Problem 1

(a) First, we can expand and simplify the equation for the modified loss

$$\begin{aligned}
\tilde{\mathcal{L}}(\theta) &= \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta)^2 \\
&= \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta)^T(y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta) \\
&= \frac{1}{N}\sum_{i=1}^{N} y^{(i)^2} - 2y^{(i)^T}(x^{(i)} + \delta^{(i)})^T\theta + \theta^T(x^{(i)} + \delta^{(i)})(x^{(i)} + \delta^{(i)})^T\theta \\
&= \frac{1}{N}\sum_{i=1}^{N} y^{(i)^2} - 2y^{(i)^T}x^{(i)^T}\theta - 2y^{(i)^T}\delta^{(i)^T}\theta + \theta^T(x^{(i)^2} + 2x^{(i)}\delta^{(i)^T} + \delta^{(i)^2})\theta \\
&= \frac{1}{N}\sum_{i=1}^{N} y^{(i)^2} - 2y^{(i)^T}x^{(i)^T}\theta - 2y^{(i)^T}\delta^{(i)^T}\theta + \theta^T x^{(i)^2}\theta + 2\theta^T x^{(i)}\delta^{(i)^T}\theta + \theta^T\delta^{(i)^2}\theta \\
&= \frac{1}{N}\sum_{i=1}^{N} \left(y^{(i)^2} - 2y^{(i)^T}x^{(i)^T}\theta + \theta^T x^{(i)^2}\theta\right) + \left(-2y^{(i)^T}\delta^{(i)^T}\theta + 2\theta^T x^{(i)}\delta^{(i)^T}\theta + \theta^T\delta^{(i)^2}\theta\right) \\
&= \mathcal{L}(\theta) + \frac{1}{N}\sum_{i=1}^{N} \left(-2y^{(i)^T}\delta^{(i)^T}\theta + 2\theta^T x^{(i)}\delta^{(i)^T}\theta + \theta^T\delta^{(i)^2}\theta\right).
\end{aligned}$$

Then, we can simplify the equation for the expected value of the modified loss

$$\begin{aligned}
\mathbb{E}_{\delta\sim\mathcal{N}}[\tilde{\mathcal{L}}(\theta)] &= \mathbb{E}_{\delta\sim\mathcal{N}}\left[\frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta)^2\right] \\
&= \mathcal{L}(\theta) + \frac{1}{N}\sum_{i=1}^{N} -2y^{(i)^T}\mathbb{E}_{\delta\sim\mathcal{N}}[\delta^{(i)^T}]\theta + 2\theta^T x^{(i)}\mathbb{E}_{\delta\sim\mathcal{N}}[\delta^{(i)^T}]\theta + \theta^T\mathbb{E}_{\delta\sim\mathcal{N}}[\delta^{(i)^2}]\theta \\
&= \mathcal{L}(\theta) + \frac{1}{N}\sum_{i=1}^{N} \theta^T\mathbb{E}_{\delta\sim\mathcal{N}}[\delta^{(i)^2}]\theta \\
&= \mathcal{L}(\theta) + \frac{1}{N}\sum_{i=1}^{N} \theta^T\sigma^2\mathbf{I}\theta \\
&= \mathcal{L}(\theta) + \frac{1}{N}\sum_{i=1}^{N} \sigma^2\theta^T\theta \\
&= \mathcal{L}(\theta) + \sigma^2\theta^T\theta.
\end{aligned}$$

(b) The addition of the noise would penalize large weights, as the regularization term is proportional to the magnitude of θ. When the regularization term increases, so does the expected value of the cost function, so it imposes a bias towards smaller weights.

(c) As $\sigma \to 0$, the regularization term approaches zero as well. This means that the model essentially reduces to the linear regression model without any regularization. In other words, the resulting model is likely to be more complex and has a higher chance to overfit the data.

(d) As $\sigma \to \infty$, the regularization term approaches ∞ as well. This means that the model penalizes large weights more. This means that the resulting model is likely to be less complex and has a lower chance to overfit the data.

Problem 2

knn.py:

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

        dists[i, j] = norm(X[i] - self.X_train[j])
      # ================================================================ #
      # END YOUR CODE HERE
      # ================================================================ #

    return dists

  def compute_L2_distances_vectorized(self, X):
    """
    Compute the distance between each test point in X and each training point
    in self.X_train WITHOUT using any for loops.

    Inputs:
    - X: A numpy array of shape (num_test, D) containing test data.

    Returns:
    - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
      is the Euclidean distance between the ith test point and the jth training
      point.
    """
    num_test = X.shape[0]
    num_train = self.X_train.shape[0]
    dists = np.zeros((num_test, num_train))

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

    X_train_squared = np.sum(self.X_train ** 2, axis=1).flatten()
    X_test_squared = np.array([np.sum(X ** 2, axis=1).flatten()]).T
    dists = np.sqrt((X_train_squared + X_test_squared) - 2*(X @ self.X_train.T))

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

  closest_train = np.argsort(dists[i])[:k]
  closest_y = sorted([self.y_train[i] for i in closest_train])
  y_pred[i] = max(set(closest_y), key=closest_y.count)

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

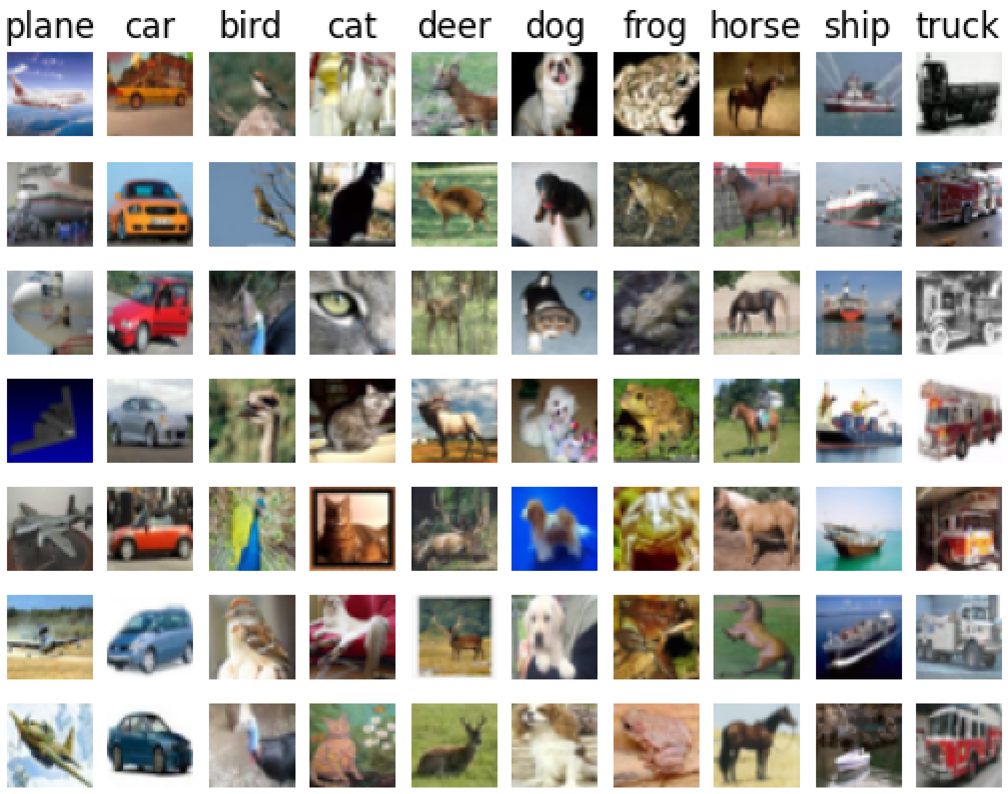
Import the appropriate libraries

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 [37]:
```
# 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 [38]:
```
# 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 [39]:
```
# 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 [19]:
```
# Import the KNN class

from nndl import KNN
```

In [20]:
```
# 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.train()` simply sets the instance variables of the KNN class to the training data (i.e. stores the training images and their corresponding labels).

(2) The training step is very fast because all it does is set instance variables. However, this also means that the model is storing all of the data in memory, which is very inefficient with large datasets.

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: 13.604048252105713
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 ar
# 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: 0.09776687622070312
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 [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.
# ================================================================ #
```

```
y_pred = knn.predict_labels(dists_L2, k=1)
error = 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 [31]:

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

X_train_folds = []
y_train_folds =  []

# ================================================================ #
# YOUR CODE HERE:
#   Split the training data into num_folds (i.e., 5) folds.
#   X_train_folds is a list, where X_train_folds[i] contains the
#      data points in fold i.
#   y_train_folds is also a list, where y_train_folds[i] contains
#      the corresponding labels for the data in X_train_folds[i]
# ================================================================ #

np.random.seed(1)

shuffled = np.random.permutation(num_training)
X_train_shuffled = X_train[shuffled]
y_train_shuffled = y_train[shuffled]
fold_size = num_training // num_folds

for i in range(num_folds):
    X_train_folds.append(X_train_shuffled[i*fold_size:(i+1)*fold_size])
    y_train_folds.append(y_train_shuffled[i*fold_size:(i+1)*fold_size])

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

Optimizing the number of nearest neighbors hyperparameter.

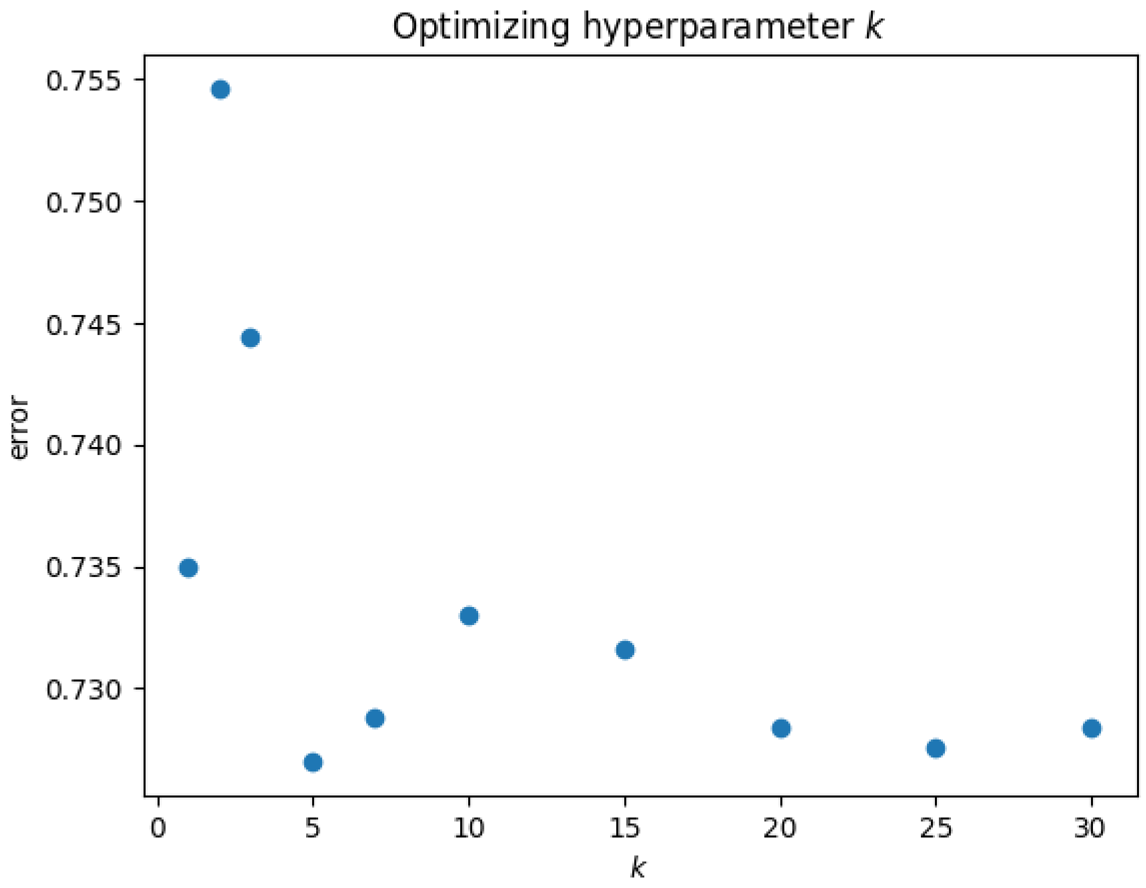
In this section, we select different numbers of nearest neighbors and assess which one has the lowest k-fold cross validation error.

In [32]:
```
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.
# ================================================================ #

ks_errors = []
for k in ks:
    print(f'Evaluating k={k}...')
    k_errors = []
    for i in range(num_folds):
        X_val, y_val = X_train_folds[i], y_train_folds[i]
        X_train_cv = np.concatenate(X_train_folds[:i] + X_train_folds[i+1:])
        y_train_cv = np.concatenate(y_train_folds[:i] + y_train_folds[i+1:])

        knn.train(X_train_cv, y_train_cv)
        dists = knn.compute_L2_distances_vectorized(X=X_val)

        y_pred = knn.predict_labels(dists, k=k)
        error = np.sum(y_pred != y_val) / fold_size
        k_errors.append(error)
    ks_errors.append(np.mean(k_errors))

k_opt = ks[np.argmin(ks_errors)]
print(f'Best value of k: {k_opt} (error = {min(ks_errors)})')

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

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

```
Evaluating k=1...
Evaluating k=2...
Evaluating k=3...
Evaluating k=5...
Evaluating k=7...
Evaluating k=10...
Evaluating k=15...
Evaluating k=20...
Evaluating k=25...
Evaluating k=30...
Best value of k: 5 (error = 0.727)
Computation time: 16.75
```

In [33]:
```
plt.scatter(ks, ks_errors)
plt.xlabel('$k$')
plt.ylabel('error')
plt.title('Optimizing hyperparameter $k$')
```

Out[33]:
```
Text(0.5, 1.0, 'Optimizing hyperparameter $k$')
```



Questions:

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

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

Answers:

(1) $k = 5$ is the best value among the tested k's.

(2) The cross-validation error for $k = 5$ is 0.727.

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 [34]:
```
time_start =time.time()

L1_norm = lambda x: np.linalg.norm(x, ord=1)
L2_norm = lambda x: np.linalg.norm(x, ord=2)
Linf_norm = lambda x: np.linalg.norm(x, ord= np.inf)
norms = [L1_norm, L2_norm, Linf_norm]

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the cross-validation error for each norm in norms, testing
#   the trained model on each of the 5 folds.  Average these errors
#   together and make a plot of the norm used vs the cross-validation error
#   Use the best cross-validation k from the previous part.
#
#   Feel free to use the compute_distances function.  We're testing just
#   three norms, but be advised that this could still take some time.
#   You're welcome to write a vectorized form of the L1- and Linf- norms
#   to speed this up, but it is not necessary.
# ================================================================ #

norms_errors = []
for i, norm in enumerate(norms):
    print(f'Evaluating norm #{i+1}...')
    norm_errors = []
    for i in range(num_folds):
        X_cv_test, y_cv_test = X_train_folds[i], y_train_folds[i]
        X_cv_train = np.concatenate(X_train_folds[:i] + X_train_folds[i+1:])
        y_cv_train = np.concatenate(y_train_folds[:i] + y_train_folds[i+1:])

        knn.train(X_cv_train, y_cv_train)
        dists = knn.compute_distances(X=X_cv_test, norm=norm)

        y_pred = knn.predict_labels(dists, k=k_opt)
        error = np.sum(y_pred != y_cv_test) / fold_size
        norm_errors.append(error)
    norms_errors.append(np.mean(norm_errors))

i_opt = np.argmin(norms_errors)
print(f'Best norm: norm #{i_opt+1} (error = {norms_errors[i_opt]})')

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

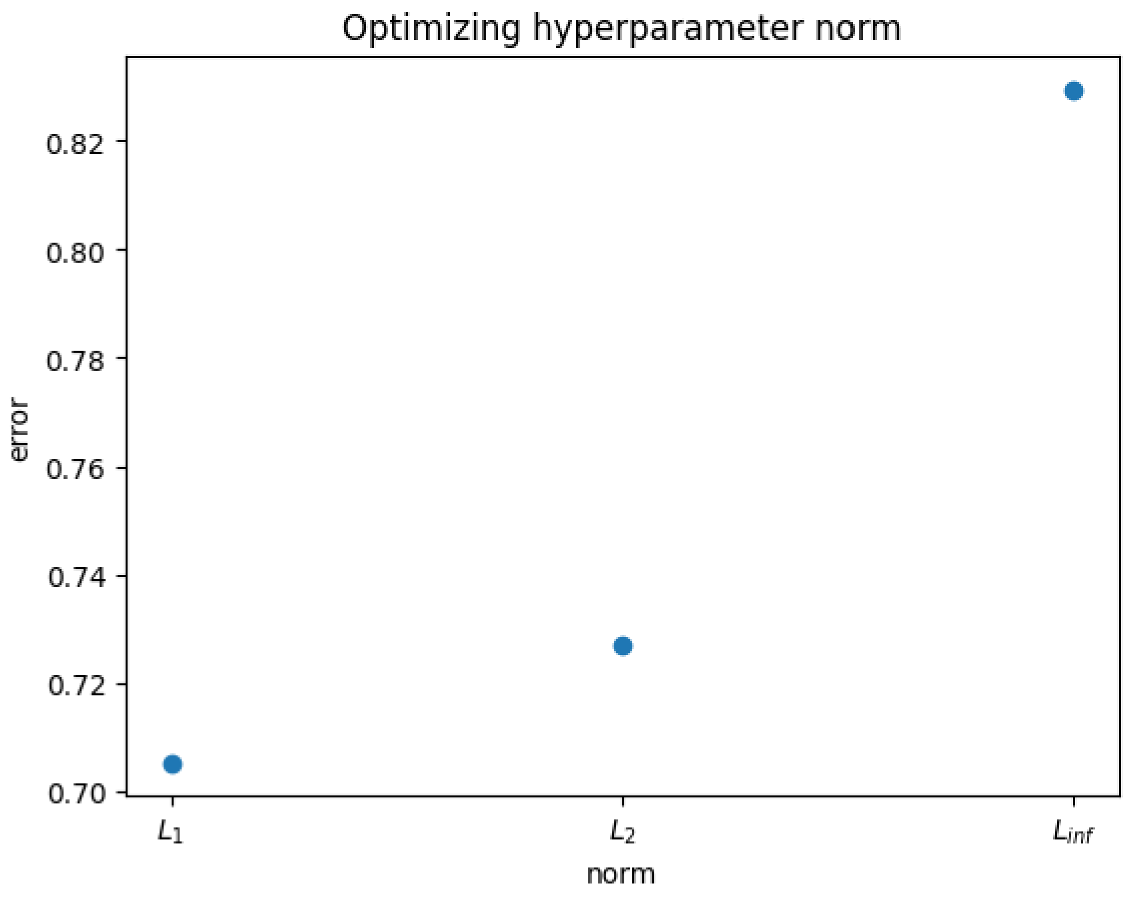
```
Evaluating norm #1...
Evaluating norm #2...
Evaluating norm #3...
Best norm: norm #1 (error = 0.7051999999999999)
Computation time: 269.35
```

In [35]:

```
norm_labels = ['$L_1$', '$L_2$', '$L_{inf}$']
plt.xticks(np.arange(3), norm_labels)
plt.scatter(norm_labels, norms_errors)
plt.xlabel('norm')
plt.ylabel('error')
plt.title('Optimizing hyperparameter norm')
```

Out[35]:

```
Text(0.5, 1.0, 'Optimizing hyperparameter norm')
```



Questions:

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

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

Answers:

(1) The L1-norm has the best cross-validation error.

(2) The cross-validation error for the L1-norm with $k = 5$ is about 0.705.

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 [40]:
```
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)
dists_L1 = knn.compute_distances(X=X_test, norm=L1_norm)
y_pred = knn.predict_labels(dists_L1, k=k_opt)
error = np.sum(y_pred != y_test) / num_test

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

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

```
Error rate achieved: 0.7
```

Question:

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

Answer:

My error improved from 0.726 to 0.7, a difference of 0.026.

In [ ]:

Problem 3

We are given the probabilistic model

$$\Pr(y^{(j)} = i|\mathbf{x}^{(j)}, \theta) = \text{softmax}_i(\mathbf{x}^{(j)}),$$

where

$$\text{softmax}_i(\mathbf{x}) = \frac{e^{\mathbf{w}_i^T\mathbf{x}+b_i}}{\sum_{k=1}^{c} e^{\mathbf{w}_k^T\mathbf{x}+b_k}}.$$

The likelihood of observing the samples, given θ, is defined by

$$\begin{aligned}
p(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}, y^{(1)}, \dots, y^{(m)}|\theta) &= \prod_{i=1}^{m} p(\mathbf{x}^{(i)}, y^{(i)}|\theta)\\
&= \prod_{i=1}^{m} p(\mathbf{x}^{(i)}|\theta)p(y^{(i)}|\mathbf{x}^{(i)}, \theta)\\
&= \prod_{i=1}^{m} p(\mathbf{x}^{(i)}|\theta)\text{softmax}_{y^{(i)}}(\mathbf{x}^{(j)})\\
\mathcal{L} &= \log\left[\prod_{i=1}^{m} p(\mathbf{x}^{(i)}|\theta)\text{softmax}_{y^{(i)}}(\mathbf{x}^{(i)})\right]\\
&= \sum_{i=1}^{m} \log\left(p(\mathbf{x}^{(i)}|\theta)\right) + \log\left(\text{softmax}_{y^{(i)}}(\mathbf{x}^{(i)})\right)\\
&= \sum_{i=1}^{m} \log\left(p(\mathbf{x}^{(i)}|\theta)\right) + \log\left(\frac{e^{\mathbf{w}_{y^{(i)}}^T\mathbf{x}^{(i)}+b_{y^{(i)}}}}{\sum_{k=1}^{c} e^{\mathbf{w}_k^T\mathbf{x}^{(i)}+b_k}}\right)\\
&= \sum_{i=1}^{m} \log\left(p(\mathbf{x}^{(i)}|\theta)\right) + \left(\mathbf{w}_{y^{(i)}}^T\mathbf{x}^{(i)} + b_{y^{(i)}}\right) - \log\left(\sum_{k=1}^{c} e^{\mathbf{w}_k^T\mathbf{x}^{(i)}+b_k}\right).
\end{aligned}$$

We can solve for the gradients of $\mathcal{L}$ w.r.t. the parameters $b_i, \mathbf{w}_i$ for $i \in [1, c]$:

$$\begin{aligned}
\nabla_{b_i}\mathcal{L} &= \nabla_{b_i}\left[\sum_{j=1}^{m} \log\left(p(\mathbf{x}^{(j)}|\theta)\right) + \left(\mathbf{w}_{y^{(j)}}^T\mathbf{x}^{(j)} + b_{y^{(j)}}\right) - \log\left(\sum_{k=1}^{c} e^{\mathbf{w}_k^T\mathbf{x}^{(j)}+b_k}\right)\right]\\
&= \nabla_{b_i}\left[\sum_{j=1}^{m} b_{y^{(j)}} - \log\left(\sum_{k=1}^{c} e^{\mathbf{w}_k^T\mathbf{x}^{(j)}+b_k}\right)\right]\\
&= \nabla_{b_i}\left[\sum_{j=1}^{m} b_{y^{(j)}} - \log\left(e^{\mathbf{w}_1^T\mathbf{x}^{(j)}+b_1} + \ldots + e^{\mathbf{w}_i^T\mathbf{x}^{(j)}+b_i} + \ldots + e^{\mathbf{w}_c^T\mathbf{x}^{(j)}+b_c}\right)\right]\\
&= \sum_{j=1}^{m}\left[\mathbb{1}_{\{y^{(j)}=i\}} - \frac{e^{\mathbf{w}_i^T\mathbf{x}^{(j)}+b_i}}{\sum_{k=1}^{c} e^{\mathbf{w}_k^T\mathbf{x}^{(j)}+b_k}}\right],
\end{aligned}$$

and

$$
\begin{aligned}
\nabla_{\mathbf{w}_i}\mathcal{L} &= \nabla_{\mathbf{b}_i}\left[\sum_{j=1}^{m}\log\left(p(\mathbf{x}^{(j)}|\theta)\right) + \left(\mathbf{w}_{y^{(j)}}^{T}\mathbf{x}^{(j)} + b_{y^{(j)}}\right) - \log\left(\sum_{k=1}^{c} e^{\mathbf{w}_k^T\mathbf{x}^{(j)}+b_k}\right)\right] \\
&= \nabla_{\mathbf{w}_i}\left[\sum_{j=1}^{m}\mathbf{w}_{y^{(j)}}^{T}\mathbf{x}^{(j)} - \log\left(\sum_{k=1}^{c} e^{\mathbf{w}_k^T\mathbf{x}^{(j)}+b_k}\right)\right] \\
&= \nabla_{\mathbf{w}_i}\left[\left(\sum_{j=1}^{m}\mathbf{w}_{y^{(j)}}^{T}\mathbf{x}^{(j)}\right) - \log\left(e^{\mathbf{w}_1^T\mathbf{x}^{(j)}+b_1} + \ldots + e^{\mathbf{w}_i^T\mathbf{x}^{(j)}+b_i} + \ldots + e^{\mathbf{w}_m^T\mathbf{x}^{(j)}+b_m}\right)\right] \\
&= \sum_{j=1}^{m}\left[\mathbb{1}_{\{y^{(j)}=i\}} - \frac{e^{\mathbf{w}_i^T\mathbf{x}^{(j)}+b_i}}{\sum_{k=1}^{c} e^{\mathbf{w}_k^T\mathbf{x}^{(j)}+b_k}}\right]\mathbf{x}^{(j)},
\end{aligned}
$$

where $\mathbb{1}$ is an indicator function.

Problem 4

We are given the average hinge loss function

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{K}\sum_{i=1}^{K} \text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}),$$

where the hinge loss per training sample is defined as

$$\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)}) = \max\Big(0, 1 - y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b)\Big).$$

We can solve for the gradients of $\mathcal{L}$ w.r.t. the parameters $b, \mathbf{w}$:

$$\begin{aligned}
\nabla_b \mathcal{L}(\mathbf{w}, b) &= \nabla_b \left[\frac{1}{K}\sum_{i=1}^{K} \text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)})\right] \\
&= \nabla_b \left[\frac{1}{K}\sum_{i=1}^{K} \max\Big(0, 1 - y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b)\Big)\right] \\
&= -\frac{1}{K}\sum_{i=1}^{K} \mathbb{1}_{\{\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)})>0\}} \cdot y^{(i)}
\end{aligned}$$

and

$$\begin{aligned}
\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b) &= \nabla_{\mathbf{w}} \left[\frac{1}{K}\sum_{i=1}^{K} \text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)})\right] \\
&= \nabla_{\mathbf{w}} \left[\frac{1}{K}\sum_{i=1}^{K} \max\Big(0, 1 - y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b)\Big)\right] \\
&= -\frac{1}{K}\sum_{i=1}^{K} \mathbb{1}_{\{\text{hinge}_{y^{(i)}}(\mathbf{x}^{(i)})>0\}} \cdot y^{(i)}\mathbf{x}^{(i)},
\end{aligned}$$

where $\mathbb{1}$ is an indicator function.

Problem 5

softmax.py:

```
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.)
    # ================================================================ #

    for Xs, ys in zip(X, y):
        loss += np.log(np.sum(np.exp(Xs @ self.W.T))) - self.W[ys].T @ Xs
    loss /= y.shape[0]

    # ================================================================ #
    # END YOUR CODE HERE
    # ================================================================ #
    return loss

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

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

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

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

    c, m = self.W.shape[0], y.shape[0]
    for Xs, ys in zip(X, y):
        loss += np.log(np.sum(np.exp(np.dot(Xs, self.W.T)))) - np.dot(self.W[ys].T, Xs)
    loss /= m

    A = self.W @ X.T
    for i in range(c):
        for j, (Xs, ys) in enumerate(zip(X, y)):
            ind = 1 if ys == i else 0
            grad[i, :] += (ind - np.exp(A[i, j]) / np.sum(np.exp(A[:, j]))) * Xs
    grad = -grad / m

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

    return loss, grad

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

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

      oldval = self.W[ix]
      self.W[ix] = oldval + h # increment by h
      fxph = self.loss(X, y)
      self.W[ix] = oldval - h # decrement by h
      fxmh = self.loss(X,y) # evaluate f(x - h)
      self.W[ix] = oldval # reset
      grad_numerical = (fxph - fxmh) / (2 * h)
      grad_analytic = your_grad[ix]
      rel_error = abs(grad_numerical - grad_analytic) / (abs(grad_numerical) + abs(grad_analytic))
      print('numerical: %f analytic: %f, relative error: %e' % (grad_numerical, grad_analytic, rel_erro

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

    c, m = self.W.shape[0], y.shape[0]
    A = X @ self.W.T
    loss += np.sum(np.log(np.sum(np.exp(X @ self.W.T), axis=1)))
    loss -= np.sum(A[np.arange(len(A)), y])
    loss /= m

    A = self.W @ X.T
    exp = np.repeat(np.array([np.sum(np.exp(A), axis=0)]), c, axis=0)
    ind = np.arange(c)[:, None] == y
    grad = (ind - np.exp(A) / exp) @ X
    grad /= -m

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

    return loss, grad

  def train(self, X, y, learning_rate=1e-3, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this linear classifier using stochastic gradient descent.

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

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

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

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

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

      # ================================================================ #
      # YOUR CODE HERE:
      #   Sample batch_size elements from the training data for use in
      #     gradient descent.  After sampling,
      #     - X_batch should have shape: (batch_size, dim)
      #     - 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.
      # ================================================================ #

      sample = np.random.choice(np.arange(num_train), size=batch_size)
      X_batch, y_batch = X[sample], y[sample]

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

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

      # ================================================================ #
      # YOUR CODE HERE:
      #   Update the parameters, self.W, with a gradient step
      # ================================================================ #

      self.W -= learning_rate * grad

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

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

    return loss_history

  def predict(self, X):
    """
    Inputs:
```

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

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

A = X @ self.W.T
c = self.W.shape[0]
exp = np.repeat(np.array([np.sum(np.exp(A), axis=1)]), c, axis=0).T
B = np.exp(A) / exp
y_pred = np.argmax(B, 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 [2]:

```
def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000, n
    """
    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_dat
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 nndl 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 ran

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:

The loss for a single sample x_i, y_i is given by $\log\left(\sum_{i=1}^{c} e^{a_{y^{(j)}}(x^{(i)})}\right) - a_{y^{(i)}}(x^{(i)})$. However, $\forall c, a_c(x^{(i)}) = 0$ because the weights were initialized to zeroes. Thus, the loss for this sample is $\log(10) - 0 \approx 2.3$. The average loss across all samples is also about 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 you
softmax.grad_check_sparse(X_dev, y_dev, grad)
```

```
numerical: -0.725555 analytic: -0.725555, relative error: 6.256188e-08
numerical: 0.814312 analytic: 0.814312, relative error: 2.461743e-08
numerical: 0.054375 analytic: 0.054375, relative error: 2.051445e-07
numerical: 0.913922 analytic: 0.913922, relative error: 2.312263e-09
numerical: 0.583014 analytic: 0.583014, relative error: 7.290683e-08
numerical: 1.163436 analytic: 1.163436, relative error: 9.776344e-10
numerical: -1.210180 analytic: -1.210180, relative error: 3.043019e-08
numerical: 0.812345 analytic: 0.812345, relative error: 2.285858e-08
numerical: 0.970107 analytic: 0.970107, relative error: 9.119929e-09
numerical: -2.675745 analytic: -2.675745, relative error: 1.786743e-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.linal

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

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

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

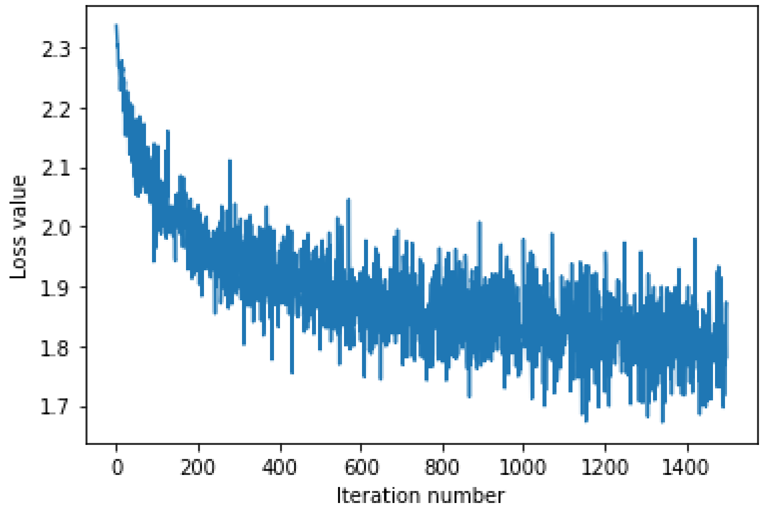
```
Normal loss / grad_norm: 2.3007035049524447 / 316.0883329661394 computed in 0.
07978391647338867s
Vectorized loss / grad: 2.3007035049524447 / 316.0883329661394 computed in 0.0
071680545806884766s
difference in loss / grad: 0.0 /2.1116644675360089e-13
```

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).

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 2.055722261385083
iteration 200 / 1500: loss 2.0357745120662813
iteration 300 / 1500: loss 1.9813348165609883
iteration 400 / 1500: loss 1.9583142443981612
iteration 500 / 1500: loss 1.8622653073541355
iteration 600 / 1500: loss 1.853261145435938
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 1.847079791353263
iteration 1200 / 1500: loss 1.8411450268664082
iteration 1300 / 1500: loss 1.7910402495792102
iteration 1400 / 1500: loss 1.8705803029382262
That took 6.11906099319458s
```



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

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

In [12]:

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

Out[12]:

```
2.220446049250313e-16
```

In [13]:

```
# ================================================================ #
# 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 erro
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #

learning_rates = [1e-9, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4]
training_accs = []
validation_accs = []

for lr in learning_rates:
    print(f'testing learning rate = {lr}')
    loss_hist = softmax.train(X_train, y_train, learning_rate=lr,
                      num_iters=1500, verbose=True)

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

    training_accs.append(training_acc)
    validation_accs.append(validation_acc)

i_opt = np.argmax(validation_accs)
print(f'Best learning rate = {learning_rates[i_opt]}')
print(f'Best validation accuracy = {validation_accs[i_opt]}')

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

```
testing learning rate = 1e-09
iteration 0 / 1500: loss 2.335383545089155
iteration 100 / 1500: loss 2.290307684881017
iteration 200 / 1500: loss 2.3046836107871793
iteration 300 / 1500: loss 2.3116427157905375
iteration 400 / 1500: loss 2.2910325322239014
iteration 500 / 1500: loss 2.26664217848507
iteration 600 / 1500: loss 2.2603303993352055
iteration 700 / 1500: loss 2.292917706392274
iteration 800 / 1500: loss 2.2408415553338314
iteration 900 / 1500: loss 2.2339458978340336
iteration 1000 / 1500: loss 2.2521555061624934
iteration 1100 / 1500: loss 2.2281985014544405
iteration 1200 / 1500: loss 2.2253761326579906
iteration 1300 / 1500: loss 2.2349257852214195
iteration 1400 / 1500: loss 2.205854436457119
training accuracy: 0.18277551020408164
validation accuracy: 0.181
============================
testing learning rate = 1e-08
iteration 0 / 1500: loss 2.461534698549716
iteration 100 / 1500: loss 2.3235347361836363
iteration 200 / 1500: loss 2.3068391442404983
iteration 300 / 1500: loss 2.18398852912162
iteration 400 / 1500: loss 2.1983886403372472
iteration 500 / 1500: loss 2.1590121263647473
iteration 600 / 1500: loss 2.2155857931274827
iteration 700 / 1500: loss 2.059508717536746
iteration 800 / 1500: loss 2.079568544534294
iteration 900 / 1500: loss 2.083140570997886
iteration 1000 / 1500: loss 2.1134412654769594
iteration 1100 / 1500: loss 2.0481602224153694
iteration 1200 / 1500: loss 2.0819298801747625
iteration 1300 / 1500: loss 2.08840800333523
iteration 1400 / 1500: loss 1.992407453290292
training accuracy: 0.2889591836734694
validation accuracy: 0.281
============================
testing learning rate = 1e-07
iteration 0 / 1500: loss 2.3790388831757823
iteration 100 / 1500: loss 2.021765014137576
iteration 200 / 1500: loss 2.019535925434741
iteration 300 / 1500: loss 1.9657164929328355
iteration 400 / 1500: loss 1.949176442001835
iteration 500 / 1500: loss 1.8777026252392124
iteration 600 / 1500: loss 1.7600982072534423
iteration 700 / 1500: loss 1.7979690392719154
iteration 800 / 1500: loss 1.7865301980842139
iteration 900 / 1500: loss 1.8773422082822286
iteration 1000 / 1500: loss 1.8884035413970162
iteration 1100 / 1500: loss 1.8407999060483053
iteration 1200 / 1500: loss 1.7620842039331626
iteration 1300 / 1500: loss 1.8525733340180983
iteration 1400 / 1500: loss 1.8450029538557828
training accuracy: 0.3816530612244898
validation accuracy: 0.398
============================
testing learning rate = 1e-06
iteration 0 / 1500: loss 2.338799247622096
iteration 100 / 1500: loss 1.7818084210560432
iteration 200 / 1500: loss 1.7952543254925757
iteration 300 / 1500: loss 1.7842238060601288
iteration 400 / 1500: loss 1.6905275374959206
iteration 500 / 1500: loss 1.7427153070566361
iteration 600 / 1500: loss 1.883644127038289
iteration 700 / 1500: loss 1.7794067821062862
iteration 800 / 1500: loss 1.7612266103626752
iteration 900 / 1500: loss 1.7845200090125812
iteration 1000 / 1500: loss 1.677484226154133
iteration 1100 / 1500: loss 1.745357310826754
iteration 1200 / 1500: loss 1.671043661661188
iteration 1300 / 1500: loss 1.5705155182884227
iteration 1400 / 1500: loss 1.8135240459679844
training accuracy: 0.42248979591836733
validation accuracy: 0.411
============================
testing learning rate = 1e-05
iteration 0 / 1500: loss 2.372474398135772
iteration 100 / 1500: loss 2.746763626832451
iteration 200 / 1500: loss 2.762840192886496
iteration 300 / 1500: loss 3.6320437393723735
iteration 400 / 1500: loss 3.1469483662553226
iteration 500 / 1500: loss 2.72847503425027
iteration 600 / 1500: loss 2.247140558957927
iteration 700 / 1500: loss 2.10612621766827
iteration 800 / 1500: loss 2.5727614605365137
iteration 900 / 1500: loss 2.719212383488865
iteration 1000 / 1500: loss 2.9356784661682527
iteration 1100 / 1500: loss 2.4693571722740644
iteration 1200 / 1500: loss 3.245680697004521
iteration 1300 / 1500: loss 2.850313415033499
iteration 1400 / 1500: loss 2.628896506058373
training accuracy: 0.3109387755102041
validation accuracy: 0.287
============================
testing learning rate = 0.0001
iteration 0 / 1500: loss 2.352610559430411
iteration 100 / 1500: loss 34.954085702597055
iteration 200 / 1500: loss 36.24954095131478
iteration 300 / 1500: loss 22.35673908317269
iteration 400 / 1500: loss 26.33526586147925
iteration 500 / 1500: loss 28.236269113026818
iteration 600 / 1500: loss 13.613559072782623
iteration 700 / 1500: loss 38.25388506459891
iteration 800 / 1500: loss 34.84834814745934
iteration 900 / 1500: loss 28.792825896923187
iteration 1000 / 1500: loss 33.72950838481193
iteration 1100 / 1500: loss 34.84499414300405
iteration 1200 / 1500: loss 25.87341796098307
iteration 1300 / 1500: loss 21.389529997010367
iteration 1400 / 1500: loss 19.52189829615041
training accuracy: 0.2483469387755102
validation accuracy: 0.235
============================
Best learning rate = 1e-06
Best validation accuracy = 0.411
```

The best learning rate was $1e-6$ with a corresponding validation accuracy of 0.411.

```
In [ ]:
```