        exp_a[:,i] = np.exp(class_scores)                 # turning class scores to probabilit

        Loss -= np.log(exp_a[y[i],i]/np.sum(exp_a[:,i]))


        #p[:,i] = exp_a[:,i]/np.sum(exp_a[:,i])                # p now is a valid probability
        #print(p[:,i])

        loss += Loss
        #print(Loss,i)

    pass
    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.
      """

    # Initialize the loss and gradient to zero.
    loss = 0.0
    grad = np.zeros_like(self.W)
    grad_tmp = np.zeros_like(self.W)
    num_classes = self.W.shape[0]     # C = num_classes
    num_train = X.shape[0]

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the gradient
    #   as the variable grad.
    # ================================================================ #

    exp_a = np.zeros((num_classes,num_train))
    for i in np.arange(num_train):

        Loss = 0.0

        class_scores = np.dot(self.W,X[i,:].T)        # calculating class scores (C x 1 vector
        class_scores -= np.max(class_scores)          # considering the possible issue for nur

        exp_a[:,i] = np.exp(class_scores)                 # turning class scores to probabilit

        Loss -= np.log(exp_a[y[i],i]/np.sum(exp_a[:,i]))


        #if i==0:
        grada = np.zeros(X.shape[1])

        for j in range(num_classes):
            if j != y[i]:
                grad_tmp[j,:] = X[i,:].T * (exp_a[j,i] / np.sum(exp_a[:,i]))
            else:
                grad_tmp[j,:] = X[i,:].T * (exp_a[j,i] / np.sum(exp_a[:,i])) - X[i,:].T

        grad += grad_tmp
        loss += Loss

    pass


    loss /= num_train
    grad /= num_train
    # ================================================================ #
    # 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_analyt
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic,

  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.
    # ================================================================ #

    num_train = X.shape[0]
    num_classes = self.W.shape[0]

#     # vectorized loss calculation #
    class_scores_matrix = np.dot(self.W,X.T)       # calculating class scores matrix (C x m):
    class_scores_matrix -= np.max(class_scores_matrix)      # considering the possible issue
    exp_a = np.exp(class_scores_matrix)           # calculating the exponents

#     y_exp = np.array(exp_a[y, np.arange(0, class_scores_matrix.shape[1])])
#     #print(exp_a[:,:3])
#     #print(y[:3])
#     #print(y_exp[:3])

#     tt = np.sum(exp_a,axis=0)
#     tt2 = np.divide(tt,y_exp)
#     print(num_train)
#     tt3 = np.power(tt2,1/num_train)
#     loss = np.log(np.prod(tt3))



    (C, D) = self.W.shape
    N = X.shape[0]

    scores = np.dot(self.W, X.T)
    scores -= np.max(scores) # shift by log C to avoid numerical instability

    y_mat = np.zeros(shape = (C, N))
    y_mat[y, range(N)] = 1

    # matrix of all zeros except for a single wx + log C value in each column that corresponds
    # quantity we need to subtract from each row of scores
    correct_wx = np.multiply(y_mat, scores)

    # create a single row of the correct wx_y + log C values for each data point
    sums_wy = np.sum(correct_wx, axis=0) # sum over each column

    exp_scores = np.exp(scores)
    sums_exp = np.sum(exp_scores, axis=0) # sum over each column
    result = np.log(sums_exp)

    result -= sums_wy

    loss = np.sum(result)
    loss /= num_train


    # vectorized gradient calculation #
    exp_a_sum = np.sum(exp_a,axis=0)

    y_mat_corres = np.zeros(shape = (num_classes, num_train))
    y_mat_corres[y, range(num_train)] = 1
    sum_exp_scores = np.sum(exp_a, axis=0)
    sum_exp_scores = 1.0 / exp_a_sum   # division by sum over columns
    exp_a *= sum_exp_scores
    grad = np.dot(exp_a, X)
    grad -= np.dot(y_mat_corres, X)
    grad /= num_train


    # ================================================================ #
    # 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.
        # ================================================================ #
        mask = np.random.choice(num_train, batch_size, replace=True)

        X_batch = X[mask]                          # (dim, batch_size)
        y_batch = y[mask]                          # (batch_size,)

        pass


        # ================================================================ #
        # 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
        # ================================================================ #
        pass

        self.W = 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.
# ================================================================ #

y_pred = np.argmax(np.exp(self.W.dot(X.T)), axis=0)

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

return y_pred
```

softmax

January 31, 2018

0.1 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.

```
In [1]: 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
```

```
In [2]: def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, num_dev=5
            """
            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,)
```

0.2 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.

```
In [3]: from nndl import Softmax
```

```
In [21]: # 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 see

         np.random.seed(1)

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

         softmax = Softmax(dims=[num_classes, num_features])
```

Softmax loss

```
In [22]: ## Implement the loss function of the softmax using a for loop over
         #  the number of examples

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

```
In [23]: print(loss)
```

```
2.3277607028048966
```

0.3 Question:

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

0.4 Answer:

Since we have generated initial W from a normal distribution, this assigns each data point, on average, to each class equally likely and hence we expect that all class scores and coressponding exponentials on the same order and similar. Hence each term in the loss function can be approximated by:

$$Loss = \frac{1}{m}\sum_{i=1}^{m}\left(\log\sum_{j=1}^{c}\exp^{a_j(x)} - a_{y^{(i)}}(x^{(i)})\right) \approx \frac{1}{m}\sum_{i=1}^{m}\left(\log(c) + a_{y^{(i)}}(x^{(i)}) - a_{y^{(i)}}(x^{(i)})\right) = \ln(c) = \ln(10) \approx 2.3 \quad (1)$$

(2)

Softmax gradient

```
In [12]: ## 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 implem
         softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: 1.217249 analytic: 1.217248, relative error: 4.557511e-08
numerical: 0.046448 analytic: 0.046448, relative error: 2.843581e-07
numerical: 2.457286 analytic: 2.457286, relative error: 4.088560e-09
numerical: 1.909221 analytic: 1.909221, relative error: 3.183455e-08
numerical: -0.701150 analytic: -0.701150, relative error: 6.247204e-08
numerical: 0.376225 analytic: 0.376225, relative error: 9.460772e-08
numerical: 1.464865 analytic: 1.464865, relative error: 1.364491e-08
numerical: 2.896573 analytic: 2.896573, relative error: 2.137011e-08
numerical: 2.125920 analytic: 2.125920, relative error: 2.336915e-09
numerical: 2.291406 analytic: 2.291406, relative error: 1.930837e-08
```

0.5 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.

```
In [13]: import time
```

```
In [14]: ## 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(

         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.li

         # The losses should match but your vectorized implementation should be much faster.
         print('difference in loss / grad: {} /{} '.format(loss - loss_vectorized, np.linalg.no
         # You should notice a speedup with the same output.
```

```
Normal loss / grad_norm: 2.3412243834931 / 354.2632504124683 computed in 0.23025226593017578s
Vectorized loss / grad: 2.3412243834930972 / 354.2632504124684 computed in 0.01799798011779785s
difference in loss / grad: 2.6645352591003757e-15 /2.3977170511309566e-13
```

0.6 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).

0.7 Question:

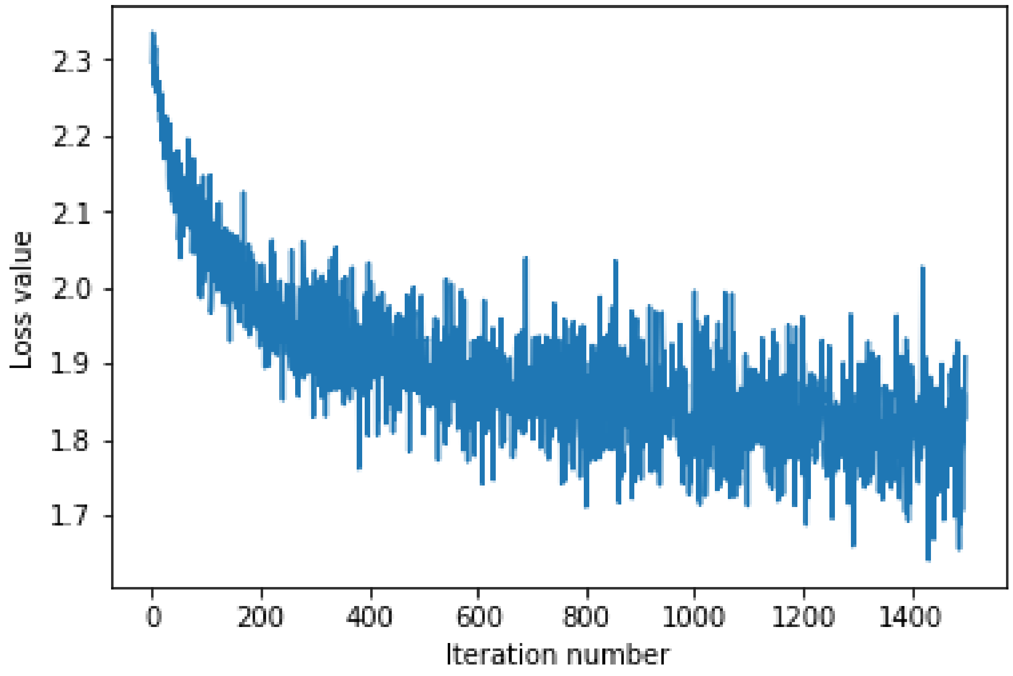
How should the softmax gradient descent training step differ from the svm training step, if at all?

0.8 Answer:

The gradient descent algorithm is not different for SVM and softmax. In fact in both we have the same gradient descenet formulation for updating W matrix which is: next = current - (learning rate) * gradient. The difference though is in finding the gradient since SVM and softmax have different loss functions and hence the stage for calculation gradient of this loss function with respect to W will be different while the general formulation of the gradient descenet is the same.

```
In [15]: # 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 2.296488526280271
iteration 100 / 1500: loss 2.0660917384069237
iteration 200 / 1500: loss 1.984464582884246
iteration 300 / 1500: loss 2.021877859713946
iteration 400 / 1500: loss 1.8812306259936602
iteration 500 / 1500: loss 1.8444950123567299
iteration 600 / 1500: loss 1.8409900570791933
iteration 700 / 1500: loss 1.864696992640151
iteration 800 / 1500: loss 1.710680462299075
iteration 900 / 1500: loss 1.9259869402188243
iteration 1000 / 1500: loss 1.9948055367924393
iteration 1100 / 1500: loss 1.8801523046244102
iteration 1200 / 1500: loss 1.7531563591557147
iteration 1300 / 1500: loss 1.860891600642588
iteration 1400 / 1500: loss 1.8974208454862467
That took 11.110643863677979s
```



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

```
In [16]: ## Implement softmax.predict() and use it to compute the training and testing error.

         y_train_pred = softmax.predict(X_train)
         print(y_train)
         print(y_train_pred)
         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)), ))
```

```
[6 9 9 ... 4 9 3]
[6 1 9 ... 4 1 8]
training accuracy: 0.38185714285714284
```

```
validation accuracy: 0.389
```

0.9 Optimize the softmax classifier

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

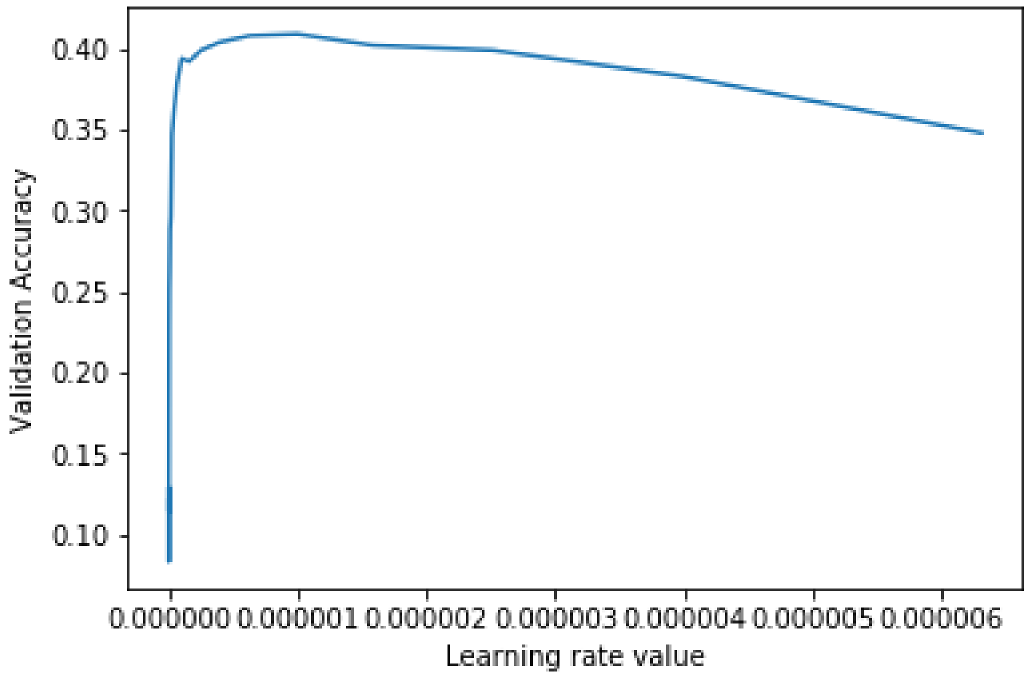
```
In [19]: np.finfo(float).eps
```

```
Out[19]: 2.220446049250313e-16
```

```
In [20]: # ================================================================ #
         # 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.
         # ================================================================ #

         Learning_rate = 10**np.arange(-10,-5,0.2)
         #print(Learning_rate)
         accuracy_test = []

         for Lr in Learning_rate:
             loss_hist = softmax.train(X_train, y_train, Lr, num_iters=1500, verbose=False)
             y_val_pred = softmax.predict(X_val)
             accuracy_test.append(np.mean(np.equal(y_val, y_val_pred)))

         Learning_rate_best_index = np.argmax(accuracy_test)
         Learning_rate_best = Learning_rate[Learning_rate_best_index]

         softmax.train(X_train, y_train, Learning_rate_best, num_iters=1500, verbose=False)

         y_est_test = softmax.predict(X_test)
         accuracy_test_f = np.mean(np.equal(y_test, y_est_test))

         best_valication_accuracy =  1 - accuracy_test[Learning_rate_best_index]
         error_test = 1 - accuracy_test_f

         plt.figure()
         plt.plot(Learning_rate, accuracy_test)
         plt.xlabel('Learning rate value')
         plt.ylabel('Validation Accuracy')

         print('Best Learning Rate: ', Learning_rate_best)
```

```
        print('Best Validation Accuracy: ', accuracy_test[Learning_rate_best_index])
        print('Best Validation Error: ', best_valication_accuracy)
        print('Test Accuracy: ', accuracy_test_f)
        print('Test Error: ', error_test)


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

```
Best Learning Rate:  9.999999999999673e-07
Best Validation Accuracy:  0.409
Best Validation Error:  0.591
Test Accuracy:  0.399
Test Error:  0.601
```



svm

January 31, 2018

0.1 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.

0.2 Importing libraries and data setup

```
In [288]: 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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

```
In [289]: # 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,)
```

```
In [290]: # 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', 't
          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()
```

```
In [291]: # 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)
```

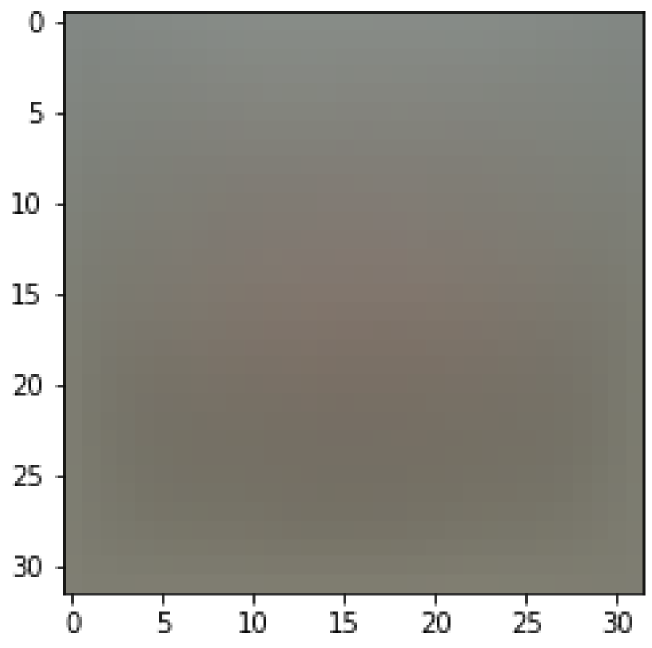
```
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,)
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
Test labels shape:  (1000,)
Dev data shape:  (500, 32, 32, 3)
Dev labels shape:  (500,)
```

```
In [292]: # 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)
```

```
In [293]: # 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]
```



```
In [294]: # 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
```

```
In [295]: # 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)
```

0.3 Question:

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

0.4 Answer:

(1) In KNN we did not subtract the mean from the data, or in other words we did not 'center' the data, since it works based on similarity definition using any of the norms (L1,L2,..). These norms are applied which work on vectors connecting data points in vector spaces. No matter whether we center our data or not, the distances (obtained from norms) in the vector space does not change. Hence, the result of the KNN will not be affected. At high level, centering the data does not change the distance-based similarity used for knn so we did not need to.

For SVM, on the other hand, centering the data will affect the results obtained for W matrix. We need to actually center the data since data belonging to different dimensions are dissimilar and SVM would favor the inputs that are "larger" and more far away from maximum-margin hyperplanes. In fact, there is not direct notion of relative distance of feature points, but rather distance to "optimal" hyperplaces are important.

0.5 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.

```
In [296]: from nndl.svm import SVM
In [297]: # 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 

          np.random.seed(1)

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

          svm = SVM(dims=[num_classes, num_features])
          print(svm.W.shape)
```

```
(10, 3073)
```

SVM loss

```
In [298]: ## 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
```

```
28739.84972087688
49000
The training set loss is 15569.977915410242.
```

SVM gradient

```
In [299]: ## 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)
          # Compare your gradient to a numerical gradient check.
          # You should see relative gradient errors on the order of 1e-07 or less if you imple
          svm.grad_check_sparse(X_dev, y_dev, grad)
```

```
48567.07743923132
500
48567.07829412234
500
numerical: -3.689838 analytic: -3.689838, relative error: 6.087010e-08
48567.077866676824
500
```

```
48567.077866676824
500
numerical: 3.165714 analytic: 3.165714, relative error: 1.905350e-08
48567.077584946215
500
48567.07814840744
500
numerical: -9.331309 analytic: -9.331309, relative error: 8.489963e-09
48567.077866676824
500
48567.077866676824
500
numerical: 11.571089 analytic: 11.571089, relative error: 1.882846e-08
48567.077581836624
500
48567.07815151703
500
numerical: -1.153990 analytic: -1.153991, relative error: 3.275240e-07
48567.07721458295
500
48567.0785187707
500
numerical: -4.315784 analytic: -4.315784, relative error: 7.170807e-09
48567.078503908255
500
48567.0772294454
500
numerical: -3.670100 analytic: -3.670099, relative error: 8.712966e-08
48567.07765864887
500
48567.07807470479
500
numerical: -12.606532 analytic: -12.606531, relative error: 1.200316e-08
48567.07736552907
500
48567.07836782458
500
numerical: -3.790232 analytic: -3.790232, relative error: 1.973881e-08
48567.08325428009
500
48567.072479073555
500
numerical: -14.428070 analytic: -14.428070, relative error: 6.323635e-09
```

0.6 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.

```
In [300]: import time
```

```
In [301]: ## 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

          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.li

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

          # You should notice a speedup with the same output, i.e., differences on the order o
```

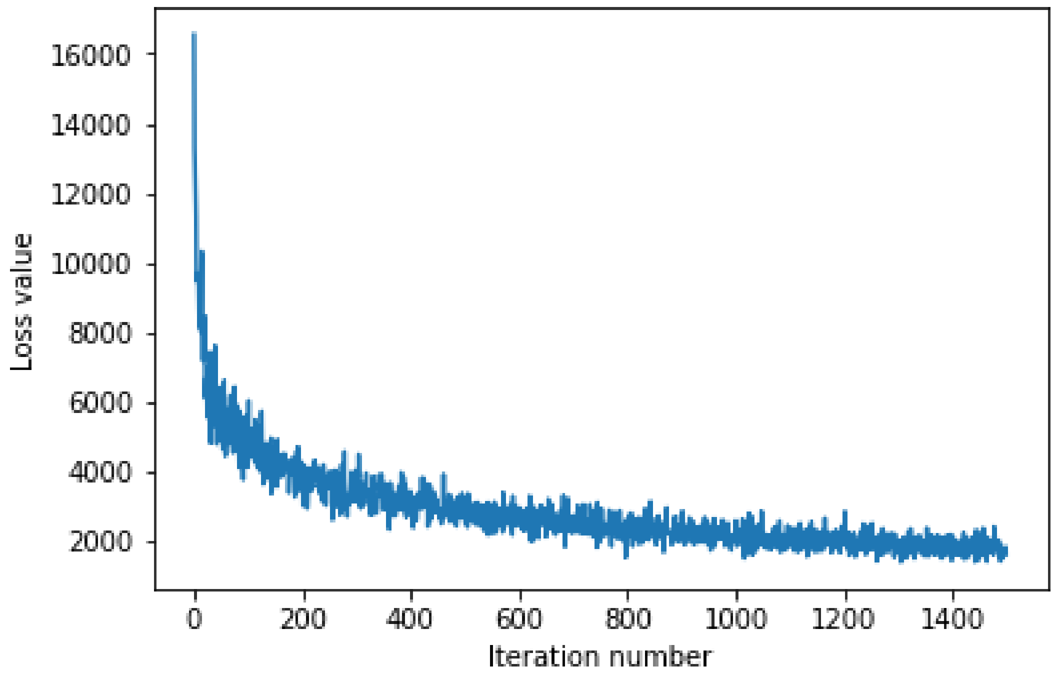
```
Normal loss / grad_norm: 15966.609801313683 / 2149.29158910058 computed in 0.10100221633911133
Vectorized loss / grad: 15966.609801313687 / 2149.2915891005805 computed in 0.00700092315673828
difference in loss / grad: -3.637978807091713e-12 / 7.23372434858787e-12
```

0.7 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).

```
In [302]: # 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 4017.3331379427877
iteration 300 / 1500: loss 3681.9226471953625
iteration 400 / 1500: loss 2732.6164373988995
iteration 500 / 1500: loss 2786.6378424645054
iteration 600 / 1500: loss 2837.0357842782673
iteration 700 / 1500: loss 2206.2348687399326
iteration 800 / 1500: loss 2269.03882411698
iteration 900 / 1500: loss 2543.23781538592
iteration 1000 / 1500: loss 2566.6921357268257
iteration 1100 / 1500: loss 2182.068905905164
iteration 1200 / 1500: loss 1861.118224425045
iteration 1300 / 1500: loss 1982.9013858528256
iteration 1400 / 1500: loss 1927.5204158582114
That took 8.729506254196167s
```



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

```
In [303]: ## 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)), ))
          y_test_pred = svm.predict(X_test)
          print('test accuracy: {}'.format(np.mean(np.equal(y_test, y_test_pred)), ))


          print('training error: {}'.format(1-np.mean(np.equal(y_train,y_train_pred), )))
          print('validation errror: {}'.format(1-np.mean(np.equal(y_val, y_val_pred)), ))
          print('test errror: {}'.format(1-np.mean(np.equal(y_test, y_test_pred)), ))
```

```
training accuracy: 0.28530612244897957
validation accuracy: 0.3
test accuracy: 0.248
training error: 0.7146938775510204
validation errror: 0.7
test errror: 0.752
```

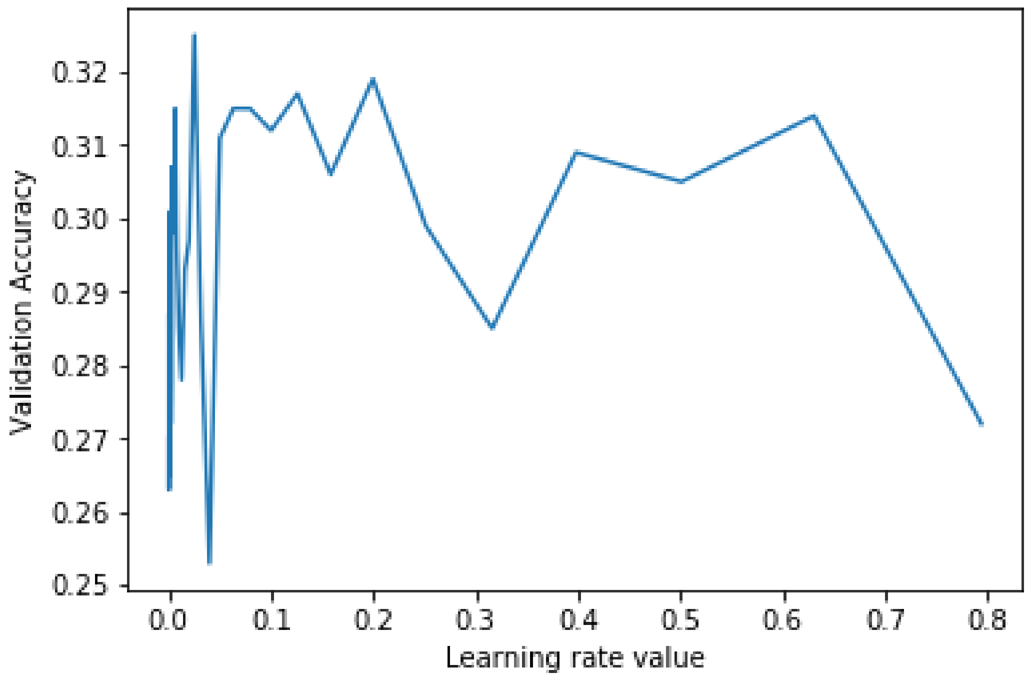
0.8 Optimize the SVM

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).

```
In [304]: # ================================================================ #
          # 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
          # ================================================================ #


          Learning_rate = 10**np.arange(-4,0,0.1)
          #print(Learning_rate)
          accuracy_test = []

          for Lr in Learning_rate:
              loss_hist = svm.train(X_train, y_train, Lr, num_iters=1500, verbose=False)
              y_val_pred = svm.predict(X_val)
              accuracy_test.append(np.mean(np.equal(y_val, y_val_pred)))

          Learning_rate_best_index = np.argmax(accuracy_test)
          Learning_rate_best = Learning_rate[Learning_rate_best_index]
```

```
        svm.train(X_train, y_train, Learning_rate_best, num_iters=1500, verbose=False)

        y_est_test = svm.predict(X_test)
        accuracy_test_f = np.mean(np.equal(y_test, y_est_test))

        best_valication_accuracy =  1 - accuracy_test[Learning_rate_best_index]
        error_test = 1 - accuracy_test_f

        plt.figure()
        plt.plot(Learning_rate, accuracy_test)
        plt.xlabel('Learning rate value')
        plt.ylabel('Validation Accuracy')

        print('Best Learning Rate: ', Learning_rate_best)
        print('Best Validation Accuracy: ', accuracy_test[Learning_rate_best_index])
        print('Best Validation Error: ', best_valication_accuracy)
        print('Test Accuracy: ', accuracy_test_f)
        print('Test Error: ', error_test)

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

```
Best Learning Rate:  0.025118864315095923
Best Validation Accuracy:  0.325
Best Validation Error:  0.675
Test Accuracy:  0.306
Test Error:  0.694
```


Validation Accuracy
0.25
0.26
0.27
0.28
0.29
0.30
0.31
0.32
Learning rate value
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8

```
import numpy as np
import pdb

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


    for i in np.arange(num_train):
        lost = 0.0
    # ================================================================ #
    # 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.)
    # ================================================================ #
        y_est = np.dot(self.W,X[i,:].T)
        class_est = y_est[y[i]]
        #print(y_est)
        #print(class_est)
        #print(y[i])
        for j in range(num_classes):
            if j != y[i]:
                #print(class_est)
                #print(y_est[j])
                lost += np.maximum(0,1-class_est+y_est[j])
        loss += lost

    pass


    #print(lost)
    loss /= num_train
    print(lost)
    print(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)        # for whole data
  grad_tmp = np.zeros_like(self.W)    # for each data

  for i in np.arange(num_train):
  # ================================================================ #
  # YOUR CODE HERE:
  #   Calculate the SVM loss and the gradient.  Store the gradient in
  #   the variable grad.
  # ================================================================ #
      lost = 0.0
      y_est = np.dot(self.W,X[i,:].T)
      class_est = y_est[y[i]]

      #print(y_est)
      #print(class_est)
      #print(y[i])

      #if i==0:
      grada = np.zeros(X.shape[1])
  #lost: ith training data contribution to cost function
      for j in range(num_classes):
          if j != y[i]:
              #print(class_est)
              #print(y_est[j])
              lost += np.maximum(0,1-class_est+y_est[j])
              if (1-class_est+y_est[j] > 0):                    #
                  grad_tmp[j,:] = X[i,:].T                      #
                  grada -= X[i,:].T                             #  in this block we update
              else:                                             #
                  grad_tmp[j,:] = np.zeros(X.shape[1])          #

      grad_tmp[y[i],:] = grada                                  #  in this block we update

      grad += grad_tmp
      loss += lost




  loss /= num_train
  grad /= num_train

  #print(grad[0,:])
  pass

  # ================================================================ #
  # 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_analyt
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic,

  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(self.W, X.T)

    bias_by_one = np.ones(scores.shape) # adding bias 1 in vector form

    training_scores = np.ones(scores.shape) * [scores[y, np.arange(0, scores.shape[1])]]

    Loss = scores + bias_by_one - training_scores
    Loss_mod = Loss
    # performing max function against zero in vector form
    # also we should not count the remaining that are equal to 1
    Loss_mod[Loss_mod < 0] = 0
    Loss_mod[y, np.arange(0, scores.shape[1])] = 0 # not counting elements that are equal to 1
    loss = np.sum(Loss_mod)

    # Averaging over number of training data
    num_train = X.shape[0]
    loss /= num_train


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



    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the SVM grad WITHOUT any for loops.
    # ================================================================ #

    Loss_mod2 = Loss

    Loss_mod2[Loss_mod2 < 0] = 0    # we don't care about the ones that have negative margins
    Loss_mod2[Loss_mod2 > 0] = 1    # the positive margines contribute to the loss with x then
    Loss_mod2[y, np.arange(0, scores.shape[1])] = 0 # we take care of these rows corresponding

    Loss_mod2[y, np.arange(0, scores.shape[1])] = -1 * np.sum(Loss_mod2, axis=0) # rows corres
    grad = np.dot(Loss_mod2, X)

    # Averaging over number of training data
    num_train = X.shape[0]
    grad /= num_train



    # ================================================================ #
    # 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.
      # ================================================================ #

      mask = np.random.choice(num_train, batch_size, replace=True)

      X_batch = X[mask]                          # (dim, batch_size)
      y_batch = y[mask]                          # (batch_size,)

      # ================================================================ #
      # 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 = 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.
  # ================================================================ #

  y_pred = np.argmax(self.W.dot(X.T), axis=0)

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

  return y_pred
```