Margaret Capetz
147 HW2

1) Noisy linear regression.

$D = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(N)}, y^{(N)})\}$

dataset

$x^{(i)} \in \mathbb{R}^d$ feature vector of i^{th} house $\quad \theta \in \mathbb{R}^d$ parameter vector

$y^{(i)} \in \mathbb{R}$ is the house price

parameter regularization

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - x^{(i)T}\theta)^2$$

Add zero-mean gaussian noise of known variance σ^2 from the distribution

$\mathcal{N}(0, \sigma^2 I)$, $\sigma \in \mathbb{R}$, $I \in \mathbb{R}^{d \times d}$

$$\tilde{\mathcal{L}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta)^2$$

a) Express the expectation of modified loss, where R is not a function of D

$\mathbb{E}_{\delta \sim N}[\tilde{\mathcal{L}}(\theta)] = \mathcal{L}(\theta) + R$

Note $\mathbb{E}_{\delta \sim N}[\delta\delta^T] = \sigma^2 I$

$$\mathbb{E}_{\delta \sim N}[\tilde{\mathcal{L}}(\theta)] = \mathbb{E}\left(\frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta)^2\right) \quad *\mathbb{E} \text{ is linear operator}$$

"varying δ over $\mathcal{N}$ distribution, then finding avg"

$$= \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}\left((y^{(i)} - (x^{(i)} + \delta^{(i)})^T\theta)^2\right)$$

can either multiply out the squared, or since it is a scalar can do $A^2 = A^TA$

$$= \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}\left((y^{(i)} - (x^{(i)T} + \delta^{(i)T})\theta)^2\right)$$

δ is zero mean gaussian variable, then $\mathbb{E}(\delta) = \mathbb{E}(\delta^T) = 0$

$$= \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}\left((y^{(i)} - x^{(i)T}\theta + \delta^{(i)T}\theta)^2\right)$$

$$= \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}\left((y^{(i)} - x^{(i)T}\theta)^2 + 2(y^{(i)} - x^{(i)T}\theta)(\delta^{(i)T}\theta) + (\delta^{(i)T}\theta)^2\right)$$

$$= \mathcal{L}(\theta) + \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}\left(2(y^{(i)} - x^{(i)T}\theta)(\delta^{(i)T}\theta) + (\delta^{(i)T}\theta)^2\right)$$

$(\delta^{(i)T}\theta) = 0$

don't depend on δ, so constants

$$= \mathcal{L}(\theta) + \frac{1}{N}\sum_{i=1}^{N} \mathbb{E}\left((\delta^{(i)T}\theta)^T (\delta^{(i)T}\theta) \right)$$

$$= \mathcal{L}(\theta) + \frac{1}{N}\sum_{i=1}^{N} \mathbb{E}\left(\theta^T \delta^{(i)} \delta^{(i)T} \theta \right)$$

$$= \mathcal{L}(\theta) + \frac{1}{N}\sum_{i=1}^{N} \theta^T \mathbb{E}\left(\delta^{(i)} \delta^{(i)T} \right) \theta$$

$$= \mathcal{L}(\theta) + \frac{1}{N}\sum_{i=1}^{N} \theta^T \sigma^2 I \theta$$

$$= \mathcal{L}(\theta) + \frac{\sigma^2}{N}\sum_{i=1}^{N} \theta^T\theta$$

$$= \mathcal{L}(\theta) + \frac{\sigma^2}{N}(N \cdot \theta^T\theta)$$

$$\mathbb{E}_{\delta \sim N}[\tilde{\mathcal{L}}(\theta)] = \mathcal{L}(\theta) + \sigma^2 \theta^T\theta.$$

L1 Norm: $\mathcal{L}(\theta) + \lambda \Sigma |W_i|$

L2 Norm: $\mathcal{L}(\theta) + \lambda \Sigma W_i^2$

$$\|W\|_1 = \sum_{i=1}^{p} |W_i|$$

$$\|W\|_2 = \left(\sum_{i=1}^{p} W_i^2 \right)^{1/2}$$

$\theta^T\theta = \|\theta\|_2^2$

L1, L2, mix of L1, L2

b) Under expectation what regularization effect would the addition of the noise have on the model?

L2 Normalization.

c) suppose $\sigma \to 0$, less and less effect of regularization → more overfitting

d) suppose $\sigma \to \infty$, greater and greater effect of regularization → more underfitting
weights drop to zero, no learning

2) K-nearest neighbors — see Jupyter NB

3) softmax classifier gradient derivation

$(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$ where $x^{(j)} \in \mathbb{R}^n$, $y^{(i)} = \{1, \ldots, c\}$, $j = 1, \ldots, m$

parameters $\theta = \{ W_i, b_i \}_{i=1,\ldots,c}$

probablistic model is

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

$$\text{softmax}_i(x) = \frac{e^{W_i^T x + b_i}}{\sum_{k=1}^{c} e^{W_k^T x + b_k}}$$

Derive the log likelihood $\mathcal{L}$, and its gradient w.r.t. the parameters $\nabla_{W_i}\mathcal{L}$, $\nabla_{b_i}\mathcal{L}$ for $i = 1, \ldots, c$

Note: $\tilde{x} = \begin{bmatrix} x \\ 1 \end{bmatrix}$, $\tilde{w}_i = \begin{bmatrix} w_i \\ b_i \end{bmatrix}$, then $a_i(x) = w_i^T x + b_i = \tilde{w}_i^T \tilde{x}$

this unifies $\nabla_{w_i} \mathcal{L}$ and $\nabla_{b_i} \mathcal{L}$ into $\nabla_{\tilde{w}_i} \mathcal{L}$

First, take the log of the softmax.

$$\log \text{softmax}_i(\tilde{x}) = \log\left(\frac{e^{\tilde{w}_i^T \tilde{x}}}{\sum_{k=1}^{c} e^{\tilde{w}_k^T \tilde{x}}}\right) = \tilde{w}_i^T \tilde{x} - \log\left(\sum_{k=1}^{c} e^{\tilde{w}_k^T \tilde{x}}\right)$$

softmax

No need for vectorization

gradient w.r.t. $\tilde{w}_i$

$\nabla_{\tilde{w}_i} \mathcal{L}$

Now, find the gradient.

$$\nabla_{\tilde{w}_i} \mathcal{L} = \frac{\partial}{\partial \tilde{w}_i}\left(\tilde{w}_{y^{(j)}}^T \tilde{x} - \log\left(\sum_{k=1}^{c} e^{\tilde{w}_k^T \tilde{x}}\right)\right)$$

$$= \frac{\partial}{\partial \tilde{w}_i}\left(\tilde{w}_{y^{(j)}}^T \tilde{x}\right) - \frac{\partial}{\partial \tilde{w}_i} \log\left(\sum_{k=1}^{c} e^{\tilde{w}_k^T \tilde{x}}\right)$$

$$= \frac{\partial}{\partial \tilde{w}_i}\left(\tilde{w}_{y^{(j)}}^T \tilde{x}\right) - \frac{\partial}{\partial \tilde{w}_i} \log\left(\sum_{k=1}^{i-1} e^{\tilde{w}_k^T \tilde{x}} + e^{\tilde{w}_i^T \tilde{x}} + \sum_{k=i+1}^{c} e^{\tilde{w}_k^T \tilde{x}}\right)$$

$$= \frac{\partial}{\partial \tilde{w}_i}\left(\tilde{w}_{y^{(j)}}^T \tilde{x}\right) - \frac{\partial}{\partial \tilde{w}_i} \log\left(\sum_{k=1}^{i-1} e^{\tilde{w}_k^T \tilde{x}} + e^{\tilde{w}_i^T \tilde{x}} + \sum_{k=i+1}^{c} e^{\tilde{w}_k^T \tilde{x}}\right)$$

when $y^{(j)} = i$: $\nabla_{\tilde{w}_i} \mathcal{L} = \tilde{x} - \frac{e^{\tilde{w}_i^T \tilde{x}} \tilde{x}}{\left(\sum_{k=1}^{i-1} e^{\tilde{w}_k^T \tilde{x}} + e^{\tilde{w}_i^T \tilde{x}} + \sum_{k=i+1}^{c} e^{\tilde{w}_k^T \tilde{x}}\right)}$

A

Derivative Review:

$$\frac{\partial}{\partial x} \log f(x) = \frac{f'(x)}{x}$$

$$\frac{\partial}{\partial x} e^{f(x)} = e^{f(x)} f'(x)$$

else : $\nabla_{\tilde{w}_i} \mathcal{L} = 0$ -A

when $y^{(j)} = i$: $\nabla_{\tilde{w}_i} \mathcal{L} = \tilde{x} - \frac{e^{\tilde{w}_i^T \tilde{x}} \tilde{x}}{\sum_{k=1}^{c} e^{\tilde{w}_k^T \tilde{x}}}$

else : $\nabla_{\tilde{w}_i} \mathcal{L} = -\frac{e^{\tilde{w}_i^T \tilde{x}} \tilde{x}}{\sum_{k=1}^{c} e^{\tilde{w}_k^T \tilde{x}}}$

gradient of softmax

4) Hinge loss gradient

SVM: support vector machine.

dataset: $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(K)}, y^{(K)})\}$

where $x^{(i)} \in \mathbb{R}^d$ a feature, $y^{(i)} \in \{-1, 1\}$, $y^{(i)}$ is a scalar

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

$w \in \mathbb{R}^d$, bias $b \in \mathbb{R}$

Formulate avg loss of model as:

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

find $\nabla_w \mathcal{L}(w, b)$ and $\nabla_b \mathcal{L}(w, b)$

Hint: indicator function: 1 at designated points, 0 at all others

$\mathbb{1}_{\{p<1\}} = \begin{cases} 1, & \text{if } p < 1 \\ 0, & \text{otherwise} \end{cases}$

a)

$\nabla_w \mathcal{L}(w, b)$:

$$\mathcal{L}(w, b) = \begin{cases} 1 - y^{(i)}(w^T x^{(i)} + b), & \text{if } (1 - y^{(i)}(w^T x^{(i)} + b)) > 0 \\ 0, & \text{otherwise} \end{cases}$$

Take derivative separately:

$$\frac{\partial}{\partial w} (1 - y^{(i)}(w^T x^{(i)} + b))$$

$$= \frac{\partial}{\partial w} (1 - y^{(i)} w^T x^{(i)} - y^{(i)} b)$$

Matrix cookbook: $\frac{\partial a^T X^T b}{\partial X} = ba^T$

$$= 0 - x^{(i)} y^{(i)} - 0$$

$$= -x^{(i)} y^{(i)}$$

$$\frac{\partial}{\partial w}(0) = 0.$$

$$\Rightarrow \quad \nabla_w \mathcal{L}(w, b) = \begin{cases} -x^{(i)} y^{(i)}, & \text{if } (1 - y^{(i)}(w^T x^{(i)} + b)) > 0 \\ 0, & \text{otherwise} \end{cases}$$

A B

$\nabla_b \mathcal{L}(w,b)$:

$$\frac{\partial}{\partial b}\left(1 - y^{(i)}(w^T x^{(i)} + b)\right)$$

$$= \frac{\partial}{\partial b}\left(1 - y^{(i)} w^T x^{(i)} - y^{(i)} b\right)$$

$$= 0 + 0 - y^{(i)} \quad \text{since } y^{(i)} \text{ is a scalar.}$$

$$\frac{\partial}{\partial b}(0) = 0$$

$$\nabla_b \mathcal{L}(w,b) = \begin{cases} -y^{(i)} \; (C) & \text{, if } \left(1 - y^{(i)}(w^T x^{(i)} + b)\right) > 0 \\ 0 & \text{, otherwise} \end{cases}$$

$$\nabla_w \mathcal{L}(w,b) = A \cdot \mathbb{1}(B)$$

$$\nabla_b \mathcal{L}(w,b) = C \cdot \mathbb{1}(B)$$

$$\nabla_w \mathcal{L}(w,b) = -x^{(i)} y^{(i)} \cdot \mathbb{1}\left(1 - y^{(i)}(w^T x^{(i)} + b)\right)$$

$$\nabla_b \mathcal{L}(w,b) = -y^{(i)} \cdot \mathbb{1}\left(1 - y^{(i)}(w^T x^{(i)} + b)\right)$$

5) see jupyter notebook.

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

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

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

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

    # add bias dimension and transform into columns
    X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
```

```
    X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])
    X_test = np.hstack([X_test, np.ones((X_test.shape[0], 1))])
    X_dev = np.hstack([X_dev, np.ones((X_dev.shape[0], 1))])

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


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

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

Training a softmax classifier.

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

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

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

np.random.seed(1)

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

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

Softmax loss

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

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

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

```
2.327760702804897
```

Question:

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

Answer:

We take the log likelihood of 0.10 because the weights are randomly initialized, we take -log(0.10) which is close to 2.3, which makes sense.

Softmax gradient

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

```
for loop loss 2.3383493557745996
numerical: -0.542683 analytic: -0.542683, relative error: 6.983491e-08
numerical: -0.293379 analytic: -0.293379, relative error: 5.650900e-09
numerical: -0.268456 analytic: -0.268456, relative error: 9.019206e-08
numerical: 1.603418 analytic: 1.603418, relative error: 1.557917e-08
numerical: 0.957375 analytic: 0.957375, relative error: 1.202059e-08
numerical: 2.325052 analytic: 2.325052, relative error: 4.799397e-10
numerical: -0.072500 analytic: -0.072500, relative error: 4.240652e-07
numerical: 0.614202 analytic: 0.614202, relative error: 3.688366e-08
numerical: -0.597377 analytic: -0.597377, relative error: 1.051529e-09
numerical: -2.759162 analytic: -2.759162, relative error: 2.466011e-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 [ ]:
```
import time
```

In [ ]:
```
## 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)
#print(grad.shape)
toc = time.time()
print('Normal loss / grad_norm: {} / {} computed in {}s'.format(loss, np.linalg.norm(gra

tic = time.time()
```

```
loss_vectorized, grad_vectorized = softmax.fast_loss_and_grad(X_dev, y_dev)
print("****",loss_vectorized.shape, grad_vectorized.shape)
toc = time.time()
print('Vectorized loss / grad: {} / {} computed in {}s'.format(loss_vectorized, np.linal

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

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

```
for loop loss 2.3383493557745996
Normal loss / grad_norm: 2.3383493557745996 / 341.31254890164223 computed in 0.038594961
166381836s
**** () (10, 3073)
Vectorized loss / grad: 2.3383493557745996 / 341.31254890164223 computed in 0.0061707496
64306641s
difference in loss / grad: 0.0 /2.2885802908952907e-13
```

Stochastic gradient descent

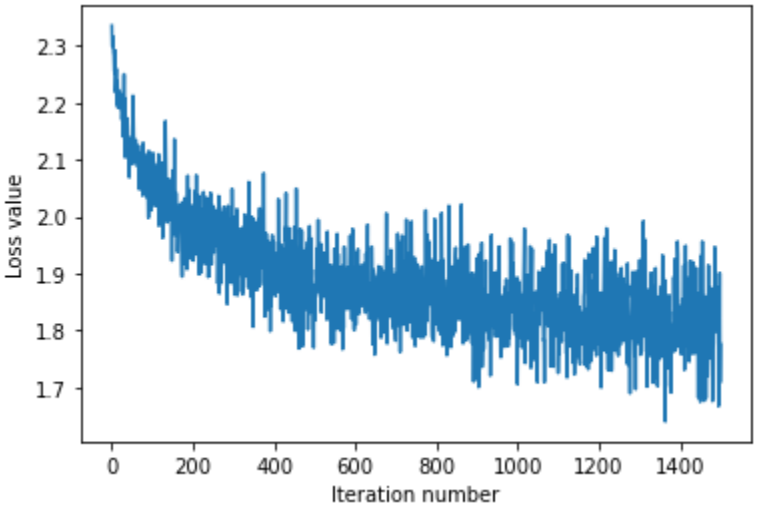
We now implement stochastic gradient descent. This uses the same principles of gradient descent we discussed in class, however, it calculates the gradient by only using examples from a subset of the training set (so each gradient calculation is faster).

In [ ]:
```
# 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 2.335383545089155
iteration 100 / 1500: loss 2.0225093946317187
iteration 200 / 1500: loss 1.982172871654982
iteration 300 / 1500: loss 1.9356442081331482
iteration 400 / 1500: loss 1.882893396815689
iteration 500 / 1500: loss 1.8181869697394497
iteration 600 / 1500: loss 1.874513153185746
iteration 700 / 1500: loss 1.836183250017359
iteration 800 / 1500: loss 1.8584086819212182
iteration 900 / 1500: loss 1.9275087067564147
iteration 1000 / 1500: loss 1.824667969507725
iteration 1100 / 1500: loss 1.7731817984393603
iteration 1200 / 1500: loss 1.8636308568113116
iteration 1300 / 1500: loss 1.924074621260815
iteration 1400 / 1500: loss 1.7846918635831293
That took 5.443572998046875s
```



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

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

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

```
training accuracy: 0.37881632653061226
validation accuracy: 0.39
```

Optimize the softmax classifier

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

In [ ]:
```
# ================================================================ #
# YOUR CODE HERE:
#   Train the Softmax classifier with different learning rates and
#     evaluate on the validation data.
#   Report:
#     - The best learning rate of the ones you tested.
#     - The best validation accuracy corresponding to the best validation error.
#
#   Select the SVM that achieved the best validation error and report
#     its error rate on the test set.
# ================================================================ #

rates = [1e-5, 1e-6, 1e-7]

print("rates")

for rate in rates:
    print(rate)

    loss_hist = softmax.train(X_train, y_train, learning_rate=rate,num_iters=1500, verbo
    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)), ))
```

```
iters = [1400, 1500, 1600, 1700, 1800]

print("iters")

for iter in iters:
    print(iter)
    loss_hist = softmax.train(X_train, y_train, learning_rate=rate,num_iters=iter, verbo
    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)), ))

print("ideal")
loss_hist = softmax.train(X_train, y_train, learning_rate=1e-6,num_iters=1700, verbose=F
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)), ))

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

```
rates
1e-05
training accuracy: 0.3072448979591837
validation accuracy: 0.303
1e-06
training accuracy: 0.4217142857142857
validation accuracy: 0.399
1e-07
training accuracy: 0.37951020408163266
validation accuracy: 0.39
iters
1400
training accuracy: 0.38312244897959186
validation accuracy: 0.389
1500
training accuracy: 0.38116326530612243
validation accuracy: 0.389
1600
training accuracy: 0.3826530612244898
validation accuracy: 0.392
1700
training accuracy: 0.38412244897959186
validation accuracy: 0.403
1800
training accuracy: 0.3859591836734694
validation accuracy: 0.397
ideal
training accuracy: 0.4199591836734694
validation accuracy: 0.402
```

Best learning rate: 1e-6 Best num iterations: 1700

Taken together, I got a training accuracy: 0.41996 validation accuracy: 0.402

```
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
```

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


    # print(X.shape)
```

```
    # print(self.W.T.shape)
    ex_sc = 0

    for i in range (X.shape[0]):
      e_x = np.exp(np.matmul(self.W, X[i])) # exp(wT * X) <-- exp of
score of each class

      ex_sc = e_x # sum of softmax scores, for the denominator

      ex_sc = ex_sc/(np.sum(ex_sc)) #

      log_sc = np.log(ex_sc) # take the log likelihood

      loss = loss - log_sc[y[i]]

    loss = loss/(X.shape[0])

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

    return loss

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

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

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

    # ================================================================
#
    # YOUR CODE HERE:
    #   Calculate the softmax loss and the gradient. Store the
gradient
    #   as the variable grad.
    # ================================================================
#
    # ex_sc = 0
    # grad_sum = 0
    # smax_all = [X.shape[0]][self.W.shape[0]] # n x c
    # for i in range (X.shape[0]):
    #   e_x = np.exp(np.matmul(self.W, X[i])) # exp(wT * X) <-- exp of
```

```
score of each class
    #   ex_sc = e_x/(np.sum(e_x)) # normalize the scores - becomes row
of softmax
    #   smax_all[i] = ex_sc

    #   ex_sc += e_x # sum of softmax scores, for the denominator

    # print("x shape", X.shape) # 500
    # print("w shape", self.W.shape) # 10

    for i in range (X.shape[0]): # i is range 500
      e_x = np.exp(np.matmul(self.W, X[i])) # exp(wT * X) <-- exp of
score of each class (numerator)
      # print("e_x shape", e_x.shape[0]) # 500
      ex_sum = e_x/np.sum(e_x, axis=0) # ex/sum ex (so this adds in
the denominator)
      # print(np.shape(ex_sum))
      log_sc = np.log(ex_sum) # take the log likelihood

      loss = loss - log_sc[y[i]]

      for j in range (self.W.shape[0]): # j is range 20
        if (j == y[i]):
          grad[j] += (ex_sum[j]-1)*X[i]
        else:
          grad[j] += (ex_sum[j])*X[i]

    # print("ex sum", ex_sum)
    # print("loss before normalization", loss)
    # print("log sc", log_sc)
    grad = grad/(X.shape[0]) # normalize the grad
    loss = loss/(X.shape[0])
    print("for loop loss", loss)
    # print(grad.shape)



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

    return loss, grad

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

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

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

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

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

    # ============================================================
#
    # YOUR CODE HERE:
    #   Calculate the softmax loss and gradient WITHOUT any for loops.
    # ============================================================
#
    e_x = np.exp(X.dot(self.W.T)) # exp(wT * X) <-- exp of score of
each class (numerator)
    ex_sum = e_x/np.sum(e_x, axis=1, keepdims=True) # ex/sum ex (so
this adds in the denominator) output is (10,)
    # print("ex_sum shape", ex_sum.shape)
    log_sc = np.log(ex_sum) # take the log likelihood

    log_sum = np.sum(log_sc[np.arange(X.shape[0]),y])
    loss = -log_sum
    loss = loss/(X.shape[0]) # normalize the loss
    # print("fast loss", loss)

    grad = (ex_sum).T.dot(X) # this is the default, otherwise if i ==
j then subtract X[i]
    # HOW TO DO MASKING?? for loss when i = j? use np arrange?
    mask_zero=np.zeros_like(ex_sum)
    mask_zero[np.arange(X.shape[0]),y]=1
    mask =  ex_sum-mask_zero # mask 500, 10
```

```
    # print("mask:", mask.shape)
    masked_grad = mask.T.dot(X) # X 500, 3072
    # grad -= masked_grad
    grad = masked_grad/(X.shape[0]) # normalize the grad  # 10, 3073
    # print("grad***", grad.shape)

    # ================================================================
#
    # 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.
      #
========================================================================= #
      # X_batch = [batch_size][dim]
      # y_batch = [batch_size]

      # rand_list = np.random.choice(X.shape[0], batch_size)
      # for i in rand_list:
      #   X_batch.append(X[i])
      # rand_list = np.random.choice(y.shape[0], batch_size)
      # for i in rand_list:
      #   y_batch.append(y[i])

      b_samples = []
      b_labels = []

      for i in range(batch_size):
        index = np.random.choice(num_train)
        b_samples.append(X[index])
        b_labels.append(y[index])

      X_batch = np.array(b_samples)
      y_batch = np.array(b_labels)

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

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

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

```
      #
========================================================== #
      self.W = self.W - grad * learning_rate

      #
========================================================== #
      # 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[0])

    # ========================================================================
#
    # YOUR CODE HERE:
    #   Predict the labels given the training data.
    # ========================================================================
#
    # print("ypred shape", y_pred.shape)
    # X = X[:3073]
    p = (self.W @ X.T).T
    # print("pshape", p.shape)
    # print("xshape", X.shape)
    # print("wshape", self.W.shape)

    for i in range(y_pred.shape[0]):
      y_pred[i] = np.argmax(p[i])
    # ========================================================================
#
    # END YOUR CODE HERE
    # ========================================================================
```

```
#
    return y_pred
```

```
        #   training point using norm(), and store the result in
dists[i, j].
        #
=========================================================== #
        # X is x-test
        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.
    # ===================================================================
#
```

```
    # broadcasting: kind of like amplifying the array

    # axis = 1 means sum the cols
    # reshape(-1, 1): (-1) takes the shape of the last dimension, puts
1 as that dimension
    # (2, ) -> (2, 1)

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


    pass

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

    return dists


  def predict_labels(self, dists, k=1):
    """
    Given a matrix of distances between test points and training
points,
    predict a label for each test point.

    Inputs:
    - dists: A numpy array of shape (num_test, num_train) where
dists[i, j]
      gives the distance betwen the ith test point and the jth
training point.

    Returns:
    - y: A numpy array of shape (num_test,) containing predicted
labels for the
      test data, where y[i] is the predicted label for the test point
X[i].
    """
    num_test = dists.shape[0]
    y_pred = np.zeros(num_test)
    for i in np.arange(num_test):
      # A list of length k storing the labels of the k nearest
neighbors to
      # the ith test point.
      closest_y = []
      #
========================================================================= #
```

```
      # YOUR CODE HERE:
      #   Use the distances to calculate and then store the labels of
      #   the k-nearest neighbors to the ith test point.  The function
      #   numpy.argsort may be useful.
      #
      #   After doing this, find the most common label of the k-
nearest
      #   neighbors.  Store the predicted label of the ith training
example
      #   as y_pred[i].  Break ties by choosing the smaller label.
      #
========================================================== #
      sorted_i = np.argsort(dists[i]) # sort the distances, the array
elements are the j's
      closest_indices = sorted_i[:k] # take only the first k elements
- but how to change the j to the label? ????

      # print(closest_indices)
      closest_y = self.y_train[closest_indices]
      # for j in closest_indices:
      #   labels.append(self.y_train[j])
      # print("i got here")
      y_pred[i] = np.bincount(closest_y).argmax() # save the most
frequent element


      pass

      #
========================================================== #
      # 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 [ ]:
```
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 dataset.

# Load matplotlib images inline
%matplotlib inline

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

np.random.seed(0)
```

In [ ]:
```
# Set the path to the CIFAR-10 data
cifar10_dir = '/Users/mcapetz/Downloads/cifar-10-batches-py' # You need to update this
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 [ ]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
```

```
        if i == 0:
            plt.title(cls)
plt.show()
```

In [ ]:

```
# 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 [ ]:

```
# Import the KNN class

from nndl import KNN
```

In [ ]:

```
# 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) Assigns X_train, y_train to their respective variables for this instance of the knn class.

(2) Pro: you can swap in any x, y training data easily, including testing data. Con: extra call of a function that affects computation time and memory.

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

import time
time_start =time.time()

dists_L2 = knn.compute_distances(X=X_test)

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

```
Time to run code: 13.994518995285034
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 [ ]:
```
# Implement the function compute_L2_distances_vectorized() in the KNN class.
# In this function, you ought to achieve the same L2 distance but WITHOUT any for loops
# Note, this is SPECIFIC for the L2 norm.

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

```
Time to run code: 0.1777641773223877
Difference in L2 distances between your KNN implementations (should be 0): 0.0
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 [ ]:
```
# Implement the function predict_labels in the KNN class.
# Calculate the training error (num_incorrect / total_samples)
#   from running knn.predict_labels with k=1

error = 1

# ================================================================ #
# YOUR CODE HERE:
#   Calculate the error rate by calling predict_labels on the test
#   data with k = 1.  Store the error rate in the variable error.
# ================================================================ #
y_pred = knn.predict_labels(dists_L2_vectorized)

error = np.sum(y_pred != y_test)/y_pred.shape
# ================================================================ #
# 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 [ ]:
```
# Create the dataset folds for cross-valdiation.
num_folds = 5

X_train_folds = [] # 5000 total so 1000 each
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]
# ============================================================ #

rand_indices = np.random.permutation(X_train.shape[0])
fold_size = int(X_train.shape[0]/num_folds)
# print(X_train.shape[0])
# print(fold_size)

i = 0
while i < X_train.shape[0]:
    # print("i", i)
    X_train_folds.append(X_train[rand_indices[i:i+fold_size]])
    y_train_folds.append(y_train[rand_indices[i:i+fold_size]])
    i += fold_size


# print(X_train.shape)
# for i in range(5):
#     print(X_train_folds[i].shape)


# print(X_train_folds[1].shape)



# ============================================================ #
# 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 [ ]:

```
time_start =time.time()

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

# ============================================================ #
# 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()

avg_errors = []

for k in ks:
    error_fold = []
```

```
    # create the folds
    for i in range(num_folds):
        # np.concatenate
        X_test_current = X_train_folds[i]
        y_test_current = y_train_folds[i]
        if i == 0:
            X_train_rest = np.array(X_train_folds[i+1:])
            X_train_rest = X_train_rest.reshape((X_train_rest.shape[0]*X_train_rest.shap
            # print("i = 0", X_train_rest.shape)
            y_train_rest = np.array(y_train_folds[i+1:])
            y_train_rest = y_train_rest.reshape((y_train_rest.shape[0]*y_train_rest.shap
            # print("y train shape", y_train_rest.shape)
        elif i == num_folds - 1:
            X_train_rest = np.array(X_train_folds[:i])
            X_train_rest = X_train_rest.reshape((X_train_rest.shape[0]*X_train_rest.shap
            # print("i = 5", X_train_rest.shape)
            y_train_rest = np.array(y_train_folds[:i])
            y_train_rest = y_train_rest.reshape((y_train_rest.shape[0]*y_train_rest.shap

        else:
            X_train_rest = np.concatenate((np.array(X_train_folds[i+1:]), np.array(X_tra
            X_train_rest = X_train_rest.reshape((X_train_rest.shape[0]*X_train_rest.shap
            # print("i = ", i, X_train_rest.shape)
            y_train_rest = np.concatenate((np.array(y_train_folds[i+1:]), np.array(y_tra
            y_train_rest = y_train_rest.reshape((y_train_rest.shape[0]*y_train_rest.shap
            # print("y train shape", y_train_rest.shape)

        # print("i", i)
        # print(np.array(X_train_folds[i+1:]).shape)
        # print(np.array(X_train_folds[:i]).shape)

        # run the knn
        knn.train(X=X_train_rest, y=y_train_rest)
        # print(X_train_rest.shape)
        dists_L2_vectorized = knn.compute_L2_distances_vectorized(X=X_test_current)
        y_pred = knn.predict_labels(dists_L2_vectorized, k=k)

        # add to the error
        error_fold.append(np.sum(y_pred != y_test_current)/y_pred.shape)

    # take the average error from all folds for this k
    avg_errors.append(np.average(error_fold))

print(avg_errors)


# test example
# m = [1, 2, 3, 4, 5]
# for i in range(num_folds):
#     print(i)
#     print("m:", m[i])
#     print(np.concatenate(np.array(m[i+1:]), np.array(m[:i])))

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

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

```
[0.7323999999999999, 0.7578, 0.7466000000000002, 0.726, 0.7304, 0.7224, 0.72539999999999
99, 0.7272000000000001, 0.7253999999999999, 0.7298]
Computation time: 25.79
```

Questions:

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

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

Answers:

(1) k = 10

(2) 0.7224

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 [ ]:

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

knn = KNN()

avg_errors = []

for norm in norms:
    error_fold = []
    # create the folds
    for i in range(num_folds):
        # np.concatenate
        X_test_current = X_train_folds[i]
        y_test_current = y_train_folds[i]
        if i == 0:
            X_train_rest = np.array(X_train_folds[i+1:])
            X_train_rest = X_train_rest.reshape((X_train_rest.shape[0]*X_train_rest.shap
            # print("i = 0", X_train_rest.shape)
            y_train_rest = np.array(y_train_folds[i+1:])
            y_train_rest = y_train_rest.reshape((y_train_rest.shape[0]*y_train_rest.shap
            # print("y train shape", y_train_rest.shape)
        elif i == num_folds - 1:
            X_train_rest = np.array(X_train_folds[:i])
```

```
            X_train_rest = X_train_rest.reshape((X_train_rest.shape[0]*X_train_rest.shap
            # print("i = 5", X_train_rest.shape)
            y_train_rest = np.array(y_train_folds[:i])
            y_train_rest = y_train_rest.reshape((y_train_rest.shape[0]*y_train_rest.shap

        else:
            X_train_rest = np.concatenate((np.array(X_train_folds[i+1:]), np.array(X_tra
            X_train_rest = X_train_rest.reshape((X_train_rest.shape[0]*X_train_rest.shap
            # print("i = ", i, X_train_rest.shape)
            y_train_rest = np.concatenate((np.array(y_train_folds[i+1:]), np.array(y_tra
            y_train_rest = y_train_rest.reshape((y_train_rest.shape[0]*y_train_rest.shap
            # print("y train shape", y_train_rest.shape)

        # run the knn
        knn.train(X=X_train_rest, y=y_train_rest)
        # print(X_train_rest.shape)
        dists_L2_vectorized = knn.compute_distances(X=X_test_current, norm=norm)
        y_pred = knn.predict_labels(dists_L2_vectorized, k=5)

        # add to the error
        error_fold.append(np.sum(y_pred != y_test_current)/y_pred.shape)

    # take the average error from all folds for this k
    avg_errors.append(np.average(error_fold))

print(avg_errors)

pass

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

```
[0.6921999999999999, 0.726, 0.8367999999999999]
Computation time: 376.25
```

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

(2) 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 [ ]:

```
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=X_test_current, y=y_test_current) # CHANGE
dists = knn.compute_distances(X=X_test_current, norm=L1_norm) # L1-norm
y_pred = knn.predict_labels(dists, k=10) # k = 10

# set the error
error = np.sum(y_pred != y_test_current)/y_pred.shape

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

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

```
Error rate achieved: [0.624]
```

Question:

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

Answer:

My error is 0.102 less from the original error. I used k = 10 and the L1-norm.