

ECE C147, Winter 2020
Homework Guy Ohayon
Submitted by Guy Ohayon

1 KNN

This is the k-nearest neighbors workbook for ECE C147/C247 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 jupyter 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

```
In [2]: 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
```

```
In [11]: # Set the path to the CIFAR-10 data
cifar10_dir = './cifar-10-batches-py' # You need to update this line
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 [12]: # 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()
```



```
In [13]: # 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.

```
In [14]: # Import the KNN class
```

```
from nndl import KNN
```

```
In [15]: # 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) KNN is a nonparametric classifier. Thus, knn.train() simply saves X_train and y_train in memory (no training is necessary in KNN).

(2) Pros: this training step takes O(1) time. Cons: the cost of a nonparametric model such as KNN is that future predictions are slow, especially when the training set is big. Each prediction is dependant on the size of the training set, and so the bigger the training set, the slower the prediction.

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 [16]: # 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: 33.776381731033325
 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.

```
In [17]: # 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: 0.7717816829681396
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.

```
In [21]: # 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_pred = knn.predict_labels(dists_L2_vectorized, k=1)
error = 1 - float(np.sum(y_pred == y_test))/num_test
# ===== #
# 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.

```
In [47]: # Create the dataset folds for cross-validation.  
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]  
# ===== #  
X_train_folds = np.split(X_train, num_folds)  
y_train_folds = np.split(y_train, num_folds)  
  
# ===== #  
# END YOUR CODE HERE  
# ===== #
```

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 [76]: 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.
# ===== #

num_test_fold = float(num_training)/num_folds
errors = []
for k in ks:
    error = 0
    for i in range(num_folds):
        knn = KNN()

        X_training_fold = np.array([y for x in X_train_folds[:i] + X_train_folds[i+1:] for y in x])
        X_test_fold = np.array([y for x in X_train_folds[i:i+1] for y in x])

        y_training_fold = np.array([y for x in y_train_folds[:i] + y_train_folds[i+1:] for y in x])
        y_test_fold = np.array([y for x in y_train_folds[i:i+1] for y in x])

        knn.train(X_training_fold, y_training_fold)
        test_fold_L2_distances = knn.compute_L2_distances_vectorized(X_test_fold)

        y_pred = knn.predict_labels(test_fold_L2_distances, k)
        error += (1 - float(np.sum(y_pred == y_test_fold))/num_test_fold)

    errors.append(error/num_folds)

print(errors)
plt.plot(ks, errors, '-o')
plt.xlabel('k')
plt.ylabel('cross-validation error')
plt.show

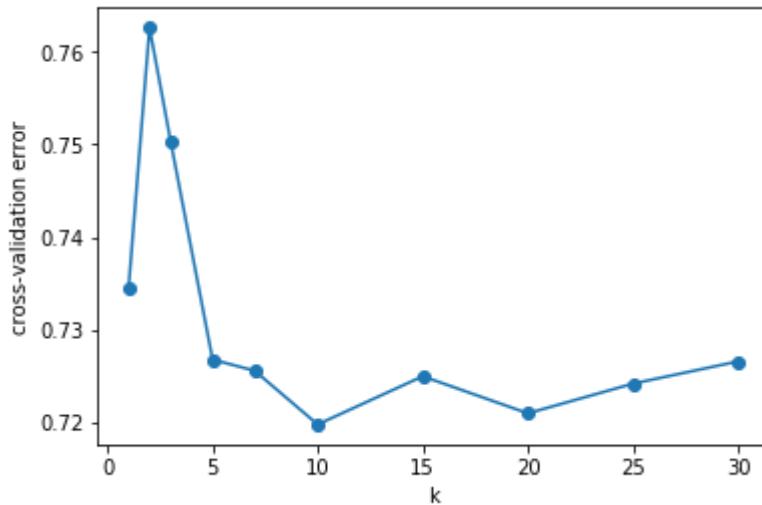
# ===== #
# END YOUR CODE HERE
# ===== #

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

```
[0.7344, 0.7626000000000002, 0.7504000000000001, 0.7267999999999999,
```

```
0.7256, 0.7198, 0.725, 0.7210000000000001, 0.7242, 0.7266]
```

```
Computation time: 83.62
```



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) The best value is $k = 10$.
- (2) The cross-validation error for $k = 10$ is 0.7198.

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 [79]: 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- nor
# ms
# to speed this up, but it is not necessary.
# ===== #
errors = []
for norm in norms:
    error = 0
    for i in range(num_folds):
        knn = KNN()

        X_training_fold = np.array([y for x in X_train_folds[:i] + X_t
rain_folds[i+1:] for y in x])
        X_test_fold = np.array([y for x in X_train_folds[i:i+1] for y
in x])

        y_training_fold = np.array([y for x in y_train_folds[:i] + y_t
rain_folds[i+1:] for y in x])
        y_test_fold = np.array([y for x in y_train_folds[i:i+1] for y
in x])

        knn.train(X_training_fold, y_training_fold)
        test_fold_L2_distances = knn.compute_distances(X_test_fold, no
rm)

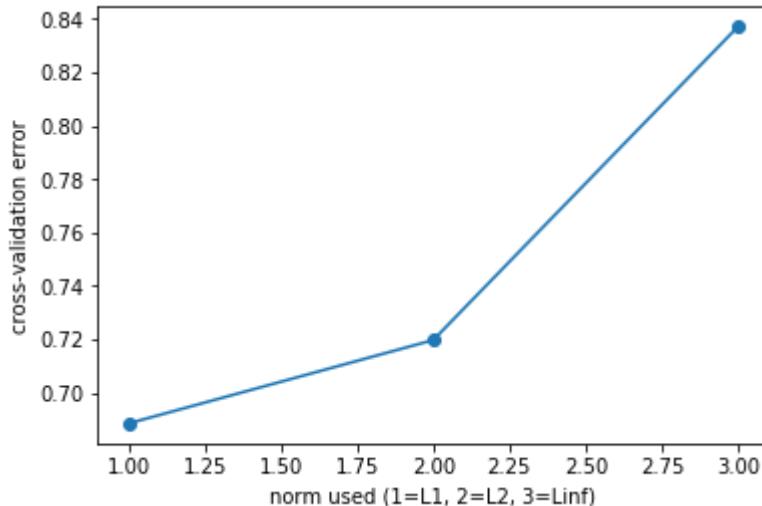
        y_pred = knn.predict_labels(test_fold_L2_distances, 10)
        error += (1 - float(np.sum(y_pred == y_test_fold))/num_test_fo
ld)

    errors.append(error/num_folds)

print(errors)
plt.plot([1, 2, 3], errors, '-o')
plt.xlabel('norm used (1=L1, 2=L2, 3=Linf)')
plt.ylabel('cross-validation error')
plt.show
# ===== #
# END YOUR CODE HERE
```

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

[0.6885999999999999, 0.7198, 0.8370000000000001]
Computation time: 751.23



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) L_1 norm.
- (2) The cross-validation error for L_1 norm and $k = 10$ is 0.6886.

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 [83]: 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.train(X_train, y_train)
y_pred = knn.predict_labels(knn.compute_distances(X_test, L1_norm), k=10)
error = 1 - float(np.sum(y_pred == y_test))/num_test

# ===== #
# 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:

When we chose $k = 1$ and used the L_2 norm, the error was 0.726. This time, by using $k = 10$ and L_1 norm, the error was 0.722. Thus, the error improved by 0.004, which means that now 72.2% of the test examples are classified incorrectly.

knn.py

```

1  import numpy as np
2  import pdb
3
4 """
5 This code was based off of code from cs231n at Stanford University, and modified for ECE C147/C247 at UCLA.
6 """
7
8 class KNN(object):
9
10    def __init__(self):
11        pass
12
13    def train(self, X, y):
14        """
15            Inputs:
16            - X is a numpy array of size (num_examples, D)
17            - y is a numpy array of size (num_examples, )
18        """
19        self.X_train = X
20        self.y_train = y
21
22    def compute_distances(self, X, norm=None):
23        """
24            Compute the distance between each test point in X and each training point
25            in self.X_train.
26
27            Inputs:
28            - X: A numpy array of shape (num_test, D) containing test data.
29            - norm: the function with which the norm is taken.
30
31            Returns:
32            - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
33            is the Euclidean distance between the ith test point and the jth training
34            point.
35        """
36        if norm is None:
37            norm = lambda x: np.sqrt(np.sum(x**2))
38            #norm = 2
39
40        num_test = X.shape[0]
41        num_train = self.X_train.shape[0]
42        dists = np.zeros((num_test, num_train))
43        for i in np.arange(num_test):
44
45            for j in np.arange(num_train):
46                # ===== #
47                # YOUR CODE HERE:
48                #   Compute the distance between the ith test point and the jth
49                #   training point using norm(), and store the result in dists[i, j].
50                # ===== #
51
52            dists[i][j] = norm((X[i]-self.X_train[j]))
53
54            # ===== #
55            # END YOUR CODE HERE
56            # ===== #
57
58        return dists
59
60    def compute_L2_distances_vectorized(self, X):
61        """
62            Compute the distance between each test point in X and each training point
63            in self.X_train WITHOUT using any for loops.
64
65            Inputs:
66            - X: A numpy array of shape (num_test, D) containing test data.
67
68            Returns:
69            - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
70            is the Euclidean distance between the ith test point and the jth training
71            point.
72        """
73        num_test = X.shape[0]
74        num_train = self.X_train.shape[0]
75        dists = np.zeros((num_test, num_train))

```

```

76
77 # ===== #
78 # YOUR CODE HERE:
79 # Compute the L2 distance between the ith test point and the jth
80 # training point and store the result in dists[i, j]. You may
81 # NOT use a for loop (or list comprehension). You may only use
82 # numpy operations.
83 #
84 # HINT: use broadcasting. If you have a shape (N,1) array and
85 # a shape (M,) array, adding them together produces a shape (N, M)
86 # array.
87 # ===== #
88
89 X_sq = np.sum(X**2, axis=1).reshape(num_test,1)
90 X_train_sq = np.sum(self.X_train**2, axis=1).reshape(1,num_train)
91 dists = np.sqrt(X_sq + X_train_sq - 2 * X.dot(self.X_train.T))
92
93 # ===== #
94 # END YOUR CODE HERE
95 # ===== #
96
97 return dists
98
99
100 def predict_labels(self, dists, k=1):
101 """
102     Given a matrix of distances between test points and training points,
103     predict a label for each test point.
104
105     Inputs:
106     - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
107       gives the distance between the ith test point and the jth training point.
108
109     Returns:
110     - y: A numpy array of shape (num_test,) containing predicted labels for the
111       test data, where y[i] is the predicted label for the test point X[i].
112 """
113 num_test = dists.shape[0]
114 y_pred = np.zeros(num_test)
115 for i in np.arange(num_test):
116     # A list of length k storing the labels of the k nearest neighbors to
117     # the ith test point.
118     closest_y = []
119     # ===== #
120     # YOUR CODE HERE:
121     # Use the distances to calculate and then store the labels of
122     # the k-nearest neighbors to the ith test point. The function
123     # numpy.argsort may be useful.
124     #
125     # After doing this, find the most common label of the k-nearest
126     # neighbors. Store the predicted label of the ith training example
127     # as y_pred[i]. Break ties by choosing the smaller label.
128     # ===== #
129
130     knn_points = np.argsort(dists[i])[:k]
131     knn_labels = self.y_train[knn_points]
132     y_pred[i] = np.argmax(np.bincount(knn_labels))
133
134     # ===== #
135     # END YOUR CODE HERE
136     # ===== #
137
138 return y_pred
139

```

2 SVM

This is the svm workbook for ECE C147/C247 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

```
In [1]: 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
```

```
In [2]: # Set the path to the CIFAR-10 data
cifar10_dir = './cifar-10-batches-py' # You need to update this line
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 [3]: # 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()
```



```
In [4]: # 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 origina
l
# 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,)
```

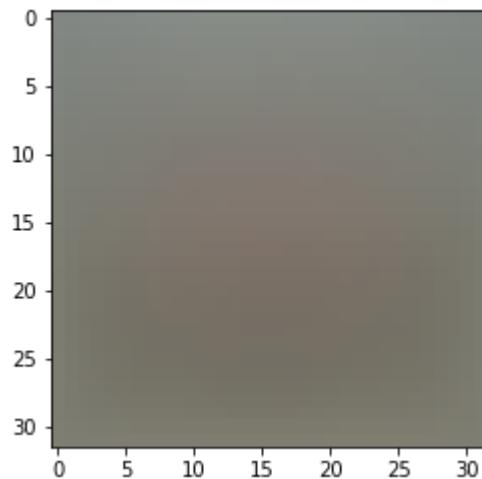
```
In [5]: # Preprocessing: reshape the image data into rows
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_val = np.reshape(X_val, (X_val.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
X_dev = np.reshape(X_dev, (X_dev.shape[0], -1))

# 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 [6]: # 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 [7]: # 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 [8]: # 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) The training step of a Support Vector Machine is $W = W - \epsilon \cdot \nabla_W Loss$, where $\nabla_W Loss$ relies linearly on the features of the examples in the training set (except when the loss is 0 occasionally). For that reason, SVM training relies heavily on the magnitude of features, especially because all features are multiplied by the same learning rate. Thus, if the features are of large magnitude, the corresponding weights will also possibly be of high magnitude. Therefore, features of high magnitude will dominate the score of an example, preventing other low-magnitude features to possibly give significant information about the score/classification of the example. Therefore, subtracting the mean of the dataset can help avoid this possible domination of high-magnitude features.

In KNN there's no significance to the feature scale because Euclidean depends merely on the distance between vectors. Therefore, mean subtraction is not necessary for KNN.

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 [9]: from nnndl.svm import SVM
```

```
In [10]: # 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

```
In [11]: ## 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

```
In [12]: ## 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 implemented the gradient correctly.  
svm.grad_check_sparse(X_dev, y_dev, grad)  
  
numerical: 0.167121 analytic: 0.167121, relative error: 8.579596e-07  
numerical: -2.101728 analytic: -2.101729, relative error: 1.953189e-07  
numerical: 2.748654 analytic: 2.748655, relative error: 8.216914e-08  
numerical: 7.120861 analytic: 7.120860, relative error: 5.563664e-08  
numerical: -0.037178 analytic: -0.037178, relative error: 4.921029e-06  
numerical: 11.391076 analytic: 11.391076, relative error: 1.297940e-08  
numerical: 6.793050 analytic: 6.793050, relative error: 4.358543e-08  
numerical: -16.868199 analytic: -16.868199, relative error: 9.581999e-11  
numerical: -12.957866 analytic: -12.957865, relative error: 1.211635e-08  
numerical: -15.639322 analytic: -15.639322, relative error: 1.755329e-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.

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

```
In [14]: ## 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_ve
ctorized, 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_vectoriz
ed, np.linalg.norm(grad - grad_vectorized)))

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

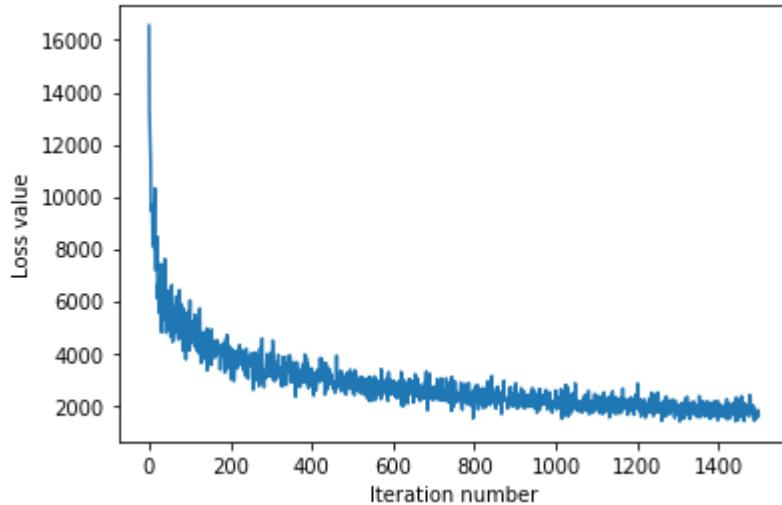
```
Normal loss / grad_norm: 15057.810738219052 / 2190.3996534895227 compu
ted in 0.07339715957641602s
Vectorized loss / grad: 15057.810738219017 / 2190.399653489523 compute
d in 0.004544496536254883s
difference in loss / grad: 3.456079866737127e-11 / 3.0849259899516946e
-12
```

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).

```
In [15]: # 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 4701.089451272713  
iteration 200 / 1500: loss 4017.333137942789  
iteration 300 / 1500: loss 3681.922647195362  
iteration 400 / 1500: loss 2732.6164373988995  
iteration 500 / 1500: loss 2786.6378424645054  
iteration 600 / 1500: loss 2837.035784278267  
iteration 700 / 1500: loss 2206.2348687399326  
iteration 800 / 1500: loss 2269.03882411698  
iteration 900 / 1500: loss 2543.2378153859195  
iteration 1000 / 1500: loss 2566.6921357268266  
iteration 1100 / 1500: loss 2182.068905905164  
iteration 1200 / 1500: loss 1861.1182244250451  
iteration 1300 / 1500: loss 1982.9013858528258  
iteration 1400 / 1500: loss 1927.5204158582117  
That took 5.560702562332153s
```



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

```
In [16]: ## 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_p_
red)), ))
```

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}).

```
In [17]: # ===== #
# 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 VALIDAT
ION 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
# ===== #
best_learning_rate = 0
best_val_accur = 0
best_test_accur = 0
for i in range(1, 10000, 200):
    learning_rate = i*1e-5
    svm.train(X_train, y_train, learning_rate=learning_rate,
              num_iters=500, verbose=False)
    y_train_pred = svm.predict(X_train)
    train_accur = np.mean(np.equal(y_train,y_train_pred))
    y_val_pred = svm.predict(X_val)
    val_accur = np.mean(np.equal(y_val, y_val_pred))

    if val_accur > best_val_accur:
        best_val_accur = val_accur
        best_learning_rate = learning_rate

    svm.train(X_train, y_train, learning_rate=best_learning_rate,
              num_iters=500, verbose=False)

    y_test_pred = svm.predict(X_test)
    test_error = 1 - np.mean(np.equal(y_test, y_test_pred))

    print('best validation accuracy: {}'.format(best_val_accur))
    print('best learning rate: {}'.format(best_learning_rate))
    print('test set error using the best learning rate: {}'.format(test_er
ror))
# ===== #
# END YOUR CODE HERE
# ===== #
```

```
best validation accuracy: 0.341
best learning rate: 0.02001
test set error using the best learning rate: 0.7110000000000001
```

svm.py

```

1  import numpy as np
2  import pdb
3
4 """
5 This code was based off of code from cs231n at Stanford University, and modified for ECE C147/C247 at UCLA.
6 """
7 class SVM(object):
8
9     def __init__(self, dims=[10, 3073]):
10         self.init_weights(dims=dims)
11
12     def init_weights(self, dims):
13         """
14             Initializes the weight matrix of the SVM. Note that it has shape (C, D)
15             where C is the number of classes and D is the feature size.
16         """
17         self.W = np.random.normal(size=dims)
18
19     def loss(self, X, y):
20         """
21             Calculates the SVM loss.
22
23             Inputs have dimension D, there are C classes, and we operate on minibatches
24             of N examples.
25
26             Inputs:
27             - X: A numpy array of shape (N, D) containing a minibatch of data.
28             - y: A numpy array of shape (N,) containing training labels; y[i] = c means
29                 that X[i] has label c, where 0 <= c < C.
30
31             Returns a tuple of:
32             - loss as single float
33         """
34
35     # compute the loss and the gradient
36     num_classes = self.W.shape[0]
37     num_train = X.shape[0]
38     loss = 0.0
39
40     for i in np.arange(num_train):
41         # ===== #
42         # YOUR CODE HERE:
43         # Calculate the normalized SVM loss, and store it as 'loss'.
44         # (That is, calculate the sum of the losses of all the training
45         # set margins, and then normalize the loss by the number of
46         # training examples.)
47         # ===== #
48         curr_label = y[i]
49         for j in range(num_classes):
50             if j != curr_label:
51                 loss += max(0, 1 + self.W[j].dot(X[i]) - self.W[curr_label].dot(X[i]))
52
53     loss /= num_train
54
55     # ===== #
56     # END YOUR CODE HERE
57     # ===== #
58
59     return loss
60
61     def loss_and_grad(self, X, y):
62         """
63             Same as self.loss(X, y), except that it also returns the gradient.
64
65             Output: grad -- a matrix of the same dimensions as W containing
66                 the gradient of the loss with respect to W.
67         """
68
69         # compute the loss and the gradient
70         num_classes = self.W.shape[0]
71         num_train = X.shape[0]
72         loss = 0.0
73         grad = np.zeros_like(self.W)
74
75         for i in np.arange(num_train):

```

```

76      # ===== #
77      # YOUR CODE HERE:
78      # Calculate the SVM loss and the gradient. Store the gradient in
79      # the variable grad.
80      # ===== #
81      curr_label = y[i]
82      for j in range(num_classes):
83          if j != curr_label:
84              a_j = self.W[j].dot(X[i])
85              a_y_i = self.W[curr_label].dot(X[i])
86              hinge_loss = max(0, 1 + a_j - a_y_i)
87              loss += hinge_loss
88          if hinge_loss > 0:
89              grad[curr_label] -= X[i]
90              grad[j] += X[i]
91
92      grad /= num_train
93      loss /= num_train
94
95      # ===== #
96      # END YOUR CODE HERE
97      # ===== #
98
99      return loss, grad
100
101 def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
102     """
103     sample a few random elements and only return numerical
104     in these dimensions.
105     """
106
107     for i in np.arange(num_checks):
108         ix = tuple([np.random.randint(m) for m in self.W.shape])
109
110         oldval = self.W[ix]
111         self.W[ix] = oldval + h # increment by h
112         fxph = self.loss(X, y)
113         self.W[ix] = oldval - h # decrement by h
114         fxmh = self.loss(X,y) # evaluate f(x - h)
115         self.W[ix] = oldval # reset
116
117         grad_numerical = (fxph - fxmh) / (2 * h)
118         grad_analytic = your_grad[ix]
119         rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) + abs(grad_analytic))
120         print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic, rel_error))
121
122 def fast_loss_and_grad(self, X, y):
123     """
124     A vectorized implementation of loss_and_grad. It shares the same
125     inputs and ouptuts as loss_and_grad.
126     """
127     loss = 0.0
128     grad = np.zeros(self.W.shape) # initialize the gradient as zero
129
130     # ===== #
131     # YOUR CODE HERE:
132     # Calculate the SVM loss WITHOUT any for loops.
133     # ===== #
134     num_classes = self.W.shape[0]
135     num_train = X.shape[0]
136     hinge = self.W.dot(X.T).T # Dim: (num_train, num_classes)
137     hinge_labeled = np.expand_dims(hinge[np.arange(hinge.shape[0]), y], axis=1) # Dims: (num_train, 1)
138     hinge = 1 + hinge - hinge_labeled # Dims: (num_train, num_classes)
139     hinge[np.arange(hinge.shape[0]), y] = 0
140     zeros = np.zeros(hinge.shape)
141     hinge = np.stack((hinge, zeros))
142     hinge = np.amax(hinge, axis=0)
143     loss = (1/float(num_train)) * np.sum(np.sum(hinge, axis=1), axis=0)
144     # ===== #
145     # END YOUR CODE HERE
146     # ===== #
147
148
149
150     # ===== #
151     # YOUR CODE HERE:
152     # Calculate the SVM grad WITHOUT any for loops.
153     # ===== #

```

```

154     indicators = hinge
155     indicators[hinge > 0] = 1
156     row_sum = np.sum(indicators, axis=1)
157     indicators[np.arange(num_train), y] = -row_sum.T
158     grad = X.T.dot(indicators).T
159     grad /= float(num_train)
160     # ===== #
161     # END YOUR CODE HERE
162     # ===== #
163
164     return loss, grad
165
166 def train(self, X, y, learning_rate=1e-3, num_iters=100,
167           batch_size=200, verbose=False):
168     """
169     Train this linear classifier using stochastic gradient descent.
170
171     Inputs:
172     - X: A numpy array of shape (N, D) containing training data; there are N
173       training samples each of dimension D.
174     - y: A numpy array of shape (N,) containing training labels; y[i] = c
175       means that X[i] has label 0 <= c < C for C classes.
176     - learning_rate: (float) learning rate for optimization.
177     - num_iters: (integer) number of steps to take when optimizing
178     - batch_size: (integer) number of training examples to use at each step.
179     - verbose: (boolean) If true, print progress during optimization.
180
181     Outputs:
182     A list containing the value of the loss function at each training iteration.
183     """
184     num_train, dim = X.shape
185     num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of classes
186
187     self.init_weights(dims=[np.max(y) + 1, X.shape[1]])    # initializes the weights of self.W
188
189     # Run stochastic gradient descent to optimize W
190     loss_history = []
191
192     for it in np.arange(num_iters):
193         X_batch = None
194         y_batch = None
195
196         # ===== #
197         # YOUR CODE HERE:
198         #   Sample batch_size elements from the training data for use in
199         #   gradient descent. After sampling,
200         #   - X_batch should have shape: (dim, batch_size)
201         #   - y_batch should have shape: (batch_size,)
202         #   The indices should be randomly generated to reduce correlations
203         #   in the dataset. Use np.random.choice. It's okay to sample with
204         #   replacement.
205         # ===== #
206         idx = np.random.randint(low=0, high=X.shape[0], size=batch_size)
207         X_batch = X[idx]
208         y_batch = y[idx]
209         # ===== #
210         # END YOUR CODE HERE
211         # ===== #
212
213         # evaluate loss and gradient
214         loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
215         loss_history.append(loss)
216
217         # ===== #
218         # YOUR CODE HERE:
219         #   Update the parameters, self.W, with a gradient step
220         # ===== #
221         self.W = self.W - learning_rate * grad
222         # ===== #
223         # END YOUR CODE HERE
224         # ===== #
225
226         if verbose and it % 100 == 0:
227             print('iteration {} / {}: loss {}'.format(it, num_iters, loss))
228
229     return loss_history
230
231 def predict(self, X):

```

```
232     """
233     Inputs:
234     - X: N x D array of training data. Each row is a D-dimensional point.
235
236     Returns:
237     - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
238     array of length N, and each element is an integer giving the predicted
239     class.
240     """
241     y_pred = np.zeros(X.shape[1])
242
243
244     # ===== #
245     # YOUR CODE HERE:
246     #   Predict the labels given the training data with the parameter self.W.
247     # ===== #
248     y_pred = np.argmax(self.W.dot(X.T), axis=0)
249     # ===== #
250     # END YOUR CODE HERE
251     # ===== #
252
253     return y_pred
254
255
```

3 Softmax classifier

This is the softmax workbook for ECE C147/C247 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 jupyter 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=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' # You need to update this line
    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.

In [3]: `from nnndl import Softmax`

In [4]: *# 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])

```

Softmax loss

In [5]: *## Implement the loss function of the softmax using a for loop over
the number of examples*

```

loss = softmax.loss(X_train, y_train)

```

In [6]: `print(loss)`

2.3277607028048757

Question:

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

Answer:

The results tells us that on average, given our model, the log of the probability that an input example $x^{(i)}$ is classified to the class $y^{(i)}$, is approximately -2.32 . Therefore, the probability that an input example $x^{(i)}$ is classified to the class $y^{(i)}$ is approximately $e^{-2.32} = 0.1$. This makes sense, because our model's parameters is randomly generated, so we expect that on average all classes are equally likely (we have 10 classes, each with probability 0.1 on average).

Softmax gradient

```
In [7]: ## 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: -1.568360 analytic: -1.568360, relative error: 3.738697e-09
numerical: 0.260181 analytic: 0.260180, relative error: 1.180620e-07
numerical: 0.358200 analytic: 0.358200, relative error: 2.820070e-08
numerical: 1.874052 analytic: 1.874052, relative error: 9.416010e-09
numerical: -0.389514 analytic: -0.389514, relative error: 1.777949e-07
numerical: 1.839289 analytic: 1.839289, relative error: 3.048306e-08
numerical: -0.290178 analytic: -0.290178, relative error: 1.709662e-07
numerical: -2.698520 analytic: -2.698520, relative error: 1.978944e-09
numerical: 1.151592 analytic: 1.151592, relative error: 1.637834e-08
numerical: -3.445950 analytic: -3.445950, relative error: 1.882229e-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.

```
In [8]: import time
```

```
In [9]: ## 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_ve
      ctorized, 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_vectoriz
      ed, np.linalg.norm(grad - grad_vectorized)))

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

```
Normal loss / grad_norm: 2.326425442427376 / 343.0326475036641 compute
d in 0.07816600799560547s
Vectorized loss / grad: 2.326425442427376 / 343.0326475036642 computed
in 0.05777096748352051s
difference in loss / grad: 0.0 /6.837463300096367e-14
```

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:

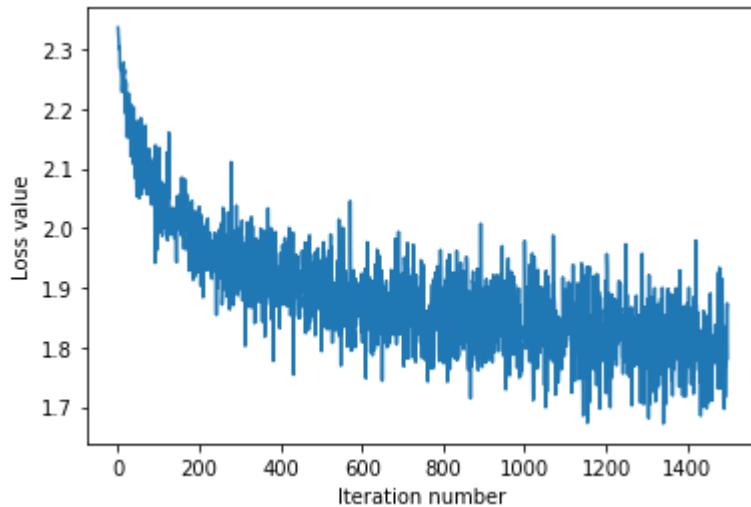
The softmax gradient descent training step is the same as the SVM training step.

```
In [10]: # 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 2.035774512066282
iteration 300 / 1500: loss 1.9813348165609888
iteration 400 / 1500: loss 1.9583142443981612
iteration 500 / 1500: loss 1.8622653073541355
iteration 600 / 1500: loss 1.8532611454359385
iteration 700 / 1500: loss 1.8353062223725827
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.8992158530357484
iteration 1000 / 1500: loss 1.9783503540252303
iteration 1100 / 1500: loss 1.8470797913532633
iteration 1200 / 1500: loss 1.8411450268664085
iteration 1300 / 1500: loss 1.7910402495792102
iteration 1400 / 1500: loss 1.870580302938226
That took 37.36585307121277s
```



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

```
In [11]: ## 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_p_
red)), ))
```

training accuracy: 0.3811428571428571
validation accuracy: 0.398

Optimize the softmax classifier

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

```
In [12]: np.finfo(float).eps
```

Out[12]: 2.220446049250313e-16

```
In [14]: # ===== #
# 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.
# ===== #
best_learning_rate = 0
best_val_accur = 0
best_test_accur = 0
for i in range(1, 10000, 200):
    learning_rate = i*1e-5
    softmax.train(X_train, y_train, learning_rate=learning_rate,
                  num_iters=500, verbose=False)
    y_train_pred = softmax.predict(X_train)
    train_accur = np.mean(np.equal(y_train,y_train_pred))
    y_val_pred = softmax.predict(X_val)
    val_accur = np.mean(np.equal(y_val, y_val_pred))

    if val_accur > best_val_accur:
        best_val_accur = val_accur
        best_learning_rate = learning_rate

softmax.train(X_train, y_train, learning_rate=best_learning_rate,
              num_iters=500, verbose=False)

y_test_pred = softmax.predict(X_test)
test_error = 1 - np.mean(np.equal(y_test, y_test_pred))

print('best validation accuracy: {}'.format(best_val_accur))
print('best learning rate: {}'.format(best_learning_rate))
print('test set error using the best learning rate: {}'.format(test_error))
# ===== #
# END YOUR CODE HERE
# ===== #
```

```
best validation accuracy: 0.339
best learning rate: 0.04401
test set error using the best learning rate: 0.763
```

softmax.py

```

1 import numpy as np
2
3 class Softmax(object):
4
5     def __init__(self, dims=[10, 3073]):
6         self.init_weights(dims=dims)
7
8     def init_weights(self, dims):
9         """
10        Initializes the weight matrix of the Softmax classifier.
11        Note that it has shape (C, D) where C is the number of
12        classes and D is the feature size.
13        """
14        self.W = np.random.normal(size=dims) * 0.0001
15
16    def loss(self, X, y):
17        """
18        Calculates the softmax loss.
19
20        Inputs have dimension D, there are C classes, and we operate on minibatches
21        of N examples.
22
23        Returns a tuple of:
24        - loss as single float
25        """
26
27
28        # Initialize the loss to zero.
29        loss = 0.0
30
31
32        # ===== #
33        # YOUR CODE HERE:
34        # Calculate the normalized softmax loss. Store it as the variable loss.
35        # (That is, calculate the sum of the losses of all the training
36        # set margins, and then normalize the loss by the number of
37        # training examples.)
38        # ===== #
39        minibatch_size = y.shape[0]
40        input_scores = self.W.dot(X.T)
41        loss = (1/float(minibatch_size)) * np.sum(np.log(np.sum(np.exp(input_scores), axis=0)) - np.choose(y, input_scores))
42        # ===== #
43
44        # END YOUR CODE HERE
45        # ===== #
46
47
48    def loss_and_grad(self, X, y):
49        """
50        Same as self.loss(X, y), except that it also returns the gradient.
51
52        Output: grad -- a matrix of the same dimensions as W containing
53        the gradient of the loss with respect to W.
54        """
55
56
57        # Initialize the loss and gradient to zero.
58        loss = 0.0
59        grad = np.zeros_like(self.W)
60
61
62        # ===== #
63        # YOUR CODE HERE:
64        # Calculate the softmax loss and the gradient. Store the gradient
65        # as the variable grad.
66        # ===== #
67        minibatch_size = y.shape[0]
68        input_scores = self.W.dot(X.T)
69        softmax_nominators = np.exp(self.W.dot(X.T))
70        softmax_denominators = np.sum(softmax_nominators, axis=0)
71        loss = (1 / float(minibatch_size)) * np.sum(
72            np.log(np.sum(np.exp(input_scores), axis=0)) - np.choose(y, input_scores))
73
74        for i in range(self.W.shape[0]):
75            for k in range(X.shape[0]):
76                if y[k] == i:
77                    grad[i] += X[k] * (softmax_nominators[i][k]/softmax_denominators[k] - 1)
78                else:
79                    grad[i] += X[k] * (softmax_nominators[i][k] / softmax_denominators[k])
80            grad[i] /= minibatch_size
81        # ===== #
82
83        # END YOUR CODE HERE
84        # ===== #
85
86    def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
87        """
88        sample a few random elements and only return numerical
89        in these dimensions.
90        """
91
92        for i in np.arange(num_checks):
93            ix = tuple([np.random.randint(m) for m in self.W.shape])

```

```

95     oldval = self.W[ix]
96     self.W[ix] = oldval + h # increment by h
97     fxph = self.loss(X, y)
98     self.W[ix] = oldval - h # decrement by h
99     fmxh = self.loss(X,y) # evaluate f(x - h)
100    self.W[ix] = oldval # reset
101
102    grad_numerical = (fxph - fmxh) / (2 * h)
103    grad_analytic = your_grad[ix]
104    rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) + abs(grad_analytic))
105    print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic, rel_error))
106
107 def fast_loss_and_grad(self, X, y):
108 """
109     A vectorized implementation of loss_and_grad. It shares the same
110     inputs and ouputs as loss_and_grad.
111 """
112
113 loss = 0.0
114 grad = np.zeros(self.W.shape) # initialize the gradient as zero
115
116 # ===== #
117 # YOUR CODE HERE:
118 #   Calculate the softmax loss and gradient WITHOUT any for loops.
119 # ===== #
120 minibatch_size = y.shape[0]
121 num_features = X.shape[1]
122 input_scores = self.W.dot(X.T)
123 loss = (1 / float(minibatch_size)) * np.sum(
124     np.log(np.sum(np.exp(input_scores - np.amax(input_scores, axis=0)), axis=0)) - np.choose(y, input_scores - np.amax(input_scores, axis=0)))
125 softmax_nominators = np.exp(input_scores - np.amax(input_scores, axis=0))
126 softmax_denominators = np.sum(softmax_nominators, axis=0)
127 softmax_matrix = softmax_nominators / softmax_denominators
128 softmax_matrix[y, np.arange(minibatch_size)] -= 1
129 softmax_matrix = np.tile(softmax_matrix.T, (1, 1, 1))
130 grad = (1/float(minibatch_size)) * np.sum(softmax_matrix.T * X, axis=1)
131 # ===== #
132 # END YOUR CODE HERE
133 # ===== #
134
135 return loss, grad
136
137 def train(self, X, y, learning_rate=1e-3, num_iters=100,
138           batch_size=200, verbose=False):
139 """
140     Train this linear classifier using stochastic gradient descent.
141
142     Inputs:
143     - X: A numpy array of shape (N, D) containing training data; there are N
144       training samples each of dimension D.
145     - y: A numpy array of shape (N,) containing training labels; y[i] = c
146       means that X[i] has label 0 <= c < C for C classes.
147     - learning_rate: (float) learning rate for optimization.
148     - num_iters: (integer) number of steps to take when optimizing
149     - batch_size: (integer) number of training examples to use at each step.
150     - verbose: (boolean) If true, print progress during optimization.
151
152     Outputs:
153     A list containing the value of the loss function at each training iteration.
154 """
155     num_train, dim = X.shape
156     num_classes = np.max(y) + 1 # assume y takes values 0...K-1 where K is number of classes
157
158     self.init_weights(dims=[np.max(y) + 1, X.shape[1]]) # initializes the weights of self.W
159
160     # Run stochastic gradient descent to optimize W
161     loss_history = []
162
163     for it in np.arange(num_iters):
164         X_batch = None
165         y_batch = None
166
167         # ===== #
168         # YOUR CODE HERE:
169         #   Sample batch_size elements from the training data for use in
170         #   gradient descent. After sampling,
171         #   - X_batch should have shape: (dim, batch_size)
172         #   - y_batch should have shape: (batch_size,)
173         #   The indices should be randomly generated to reduce correlations
174         #   in the dataset. Use np.random.choice. It's okay to sample with
175         #   replacement.
176         # ===== #
177         idx = np.random.randint(low=0, high=X.shape[0], size=batch_size)
178         X_batch = X[idx]
179         y_batch = y[idx]
180
181         # ===== #
182         # END YOUR CODE HERE
183         # ===== #
184
185         # evaluate loss and gradient
186         loss, grad = self.fast_loss_and_grad(X_batch, y_batch)
187         loss_history.append(loss)
188
189         # ===== #
190         # YOUR CODE HERE:
191         #   Update the parameters, self.W, with a gradient step
192         # ===== #
193         self.W = self.W - learning_rate * grad

```

```
193
194     # ===== #
195     # END YOUR CODE HERE
196     # ===== #
197
198     if verbose and it % 100 == 0:
199         print('iteration {} / {}: loss {}'.format(it, num_iters, loss))
200
201     return loss_history
202
203 def predict(self, X):
204     """
205     Inputs:
206     - X: N x D array of training data. Each row is a D-dimensional point.
207
208     Returns:
209     - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
210       array of length N, and each element is an integer giving the predicted
211       class.
212     """
213     y_pred = np.zeros(X.shape[1])
214     # ===== #
215     # YOUR CODE HERE:
216     #   Predict the labels given the training data.
217     # ===== #
218     y_pred = np.argmax(self.W.dot(X.T), axis=0)
219     # ===== #
220     # END YOUR CODE HERE
221     # ===== #
222
223     return y_pred
224
225
```