```python
import numpy as np
import pdb


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].
        # ================================================================ #

        trainSample=self.X_train[j]
        testSample = X[i]
        distance = norm(trainSample-testSample)
        dists[i, j] = distance

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

    trainsquared = np.sum(self.X_train**2, axis=1, keepdims=True)
    testsquared = np.sum(X**2, axis=1)
    multiplied = np.dot(self.X_train, X.T)
    dists = np.sqrt(trainsquared - 2*multiplied + testsquared)
    dists = dists.T

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

    return dists

  def compute_L1_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.
    # ========================================================================= #
    for i in range(0,num_test):
      # find the nearest training image to the i'th test image
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.abs(self.X_train - X[i,:]), axis = 1)
      dists[i]=distances


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

    return dists

  def compute_Linf_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.
    # ========================================================================= #
    for i in range(0,num_test):
      # find the nearest training image to the i'th test image
```

```
      # using the L1 distance (sum of absolute value differences)
      distances = np.sum(np.linalg.norm(self.X_train - X[i,:],ord="inf"), axis = 1)
      dists[i]=distances


    # ========================================================================= #
    # 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.
      # ========================================================================= #
      distance = dists[i]
      sortIndices = np.argsort(distance)
      bestOnes = sortIndices[0:k]
      closest_y = self.y_train[bestOnes]
      counts = np.bincount(closest_y)
      y_pred[i]=np.argmax(counts)
      # ========================================================================= #
      # END YOUR CODE HERE
      # ========================================================================= #

    return y_pred
```

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.

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 [1]:
```
import numpy as np # for doing most of our calculations
import matplotlib.pyplot as plt# for plotting
from utils.data_utils import load_CIFAR10 # function to load the CIFAR-10 da

# 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-ip
%load_ext autoreload
%autoreload 2
```

In [2]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = 'dataset\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', 's
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]:
```
# 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 [5]:
```
# Import the KNN class

from nndl import KNN
```

In [6]:
```
# 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) In the knn.train() function, we are simply storing all the training data samples and the corresponding labels in memory.

(2) An obivous pro is that the training step is extremely fast and simple to implement. However, a con is memory usage. As the amount of samples increase, the model needs more storage.

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 [7]:
```
# Implement the function compute_distances() in the KNN class.
# Do not worry about the input 'norm' for now; use the default definition of
#   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, '
```

```
Time to run code: 47.018675088882446
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 [8]:
```
# Implement the function compute_L2_distances_vectorized() in the KNN class.
# In this function, you ought to achieve the same L2 distance but WITHOUT an
# 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 b
```

```
Time to run code: 0.5874133110046387
Difference in L2 distances between your KNN implementations (should be 0): 0.0
```

In [12]:
```
print(dists_L2_vectorized.shape)
```

```
(500, 5000)
```

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 [9]:
```
# 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.
# ================================================================ #
distances = knn.compute_L2_distances_vectorized(X=X_test)
predictions = knn.predict_labels(distances,1)
error = sum(predictions!=y_test)/len(y_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 [10]:

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

X_train_folds = []
y_train_folds =  []
np.random.seed(24)
# ================================================================ #
# 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]
# ================================================================ #
indices = np.arange(0,X_train.shape[0])
randomIndices=np.random.permutation(indices)
foldSize = int(X_train.shape[0]/num_folds)
start = 0
end = start + foldSize
for i in range(0,num_folds-1):
    indices = randomIndices[start:end]
    foldtraindata=X_train[indices]
    foldtrainlabel = y_train[indices]
    X_train_folds.append(foldtraindata)
    y_train_folds.append(foldtrainlabel)
    start = end
    end = start+foldSize
indices = randomIndices[start:]
foldtraindata=X_train[indices]
foldtrainlabel = y_train[indices]
X_train_folds.append(foldtraindata)
y_train_folds.append(foldtrainlabel)

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


```

Optimizing the number of nearest neighbors hyperparameter.

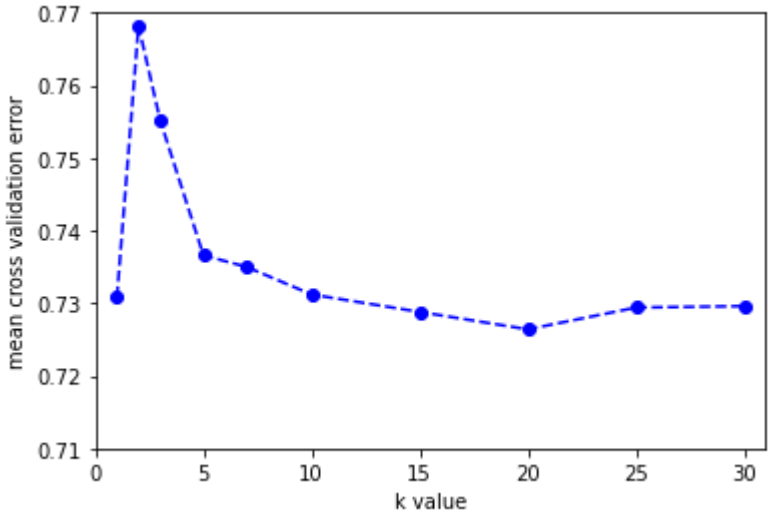
In this section, we select different numbers of nearest neighbors and assess which one has the lowest k-fold cross validation error.

In [11]:

```
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.
# ================================================================ #
knn = KNN()
kError = []
for k in ks:
    foldError = []
    for i in range(0,len(X_train_folds)):
        trainCopy = X_train_folds.copy()
        trainLabelsCopy = y_train_folds.copy()
        foldTest = trainCopy[i]
        foldTestlabel = trainLabelsCopy[i]
        trainCopy.pop(i)
        trainLabelsCopy.pop(i)
        foldTr=np.array(trainCopy)
        foldTrLab=np.array(trainLabelsCopy)
        foldTr = foldTr.reshape([foldTr.shape[0]*foldTr.shape[1],foldTr.shap
        foldTrLab = foldTrLab.reshape([foldTrLab.shape[0]*foldTrLab.shape[1]
        knn.train(foldTr,foldTrLab)
        distances = knn.compute_L2_distances_vectorized(X=foldTest)
        predictions = knn.predict_labels(distances,k)
        error = sum(predictions!=foldTestlabel)/len(foldTestlabel)
        foldError.append(error)
    print("Done cross validation for k %d. Error is %.2f"%(k,np.mean(foldErr
    kError.append(np.mean(foldError))


plt.plot(ks, kError, 'bo--')
plt.axis([0, 31, 0.71, 0.77])
plt.xlabel('k value')
plt.ylabel('mean cross validation error')
plt.show()
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

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

```
Done cross validation for k 1. Error is 0.73
Done cross validation for k 2. Error is 0.77
Done cross validation for k 3. Error is 0.76
Done cross validation for k 5. Error is 0.74
Done cross validation for k 7. Error is 0.73
Done cross validation for k 10. Error is 0.73
Done cross validation for k 15. Error is 0.73
Done cross validation for k 20. Error is 0.73
```

```
Done cross validation for k 25. Error is 0.73
Done cross validation for k 30. Error is 0.73
```




```
Computation time: 35.01
```

In [12]:
```
1 kError
```

Out[12]:
```
[0.7307999999999999,
 0.7681999999999999,
 0.7552000000000001,
 0.7365999999999999,
 0.735,
 0.7312,
 0.7288,
 0.7264,
 0.7293999999999999,
 0.7295999999999999]
```

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 of k is found to be 20.

(2) The cross validation error is 0.7264.

Optimizing the norm

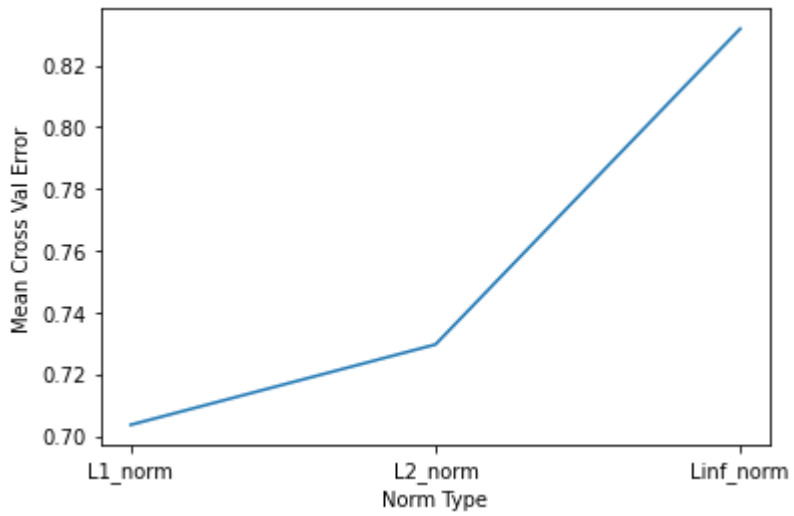
Next, we test three different norms (the 1, 2, and infinity norms) and see which distance metric results in the best cross-validation performance.

In [13]:

```
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]
normNames=["l1","l2","inf"]
# ============================================================ #
# 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.
# ============================================================ #
bestK=20
knn = KNN()
normError = []
for ind in range(0,len(norms)):
    normType = norms[ind]
    normName = normNames[ind]
    foldError = []
    for i in range(0,len(X_train_folds)):
        trainCopy = X_train_folds.copy()
        trainLabelsCopy = y_train_folds.copy()
        foldTest = trainCopy[i]
        foldTtestlabel = trainLabelsCopy[i]
        trainCopy.pop(i)
        trainLabelsCopy.pop(i)
        foldTr=np.array(trainCopy)
        foldTrLab=np.array(trainLabelsCopy)
        foldTr = foldTr.reshape([foldTr.shape[0]*foldTr.shape[1],foldTr.shap
        foldTrLab = foldTrLab.reshape([foldTrLab.shape[0]*foldTrLab.shape[1]
        knn.train(foldTr,foldTrLab)
        if normName=="l2":
           distances = knn.compute_L2_distances_vectorized(foldTest)
        else:
           distances = knn.compute_distances(foldTest,normType)

        #distances = knn.compute_distances(X=foldTest,norm=normType)
        predictions = knn.predict_labels(distances,k)
        error = sum(predictions!=foldTtestlabel)/len(foldTtestlabel)
        print("Fold done")
        foldError.append(error)
    print("Done cross validation for norm %s. Error is %.2f"%(normName,np.me
    normError.append(np.mean(foldError))
plt.figure()
normNames=['L1_norm', 'L2_norm', 'Linf_norm']
plt.plot(normNames,normError)
plt.xlabel('Norm Type')
plt.ylabel('Mean Cross Val Error')
# ============================================================ #
```

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

```
Fold done
Fold done
Fold done
Fold done
Fold done
Done cross validation for norm l1. Error is 0.70
Fold done
Fold done
Fold done
Fold done
Fold done
Done cross validation for norm l2. Error is 0.73
Fold done
Fold done
Fold done
Fold done
Fold done
Done cross validation for norm inf. Error is 0.83
Computation time: 711.44
```



In [64]:
```
normError
```

Out[64]:
```
[0.7036, 0.7295999999999999, 0.8318]
```

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) L1 norm has the best cross-validation error.

(2) It is 0.7036

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 [14]:
```
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.
# ================================================================ #
bestK=20
bestKnn=KNN()
bestKnn.train(X=X_train, y=y_train)
distances = bestKnn.compute_distances(X=X_test,norm=L1_norm)
predictions = bestKnn.predict_labels(distances,bestK)
error = sum(predictions!=y_test)/len(y_test)

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

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

```
Error rate achieved: 0.72
```

Question:

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

Answer:

The naive model with k=1 had a test error of 0.726. By implementing cross validation and choosing the best k and norm values, we managed to reduce the error to 0.72. Therefore, we improved the performance by 0.06.

Deniz Orkun Eren
905624625

ECE C247 HW #2

2) Softmax Classifier Gradient

Let $w_i^T x^{(j)} + b_i = o_i$

$$\text{Likelihood} = \prod_{j=1}^{m} \prod_{n=1}^{c} Pr(y^{(j)} = n \mid x^{(j)}, \theta)^{\delta_n(y^{(j)})}$$

where $\delta_n(y^{(j)}) = \begin{cases} 1 & \text{if } y^{(j)} = n \\ 0 & \text{otw.} \end{cases}$

We are given that $Pr(y^{(j)} = n \mid x^{(j)}, \theta) = \text{softmax}_n(x^{(j)}) = \dfrac{e^{o_n}}{\sum_{k=0}^{c} e^{o_k}}$

So, $$\text{likelihood} = \prod_{j=1}^{m} \prod_{n=1}^{c} \text{softmax}_n(x^{(j)})^{\delta_n(y^{(j)})}$$

$$\mathcal{L} = \text{Log-likelihood} = \sum_{j=1}^{m} \sum_{n=1}^{c} \delta_n(y^{(j)}) \log(\text{softmax}_n(x^{(j)}))$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial \text{softmax}_n} \cdot \frac{\partial \text{softmax}_n}{\partial o_i} \cdot \frac{\partial o_i}{\partial w_i}$$

$$\frac{dL}{do_i} = \sum_{j=1}^{m} \delta_i(y^{(j)}) \cdot \frac{1}{\text{Softmax}_i(x^{(j)})} \cdot \frac{d\,\text{Softmax}_i(x^{(j)})}{do_i} + \sum_{n \neq i} \delta_n(y^{(j)}) \cdot \frac{1}{\text{Softmax}_n(x^{(j)})} \cdot \frac{d\,\text{Softmax}_n(x^{(j)})}{do_i}$$

$$\frac{d\,\text{Softmax}_i(x^{(j)})}{do_i} = \frac{d\,\frac{e^{o_i}}{\sum_{k=1}^{c} e^{o_k}}}{do_i} = \frac{e^{o_i} \cdot \sum_{k=1}^{c} e^{o_k} - e^{o_i} \cdot e^{o_i}}{\left(\sum_{k=1}^{c} e^{o_k}\right)^2}$$

$$= \frac{e^{o_i}}{\sum_k e^{o_k}} - \left(\frac{e^{o_i}}{\sum_k e^{o_k}}\right)^2$$

$$= \text{Softmax}_i(x^{(j)}) \left[1 - \text{Softmax}_i(x^{(j)})\right]$$

$$\frac{d\,\text{Softmax}_n(x^{(j)})}{do_i} \quad \text{when } n \neq i = \frac{d\,\frac{e^{o_n}}{\sum_k e^{o_k}}}{do_i}$$

$$= \frac{0 \cdot \sum_k e^{o_k} - e^{o_n} \cdot e^{o_i}}{\left(\sum_k e^{o_k}\right)^2}$$

$$= -\text{Softmax}_n(x^{(j)}) \cdot \text{Softmax}_i(x^{(j)})$$

Plugging in to $\frac{dL}{do_i}$ gives:

$$\frac{dL}{do_i} = \sum_{j=1}^{m} \delta_i(y^{(j)}) \cdot \frac{softmax_i(x^{(j)}) \cdot [1 - softmax_i(x^{(j)})]}{softmax_i(x^{(j)})} + \sum_{n \neq i} \delta_n(y^{(j)}) \cdot \frac{-softmax_n(x^{(j)}) \cdot softmax_i(x^{(j)})}{softmax_n(x^{(j)})}$$

$$= \sum_{j=1}^{m} \delta_i(y^{(j)}) \cdot [1 - softmax_i(x^{(j)})] + \sum_{n \neq i} \delta_n(y^{(j)}) \cdot -softmax_i(x^{(j)})$$

$$= \sum_{j=1}^{m} \delta_i(y^{(j)}) - softmax_i(x^{(j)}) \cdot \sum_{n=1}^{c} \delta_n(y^{(j)})$$

We know that $\sum_{n=1}^{c} \delta_n(y^{(j)}) = 1$ since $\delta_n(y^{(j)})$ is 1 when $y^{(j)} = n$, 0 otw.

So, $$\frac{dL}{do_i} = \sum_{j=1}^{m} \delta_i(y^{(j)}) - softmax_i(x^{(j)})$$

$$\frac{do_i}{dw_i} = \frac{d\, w_i^T x^{(j)} + b_i}{dw_i} = x^{(j)}$$

$$\frac{do_i}{db_i} = \frac{d\, w_i^T x^{(j)} + b_i}{db_i} = 1$$

So, $$\frac{dL}{dw_i} = \frac{dL}{do_i} \cdot \frac{do_i}{dw_i} = \sum_{j=1}^{m} [\delta_i(y^{(j)}) - softmax_i(x^{(j)})] \cdot x^{(j)}$$

$$\frac{dL}{db_i} = \frac{dL}{do_i} \cdot \frac{do_i}{db_i} = \sum_{j=1}^{m} \delta_i(y^{(j)}) - softmax_i(x^{(j)})$$

```python
import numpy as np


class Softmax(object):

  def __init__(self, dims=[10, 3073]):
    self.init_weights(dims=dims)

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

  def loss(self, X, y):
    """
    Calculates the softmax loss.

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

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

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

    # Initialize the loss to zero.
    loss = 0.0

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

    for index in range(0,X.shape[0]):
        data = X[index]
        multiplication = weights.dot(data.T).T
        exponentials = np.exp(multiplication)
        exponentialSum = np.sum(exponentials)
        exponentials=exponentials/exponentialSum
        label = y[index]
        crossEntropy = -np.log(exponentials[label])
        losses.append(crossEntropy)
    loss = sum(losses)/len(losses)
    # ============================================================================ #
    # END YOUR CODE HERE
```

```
    # ================================================================ #

    return loss

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

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

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

    # ================================================================ #
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the gradient
    #   as the variable grad.
    # ================================================================ #
    weights = self.W
    losses=[]
    for index in range(0,X.shape[0]):
        data = X[index]
        multiplication = weights.dot(data.T).T
        exponentials = np.exp(multiplication)
        exponentialSum = np.sum(exponentials)
        exponentials=exponentials/exponentialSum
        label = y[index]
        crossEntropy = -np.log(exponentials[label])
        losses.append(crossEntropy)
        yHot = np.zeros([1,np.max(y)+1])
        yHot[0,label]=1
        data = np.reshape(data,[1,data.shape[0]])
        exponentials = np.reshape(exponentials,[1,exponentials.shape[0]])
        derivative = np.dot(data.T, (exponentials - yHot))
        grad=grad+derivative.T
    loss = sum(losses)/len(losses)
    grad = grad/[X.shape[0]]

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

    return loss, grad

  def grad_check_sparse(self, X, y, your_grad, num_checks=10, h=1e-5):
    """
    sample a few random elements and only return numerical
    in these dimensions.
    """

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

      oldval = self.W[ix]
```

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

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

  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.
    # ============================================================== #
    weights = self.W
    multiplication =weights.dot(X.T).T
    multiplication = (multiplication.T - np.amax(multiplication,axis = 1)).T
    soft = np.exp(multiplication)
    sums = np.sum(soft,axis=1)
    probs = (soft.T / sums).T
    predsForClass = probs[np.arange(y.size),y]
    loss = -np.log(predsForClass+1e-10) #To aviod log(0)
    loss = np.mean(loss)
    yHot = np.zeros([y.shape[0],9+1])
    yHot[np.arange(y.size),y] = 1
    grad = (1/X.shape[0])*np.dot(X.T, (probs - yHot)).T

    # ============================================================== #
    # 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 classe

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

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

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

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

      # evaluate loss and gradient
      loss, grad = self.fast_loss_and_grad(XBatch, yBatch)
      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.
# ================================================================ #
weights = self.W
multiplication =weights.dot(X.T).T
soft = np.exp(multiplication)
y_pred=np.argmax(soft,axis=1)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

return y_pred
```

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.

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

In [1]:
```
import random
import numpy as np
from utils.data_utils import load_CIFAR10
import matplotlib.pyplot as plt

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

In [5]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000,
    """
    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 th
    SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'dataset\cifar-10-batches-py' # You need to update this li
    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_d
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 [6]:
```
from nndl import Softmax
```

In [7]:
```
# 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 r

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)
```

```
2.3277607028048966
```

Question:

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

Answer:

Because there are a total of 10 classes, an untrained classifier will guess 0.1 probability for each class. Therefore, for each sample we would expect an error of -log(0.1) = 2.3. Therefore, the mean error is also expected to be 2.3.

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 y
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -1.266499 analytic: -1.266499, relative error: 1.289325e-08
numerical: 0.056842 analytic: 0.056842, relative error: 4.306434e-07
numerical: 0.449315 analytic: 0.449315, relative error: 2.518900e-08
numerical: 0.935716 analytic: 0.935716, relative error: 1.540367e-08
numerical: -0.512110 analytic: -0.512110, relative error: 9.848258e-08
numerical: 0.342088 analytic: 0.342088, relative error: 2.513967e-08
numerical: -0.967954 analytic: -0.967954, relative error: 6.087353e-08
numerical: -1.891557 analytic: -1.891557, relative error: 1.427829e-08
numerical: -0.247811 analytic: -0.247811, relative error: 6.318535e-08
numerical: -2.658215 analytic: -2.658215, relative error: 2.500902e-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 gradie
#    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.lin

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_vectoriz

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

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

```
Normal loss / grad_norm: 2.3230138756030048 / 281.3263564766588 computed in 0.0
7181096076965332s
Vectorized loss / grad: 2.323013874528945 / 281.3263564766588 computed in 0.009
247779846191406s
difference in loss / grad: 1.0740599520886462e-09 /1.9427077332200875e-13
```

Stochastic gradient descent

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

Question:

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

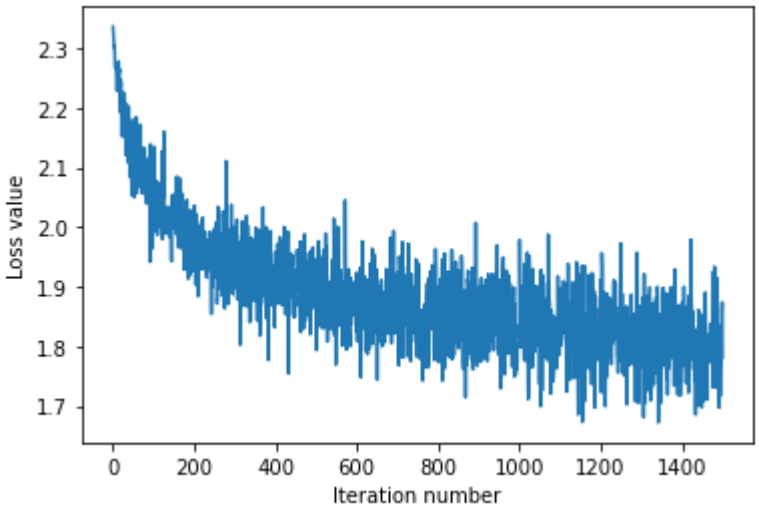
Answer:

Due to the change in loss function, the value of the gradients that are used to update the coefficient will change

In [46]:

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


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

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

```
iteration 0 / 1500: loss 2.3365926606637544
iteration 100 / 1500: loss 2.0557222613850827
iteration 200 / 1500: loss 2.0357745120662813
iteration 300 / 1500: loss 1.9813348165609888
iteration 400 / 1500: loss 1.9583142443981612
iteration 500 / 1500: loss 1.8622653073541355
iteration 600 / 1500: loss 1.8532611454359382
iteration 700 / 1500: loss 1.8353062223725827
iteration 800 / 1500: loss 1.829389246882764
iteration 900 / 1500: loss 1.8992158530357484
iteration 1000 / 1500: loss 1.97835035402523
iteration 1100 / 1500: loss 1.8470797913532633
iteration 1200 / 1500: loss 1.8411450268664082
iteration 1300 / 1500: loss 1.7910402495792102
iteration 1400 / 1500: loss 1.8705803029382257
That took 6.137814283370972s
```



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

In [47]:
```
## Implement softmax.predict() and use it to compute the training and testin

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

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

Optimize the softmax classifier

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

In [10]:
```
np.finfo(float).eps
```

Out[10]:
```
2.220446049250313e-16
```

In [11]:

```
# ================================================================ #
# 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 er
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #
softmax = Softmax(dims=[num_classes, num_features])
rates = [10**i for i in range(-10,0)]

valAccuracies = []
valLosses = []
for rate in rates:
    softmax.train(X_train, y_train, learning_rate=rate, num_iters=1500, verb
    valLoss,a = softmax.fast_loss_and_grad(X_val, y_val)
    valPreds = softmax.predict(X_val)
    valAcc = np.mean(np.equal(y_val, valPreds))
    valAccuracies.append(valAcc)
    valLosses.append(valLoss)
    print('Current rate:',rate,'validation accuracy: {}'.format(valAcc),'val
    print("Best validation loss so far {}".format(min(valLosses)))
print("-"*100)
bestIndex = np.argmin(valLosses)
bestLR = rates[bestIndex]
bestValLoss = valLosses[bestIndex]
bestValAcc = valAccuracies[bestIndex]
print("The best learning rate is %f.Best validation loss is %f. Best validat


softmax.train(X_train, y_train, learning_rate=bestLR, num_iters=1500, verbos
yTestPred = softmax.predict(X_test)
testAcc = np.mean(np.equal(y_test, yTestPred))
print('Error of the softmax classifier with best learning rate %f on the tes
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

```

```
Current rate: 1e-10 validation accuracy: 0.13 validation Loss: 2.33296360971419
3
Best validation loss so far 2.332963609714193
Current rate: 1e-09 validation accuracy: 0.179 validation Loss: 2.2418688560194
973
Best validation loss so far 2.2418688560194973
Current rate: 1e-08 validation accuracy: 0.302 validation Loss: 2.0201445650194
59
Best validation loss so far 2.020144565019459
Current rate: 1e-07 validation accuracy: 0.379 validation Loss: 1.8287151179489
731
Best validation loss so far 1.8287151179489731
Current rate: 1e-06 validation accuracy: 0.413 validation Loss: 1.7466767713611
```

```
042
Best validation loss so far 1.7466767713611042
Current rate: 1e-05 validation accuracy: 0.316 validation Loss: 2.5812750026016
156
Best validation loss so far 1.7466767713611042
Current rate: 0.0001 validation accuracy: 0.256 validation Loss: 13.73004382669
0776
Best validation loss so far 1.7466767713611042
Current rate: 0.001 validation accuracy: 0.254 validation Loss: 16.890411756297
777
Best validation loss so far 1.7466767713611042
Current rate: 0.01 validation accuracy: 0.145 validation Loss: 16.7166763205028
45
Best validation loss so far 1.7466767713611042
Current rate: 0.1 validation accuracy: 0.181 validation Loss: 17.1312330918501
Best validation loss so far 1.7466767713611042
--------------------------------------------------------------------------------
----------------------
The best learning rate is 0.000001.Best validation loss is 1.746677. Best valid
ation accuracy is 0.413000
Error of the softmax classifier with best learning rate 0.000001 on the test Se
t Error Rate is 0.622000
```