Euclidean Distance Function Without Loops (Score: 6.0 / 6.0)

1. Test cell (Score: 1.0 / 1.0)
2. Test cell (Score: 1.0 / 1.0)
3. Test cell (Score: 1.0 / 1.0)
4. Test cell (Score: 1.0 / 1.0)
5. Test cell (Score: 1.0 / 1.0)
6. Test cell (Score: 1.0 / 1.0)

About this Exercise

In the preceding activity, you derived a Euclidean distance matrix. Now that you have calculated the distance between points in terms of matrix operations, you are ready to write an efficient program that leverages NumPy's optimized functions. In this code exercise, rather than using loops, you will write a function to compute Euclidean distances between sets of vectors using NumPy functions.

Evaluation

You must complete this exercise in order to unlock the final project in this module. Your score on this assignment will not be included in the final grade calculation.

You are expected to write code where you see *# YOUR CODE HERE* within the cells of this notebook. Not all cells will be graded; code input cells followed by cells marked with *#Autograder test cell* will be graded. Upon submitting your work, the code you write at these designated positions will be assessed using an "autograder" that will run all test cells to assess your code. You will receive feedback from the autograder that will identify any errors in your code. Use this feedback to improve your code if you need to resubmit. Be sure not to change the names of any provided functions, classes, or variables within the existing code cells, as this will interfere with the autograder. Also, remember to execute all code cells sequentially, not just those you’ve edited, to ensure your code runs properly.

You can resubmit your work as many times as necessary before the submission deadline. If you experience difficulty or have questions about this exercise, use the Q&A discussion board to engage with your peers or seek assistance from the instructor.

Before starting your work, please review eCornell's policy regarding plagiarism (the presentation of someone else's work as your own without source credit).

Submit Code for Autograder Feedback

Once you have completed your work on this notebook, you will submit your code for autograder review. Follow these steps:

1. **Save your notebook.**
2. **Mark as Completed —** In the blue menu bar along the top of this code exercise window, you’ll see a menu item called **Education**. In the **Education** menu, click **Mark as Completed** to submit your code for autograder/instructor review. This process will take a moment and a progress bar will show you the status of your submission.
3. **Review your results —** Once your work is marked as complete, the results of the autograder will automatically be presented in a new tab within the code exercise window. You can click on the assessment name in this feedback window to see more details regarding specific feedback/errors in your code submission.
4. **Repeat, if necessary —** The Jupyter notebook will always remain accessible in the first tabbed window of the exercise. To reattempt the work, you will first need to click **Mark as Uncompleted** in the **Education** menu and then proceed to make edits to the notebook. Once you are ready to resubmit, follow steps one through three. You can repeat this procedure as many times as necessary.

Import NumPy and Check Python Version

First, you must import NumPy. Let's also check our version of Python. We've added the code for you for this first step.

In [1]:

```
import sys
import numpy as np # Numpy is Python's built in library for matrix operations.
from pylab import *
sys.path.append('/home/codio/workspace/.guides/hf')
from helper import *
print('You\'re running python %s' % sys.version.split(' ')[0])
```

```
You're running python 3.6.8
```

Euclidean Distances in Python

Many machine learning algorithms access their input data primarily through pairwise (Euclidean) distances, therefore it is important that we have a fast function that computes pairwise distances of input vectors.

Assume we have n row data vectors $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^d$ and m row vectors $\mathbf{z}_1, \dots, \mathbf{z}_m \in \mathbb{R}^d$. With these data vectors, let us define two matrices $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_n]^\top \in \mathbb{R}^{n\times d}$, where the i-th row is vector $\mathbf{x}_i$, and $\mathbf{Z} = [\mathbf{z}_1, \dots, \mathbf{z}_m]^\top \in \mathbb{R}^{m\times d}$. We want a distance function that takes as input these two matrices $\mathbf{X}$ and $\mathbf{Z}$ and outputs a matrix $\mathbf{D} \in \mathbb{R}^{n\times m}$, where the entries of $\mathbf{D}$ are given by

$$\mathbf{D}_{ij} = \sqrt{(\mathbf{x}_i - \mathbf{z}_j)^\top (\mathbf{x}_i - \mathbf{z}_j)}$$

A naïve implementation to compute pairwise distances may look like the code below:

In [2]:

```
def l2distanceSlow(X,Z=None):
    if Z is None:
        Z = X

    n, d = X.shape     # dimension of X
    m= Z.shape[0]   # dimension of Z
    D=np.zeros((n,m)) # allocate memory for the output matrix
    for i in range(n):     # loop over vectors in X
        for j in range(m): # loop over vectors in Z
            D[i,j]=0.0;
            for k in range(d): # loop over dimensions
                D[i,j]=D[i,j]+(X[i,k]-Z[j,k])**2; # compute l2-distance between the ith and jth vector
            D[i,j]=np.sqrt(D[i,j]); # take square root
    return D
```

Please read through the code above carefully and make sure you understand it. It is perfectly correct and will produce the correct result... eventually. To see what is wrong, try running the `l2distanceSlow` code below on an extremely small matrix `X`.

In [3]:

```
X=np.random.rand(700,100)
print("Running the naive version, please wait...")
%time Dslow=l2distanceSlow(X)
```

```
Running the naive version, please wait...

CPU times: user 1min 17s, sys: 315 ms, total: 1min 17s
Wall time: 1min 19s
```

This code defines some random data in $\mathbf{X}$ and computes the corresponding distance matrix $\mathbf{D}$. The `%time` statement determines how long this code takes to run. This implementation is much too slow for such a simple operation on a small amount of data, and writing code like this to deal with matrices in this course will result in code that takes **days** to run.

As a general rule, you should avoid tight loops at all cost. As you will see in the remainder of this exercise, you can do much better by performing bulk matrix operations using the NumPy package, which calls highly optimized compiled code behind the scenes.

Efficient Programming with NumPy

Although there is an execution overhead per line in Python, matrix operations are optimized and fast. In order to successfully program in this course, you need to free yourself from "for-loop" thinking and start thinking in terms of matrix operations. Python for scientific computing can be very fast if almost all the time is spent on a few heavy duty matrix operations. In this exercise, you will transform the function above into a few matrix operations *without any loops at all.*

The key to efficient programming in Python for machine learning in general is to think about it in terms of mathematics and not in terms of loops.

Exercises

In the following three exercises, you'll take the steps necessary to implement the euclidean distance function without loops.

Exercise 1: Inner-Product Matrix

Show that the Inner-Product Matrix (Gram matrix) can be expressed in terms of pure matrix multiplication involving the matrices $\mathbf{X}$ and $\mathbf{Z}$.

Recall that the entries of the Gram matrix $\mathbf{G}$ are of the form:

$$\mathbf{G}_{ij} = \mathbf{x}_i^\top \mathbf{z}_j$$

Once you are done with the derivation, implement the function **`innerproduct`** below.

In [4]: Student's answer (Top)

```
def innerproduct(X,Z=None):
    '''
    function innerproduct(X,Z)

    Computes the inner-product matrix.
    Syntax:
    D=innerproduct(X,Z)
    Input:
    X: nxd data matrix with n vectors (rows) of dimensionality d
    Z: mxd data matrix with m vectors (rows) of dimensionality d

    Output:
    Matrix G of size nxm
    G[i,j] is the inner-product between vectors X[i,:] and Z[j,:]

    call with only one input:
    innerproduct(X)=innerproduct(X,X)
    '''
    if Z is None: # case when there is only one input (X)
        Z=X;

    # YOUR CODE HERE
    return X@Z.T
    #raise NotImplementedError()
```

In [5]:

```
#Run this self-test cell to check your code

def innerprod_0():
    # test the output dimensions of innerproduct with one input matrix
    X = np.random.rand(700,10) # define 700 random inputs X
    test = (innerproduct(X).shape==700,700)    # check if inner-product matrix has dimension 700x700
    return test

def innerprod_1():
    # test the output dimensions of innerproduct with two matrices
    X = np.random.rand(700,10) # define 700 random inputs X
    Z = np.random.rand(200,10) # define 200 random inputs Z
    test=(innerproduct(X,Z).shape ==(700,200)) # check if inner-product matrix has dimensions 700x200
    return test

def innerprod_2():
    X = np.random.rand(700,100) # define 700 random inputs X
    IP1 = innerproduct(X) # compute inner-product matrix with YOUR code
    IP2 = innerproduct_grader(X) # compute inner-product matrix with OUR code
    test = np.linalg.norm(IP1 - IP2) # compute the norm of the difference
    return test<1e-5 # this norm should be essentially 0

def innerprod_3():
    X = np.random.rand(700,100) # define 700 random inputs X
    Z = np.random.rand(300,100) # define 300 random inputs X
    IP1 = innerproduct(X,Z) # compute inner-product matrix with YOUR code
    IP2 = innerproduct_grader(X,Z) # compute inner-product matrix with OUR code
    test = np.linalg.norm(IP1 - IP2) # compute the norm of the difference
    return test<1e-5 # this norm should be essentially 0


runtest(innerprod_0,'innerprod_0 Dimensions with 1 Matrix')
runtest(innerprod_1,'innerprod_1 Dimensions with 2 Matrices')
runtest(innerprod_2,'innerprod_2 Correctness with 1 Matrix')
runtest(innerprod_3,'innerprod_3 Correctness with 2 Matrices')
```

```
Running Test: innerprod_0 Dimensions with 1 Matrix ... ✔ Passed!
Running Test: innerprod_1 Dimensions with 2 Matrices ... ✔ Passed!
Running Test: innerprod_2 Correctness with 1 Matrix ... ✔ Passed!
Running Test: innerprod_3 Correctness with 2 Matrices ...

✔ Passed!
```

In [6]: Grade cell: `cell-innerprod1_test` Score: 1.0 / 1.0 (Top)

```
# Autograder test cell - worth 1 point
# runs innerprod_1
### BEGIN HIDDEN TESTS
X = np.random.rand(700,100)
IP1 = innerproduct(X)
IP2 = innerproduct_grader(X)

test = np.linalg.norm(IP1 - IP2)
assert test<1e-5
### END HIDDEN TESTS
```

In [7]: Grade cell: `cell-innerprod2_test` Score: 1.0 / 1.0 (Top)

```
# Autograder test cell - worth 1 Point
# runs innerprod_2
### BEGIN HIDDEN TESTS
X = np.random.rand(700,100)
Z = np.random.rand(300,100)
IP1 = innerproduct(X,Z)
IP2 = innerproduct_grader(X,Z)

test = np.linalg.norm(IP1 - IP2)
assert test<1e-5
### END HIDDEN TESTS
```

Exercise 2: Implement `calculate_S` and `calculate_R`

Recall that the *element-wise squared* Euclidean distance matrix $\mathbf{D} \odot \mathbf{D} \in \mathbb{R}^{n\times m}$ is defined by

$$[\mathbf{D} \odot \mathbf{D}]_{ij} = (\mathbf{x}_i - \mathbf{z}_j)^\top (\mathbf{x}_i - \mathbf{z}_j)$$

Also, the matrices $\mathbf{S}, \mathbf{R} \in \mathbb{R}^{n\times m}$ are defined by

$$\mathbf{S}_{ij} = \mathbf{x}_i^\top \mathbf{x}_i \quad \text{and} \quad \mathbf{R}_{ij} = \mathbf{z}_j^\top \mathbf{z}_j$$

In the previous activity, we showed that

$$\mathbf{D} \odot \mathbf{D} = \mathbf{S} + \mathbf{R} - 2\mathbf{G}$$

Later in this exercise, you will implement `l2distance` to calculate $\mathbf{D}$. But you will need $\mathbf{S}$ and $\mathbf{R}$, which you will implement now in **`calculate_S`** and **`calculate_R`**, respectively. Ensure that your functions return $\mathbf{S}$ and $\mathbf{R}$ of size $n \times m$, as they will be added to $-2\mathbf{G}$ to get $\mathbf{D} \odot \mathbf{D}$.

Think about what the $\mathbf{S}$ and $\mathbf{R}$ matrices look like. You will find that the values in each row of $\mathbf{S}$ and the values in each column of $\mathbf{R}$ do not change! This is also apparent when considering that $\mathbf{S}_{ij} = \mathbf{x}_i^\top \mathbf{x}_i$ *for all* j; similar argument for $\mathbf{R}_{ij} = \mathbf{z}_j^\top \mathbf{z}_j$ *for all* i. That is,

$$\mathbf{S} = \begin{bmatrix} \mathbf{x}_1^\top \mathbf{x}_1 & \mathbf{x}_1^\top \mathbf{x}_1 & \cdots & \mathbf{x}_1^\top \mathbf{x}_1 \\ \mathbf{x}_2^\top \mathbf{x}_2 & \mathbf{x}_2^\top \mathbf{x}_2 & \cdots & \mathbf{x}_2^\top \mathbf{x}_2 \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_n^\top \mathbf{x}_n & \mathbf{x}_n^\top \mathbf{x}_n & \cdots & \mathbf{x}_n^\top \mathbf{x}_n \end{bmatrix} \quad \text{and} \quad \mathbf{R} = \begin{bmatrix} \mathbf{z}_1^\top \mathbf{z}_1 & \mathbf{z}_2^\top \mathbf{z}_2 & \cdots & \mathbf{z}_m^\top \mathbf{z}_m \\ \mathbf{z}_1^\top \mathbf{z}_1 & \mathbf{z}_2^\top \mathbf{z}_2 & \cdots & \mathbf{z}_m^\top \mathbf{z}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{z}_1^\top \mathbf{z}_1 & \mathbf{z}_2^\top \mathbf{z}_2 & \cdots & \mathbf{z}_m^\top \mathbf{z}_m \end{bmatrix}$$

Now you just need to figure out how to calculate $\mathbf{x}_i^\top \mathbf{x}_i$ and $\mathbf{z}_j^\top \mathbf{z}_j$ without loops. You might find the fact $\mathbf{a}^\top \mathbf{a} = \sum_{k=1}^{d} a_k^2$ and repeat function `np.repeat` (and its `axis` parameter) useful.

In [8]: Student's answer (Top)

```
def calculate_S(X, n, m):
    '''
    function calculate_S(X)

    Computes the S matrix.
    Syntax:
    S=calculate_S(X)
    Input:
    X: nxd data matrix with n vectors (rows) of dimensionality d
    n: number of rows in X
    m: output number of columns in S

    Output:
    Matrix S of size nxm
    S[i,j] is the inner-product between vectors X[i,:] and X[i,:]
    '''
    assert n == X.shape[0]

    # YOUR CODE HERE
    diags=np.diagonal(X@X.T)   # 1-dimensional vector
    diags=np.array([diags]).T  # convert to 2 dimensional vector and transpose
    return np.repeat(diags,m,axis=1)
    #raise NotImplementedError()
```

In [9]: Student's answer (Top)

```python
def calculate_R(Z, n, m):
    '''
    function calculate_R(Z)

    Computes the R matrix.
    Syntax:
    R=calculate_R(Z)
    Input:
    Z: mxd data matrix with m vectors (rows) of dimensionality d
    n: output number of rows in Z
    m: number of rows in Z

    Output:
    Matrix R of size nxm
    R[i,j] is the inner-product between vectors Z[j,:] and Z[j,:]
    '''
    assert m == Z.shape[0]

    # YOUR CODE HERE
    diags=np.diagonal(Z@Z.T)   # 1-dimensional vector
    diags=np.array([diags])  # convert to 2 dimensional vector and transpose
    return np.repeat(diags,n,axis=0)
    # raise NotImplementedError()
```

In [10]:

```python
#Run this self-test cell to check your code

def calculate_S_dimensions():
    X = np.random.rand(700,100) # define random inputs
    Z = np.random.rand(800,100) # define random inputs
    n,d1=X.shape
    m,d2=Z.shape
    S1 = calculate_S(X, n, m) # compute distances from your solutions
    o1,o2=S1.shape
    return (o1==n) and (o2==m)

def calculate_S_accuracy():
    X = np.random.rand(700,100) # define random inputs
    S1 = calculate_S(X, X.shape[0], 800) # compute distances from your solutions
    S2 = calculate_S_grader(X, X.shape[0], 800) #compute distance from ground truth
    test = np.linalg.norm(S1 - S2) # compare the two
    return test<1e-5 # difference should be small

def calculate_R_dimensions():
    X = np.random.rand(700,100) # define random inputs
    Z = np.random.rand(800,100) # define random inputs
    n,d1=X.shape
    m,d2=Z.shape
    R1 = calculate_R(Z, n, m) # compute distances from your solutions
    o1,o2=R1.shape
    return (o1==n) and (o2==m)

def calculate_R_accuracy():
    Z = np.random.rand(800,100) # define random inputs
    R1 = calculate_R(Z, 700, Z.shape[0]) # compute distances from your solutions
    R2 = calculate_R_grader(Z, 700, Z.shape[0]) #compute distance from ground truth
    test = np.linalg.norm(R1 - R2) # compare the two
    return test<1e-5 # difference should be small

runtest(calculate_S_dimensions,'calculate_S_dimensions')
runtest(calculate_S_accuracy,'calculate_S_accuracy')
runtest(calculate_R_dimensions,'calculate_R_dimensions')
runtest(calculate_R_accuracy,'calculate_R_accuracy')
```

```
Running Test: calculate_S_dimensions ... ✔ Passed!
Running Test: calculate_S_accuracy ... ✔ Passed!
Running Test: calculate_R_dimensions ... ✔ Passed!
Running Test: calculate_R_accuracy ... ✔ Passed!
```

Exercise 3: Implement `l2distance`

In this exercise, you will use the above formula to implement the function `l2distance`, which computes the Euclidean distance matrix $\mathbf{D}$ without a single loop.

Recall that the element-wise square of $\mathbf{D}$ is of the form

$$\mathbf{D} \odot \mathbf{D} = \mathbf{S} + \mathbf{R} - 2\mathbf{G}$$

and the entries of $\mathbf{D}$ are

$$\mathbf{D}_{ij} = \sqrt{[\mathbf{D} \odot \mathbf{D}]_{ij}}$$

Hint: Make sure that all entries of $\mathbf{D}$ are non-negative after you take the square root $\sqrt{[\mathbf{D} \odot \mathbf{D}]_{ij}}$ because sometimes very small positive numbers can become negative due to numerical imprecision. Since all distances must be non-negative, you can simply overwrite all negative values with `0.0` to avoid unintended consequences.

In [11]: Student's answer (Top)

```python
def l2distance(X,Z=None):
    '''
    function D=l2distance(X,Z)

    Computes the Euclidean distance matrix.
    Syntax:
    D=l2distance(X,Z)
    Input:
    X: nxd data matrix with n vectors (rows) of dimensionality d
    Z: mxd data matrix with m vectors (rows) of dimensionality d

    Output:
    Matrix D of size nxm
    D(i,j) is the Euclidean distance of X(i,:) and Z(j,:)

    call with only one input:
    l2distance(X)=l2distance(X,X)
    '''
    if Z is None:
        Z=X;

    n,d1=X.shape
    m,d2=Z.shape
    assert (d1==d2), "Dimensions of input vectors must match!"


    # YOUR CODE HERE
    S=calculate_S(X,n,m)
    R=calculate_R(Z,n,m)
    G=X@Z.T
    return np.sqrt(S+R-2*G)
    # raise NotImplementedError()
```

In [12]:

```python
#Run this self-test cell to check your code

def distance_accuracy():
    X = np.random.rand(700,100) # define random inputs
    D1 = l2distance(X) # compute distances from your solutions
    D2 = l2distance_grader(X) #compute distance from ground truth
    test = np.linalg.norm(D1 - D2) # compare the two
    return test<1e-5 # difference should be small

def distance_squareroot():
    X = np.random.rand(700,100) # define random inputs
    D1 = l2distance(X) # compute distances from your solutions
    D2sq = l2distance_grader(X)**2 #compute distance from ground truth *but square them*
    test = np.linalg.norm(D1 - D2sq) # compare the two
    return test>1e-5 # difference should be big

def dimensions():
    X = np.random.rand(700,100) # define random inputs
    Z = np.random.rand(800,100) # define random inputs
    n,d1=X.shape
    m,d2=Z.shape
    D1 = l2distance(X,Z) # compute distances from your solutions
    o1,o2=D1.shape
    return (o1==n) and (o2==m)

def matrix_dist_accuracy():
    X = np.random.rand(700,100)
    Z = np.random.rand(300,100)
    D1Z = l2distance(X,Z)
    D2Z = l2distance_grader(X,Z)
    test = np.linalg.norm(D1Z - D2Z)
    return test<1e-5

runtest(distance_accuracy,'distance_accuracy')
runtest(distance_squareroot,'distance_squareroot')
runtest(dimensions,'dimensions')
runtest(matrix_dist_accuracy,'matrix_dist_accuracy')
```

```
Running Test: distance_accuracy ... ✔ Passed!
Running Test: distance_squareroot ... ✔ Passed!
Running Test: dimensions ... ✔ Passed!
Running Test: matrix_dist_accuracy ...

✔ Passed!
```

In [13]: Grade cell: `cell-distance_accuracy_test` Score: 1.0 / 1.0 (Top)

```python
# Autograder test cell - worth 1 Point
# runs distance_accuracy
### BEGIN HIDDEN TESTS
X = np.random.rand(700,100)
D1 = l2distance(X)
D2 = l2distance_grader(X)

test = np.linalg.norm(D1 - D2)
assert test<1e-5
### END HIDDEN TESTS
```

In [14]: Grade cell: `cell-distance_squareroot_test` Score: 1.0 / 1.0 (Top)

```python
# Autograder test cell - worth 1 Point
# runs distance_squareroot
### BEGIN HIDDEN TESTS
X = np.random.rand(700,100)
D1 = l2distance(X)
D2sq = l2distance_grader(X)**2

test = np.linalg.norm(D1 - D2sq)
assert test>1e-5
### END HIDDEN TESTS
```

In [15]: Grade cell: `cell-dimensions_test` Score: 1.0 / 1.0 (Top)

```python
# Autograder test cell - worth 1 Point
# runs dimensions
### BEGIN HIDDEN TESTS
X = np.random.rand(700,100) # define random inputs
Z = np.random.rand(800,100) # define random inputs
n,d1=X.shape
m,d2=Z.shape
D1 = l2distance(X,Z) # compute distances from your solutions
o1,o2=D1.shape

assert (o1==n) and (o2==m)
### END HIDDEN TESTS
```

In [16]: Grade cell: `cell-matrix_dist_accuracy_test` Score: 1.0 / 1.0 (Top)

```python
# Autograder test cell - worth 1 Point
# runs matrix_dist_accuracy
### BEGIN HIDDEN TESTS
X = np.random.rand(700,100)
Z = np.random.rand(300,100)

D1Z = l2distance(X,Z)
D2Z = l2distance_grader(X,Z)

test = np.linalg.norm(D1Z - D2Z)
assert test<1e-5
### END HIDDEN TESTS
```

Let's now compare the speed of your `l2distance` function against the previous naïve implementation:

In [17]:

```python
import time
current_time = lambda: int(round(time.time() * 1000))

X=np.random.rand(700,100)
Z=np.random.rand(300,100)

print("Running the naïve version...")
before = current_time()
Dslow=l2distanceSlow(X)
after = current_time()
t_slow = after - before
print("{:2.0f} ms".format(t_slow))

print("Running the vectorized version...")
before = current_time()
Dfast=l2distance(X)
after = current_time()
t_fast = after - before
print("{:2.0f} ms".format(t_fast))


speedup = t_slow / t_fast
print("The two methods should deviate by very little: {:05.6f}".format(norm(Dfast-Dslow)))
print("but your NumPy code was {:05.2f} times faster!".format(speedup))
```

```
Running the naïve version...

75358 ms
Running the vectorized version...
37 ms
The two methods should deviate by very little: 0.000000
but your NumPy code was 2036.70 times faster!
```

How much faster is your code now? With this implementation, you should easily be able to compute the distances between **many more** vectors. It should be clear now, even for small datasets, that the for-loop based implementation could take several days or even weeks to perform basic operations that take seconds or minutes with well-written NumPy code.