201700949 설재혁

벡터 덧셈

In [59]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all" # 마지막 값만 출력하는 것이 아닌 모든 결과
값 출력

import re, math, random # regexes, math functions, random numbers
import matplotlib.pyplot as plt # pyplot
from collections import defaultdict, Counter
from functools import partial, reduce # For python3, "reduce" function is added

import numpy as np

def vector_add(v, w):
    """adds two vectors componentwise"""
    return [v_i + w_i for v_i, w_i in zip(v,w)]

v = [x for x in range(1, 11, 2)]
w = [y for y in range(11, 21, 2)]

vector_add(v, w)
```

Out[59]:

```
[12, 16, 20, 24, 28]
```

In [60]:

```
# Numpy version
np.array(v) + np.array(w)
```

Out[60]:

```
array([12, 16, 20, 24, 28])
```

In [61]:

```
%timeit vector_add(v,w)
%timeit np.array(v) + np.array(w)
```

```
561 ns ± 34.1 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops
each)
2.19 µs ± 47.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops
each)
```

벡터 뺄셈

In [62]:

```
def vector_subtract(v,w):
    """adds two vectors componentwise"""
    return [v_i - w_i for v_i, w_i in zip(v,w)]

vector_subtract(v,w)
```

Out[62]:

```
[-10, -10, -10, -10, -10]
```

In [63]:

```
np.array(v) - np.array(w)
```

Out[63]:

```
array([-10, -10, -10, -10, -10])
```

벡터 리스트 덧셈

In [64]:

```
v = [x for x in range(1, 11, 2)]
w = [y for y in range(11, 21, 2)]

# Version 1
def vector_sum(vectors):
    return reduce(vector_add,vectors)

vectors = [v,w,v,w,v,w]
vector_sum(vectors)

#Version 2
def vector_sum_modified(vectors):
    return [sum(value) for value in zip(*vectors)]

vectors = [v,w,v,w,v,w]
vector_sum_modified(vectors)
```

Out[64]:

```
[36, 48, 60, 72, 84]
```

Out[64]:

```
[36, 48, 60, 72, 84]
```

In [65]:

```
# Numpy Operation
np.sum([v,w,v,w,v,w], axis=0)
np.sum([v,w,v,w,v,w], axis=1)
# axis=0 는 row [v,w,v,w,v,w]를 하나의 matrix로 생각했을 때, column별로 sum operation을
 하라는 의미
# axis=1 는 row [v,w,v,w,v,w]를 하나의 matrix로 생각했을 때, row별로 sum operation을 하라
는 의미
```

Out[65]:

```
array([36, 48, 60, 72, 84])
```

Out[65]:

```
array([25, 75, 25, 75, 25, 75])
```

벡터 스칼라 곱

In [26]:

```
def scalar_multiply(c, v):
    return [c * v_i for v_i in v]

scalar = 3
scalar_multiply(scalar, v)
```

Out[26]:

```
[3, 9, 15, 21, 27]
```

In [27]:

```
# Numpy version: numpy는 배열의 크기가 다르더라도 기본적인 vector 연산이 가능하도록 지원. 이를 b
roadcasting이라고 함
scalar * np.array(v)
```

Out[27]:

```
array([ 3,  9, 15, 21, 27])
```

벡터 리스트 평균

In [36]:

```
def vector_mean(vectors):
    """compute the vector whose i-th element is the mean of the i-th elements of
the input vectors"""
    n = len(vectors)
    return scalar_multiply(1/n, vector_sum(vectors))

v = [1,2,3,4]
w = [-4,-3,-2,-1]

vector_mean([v,w,v,w,v])
```

Out[36]:

```
[-1.0, 0.0, 1.0, 2.0]
```

In [46]:

```
# Numpy version
np.mean([v,w,v,w,v,w], axis=0)
# axis=0 는 row [v,w,v,w,v,w]를 하나의 matrix로 생각했을 때, column별로 mean operation을
하라는 의미
# axis=1 는 row [v,w,v,w,v,w]를 하나의 matrix로 생각했을 때, row별로 mean operation을 하
라는 의미
```

Out[46]:

```
array([-1.5, -0.5,  0.5,  1.5])
```

In [45]:

```
[v,w,v,w,v,w]
```

Out[45]:

```
[[1, 2, 3, 4],
 [-4, -3, -2, -1],
 [1, 2, 3, 4],
 [-4, -3, -2, -1],
 [1, 2, 3, 4],
 [-4, -3, -2, -1]]
```

벡터의 내적

In [47]:

```
def dot(v,w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i for v_i, w_i in zip(v, w))

v = [1,2,3,4]
w = [-4,-3,-2,-1]

dot(v, w)
```

Out[47]:

```
-20
```

In [48]:

```
# Numpy version
np.dot(v,w)
```

Out[48]:

```
-20
```

벡터 성분 제곱 값의 합

In [66]:

```
def sum_of_squares(v):
    """v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)

v = [1,2,3,4]
sum_of_squares(v) # v * v = [1,4,9,16]

# Magnitude (or length)
def magnitude(v):
    return math.sqrt(sum_of_squares(v))

magnitude(v)
```

Out[66]:

```
30
```

Out[66]:

```
5.477225575051661
```

In [53]:

```
# Numpy version
np.linalg.norm(v)
```

Out[53]:

```
5.477225575051661
```

두 벡터 사이의 거리

In [67]:

```
#original version
def squared_distance(v, w):
    return sum_of_squares(vector_subtract(v, w))

def distance(v, w):
    return math.sqrt(squared_distance(v, w))

v = [1,2,3,4]
w = [-4,-3,-2,-1]

squared_distance(v,w)
distance(v,w)
```

Out[67]:

```
100
```

Out[67]:

```
10.0
```

In [68]:

```
# Numpy version
np.linalg.norm(np.subtract(v,w))
np.sqrt(np.sum(np.subtract(v,w)**2))
```

Out[68]:

```
10.0
```

Out[68]:

```
10.0
```

행렬 형태

In [71]:

```
def shape(A):
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0
    return num_rows, num_cols

def get_row(A, i):
    return A[i]

def get_column(A, j):
    return [A_i[j] for A_i in A]

example_matrix = [[1,2,3,4,5], [11,12,13,14,15], [21,22,23,24,25]]

shape(example_matrix)
get_row(example_matrix, 0)
get_column(example_matrix, 3)
```

Out[71]:

```
(3, 5)
```

Out[71]:

```
[1, 2, 3, 4, 5]
```

Out[71]:

```
[4, 14, 24]
```

In [75]:

```
# Numpy version
np.shape(example_matrix)
example_matrix = np.array(example_matrix)
example_matrix[1] # row slicing
example_matrix[:, 3] # row slicing
```

Out[75]:

```
(3, 5)
```

Out[75]:

```
array([11, 12, 13, 14, 15])
```

Out[75]:

```
array([ 4, 14, 24])
```

행렬 생성

In [76]:

```
def make_matrix(num_rows, num_cols, entry_fn):
    """returns a num_rows x num_cols matrix whose (i,j)-th entry is entry_fn(i, 
 j)"""
    return [[entry_fn(i, j) for j in range(num_cols)]
            for i in range(num_rows)]

def is_diagonal(i, j):
    """1's on the 'diagonal', 0's everywhere else"""
    return 1 if i == j else 0

identity_matrix = make_matrix(5, 5, is_diagonal)

identity_matrix
```

Out[76]:

```
[[1, 0, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0],
 [0, 0, 0, 0, 1]]
```

In [79]:

```
# Numpy version
np.identity(5)
```

Out[79]:

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

이진 관계

In [80]:

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (5, 6), (
5, 7), (6, 8), (7, 8), (8, 9)]

friendships = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0],  # user 0
               [1, 0, 1, 1, 0, 0, 0, 0, 0, 0],  # user 1
               [1, 1, 0, 1, 0, 0, 0, 0, 0, 0],  # user 2
               [0, 1, 1, 0, 1, 0, 0, 0, 0, 0],  # user 3
               [0, 0, 0, 1, 0, 1, 0, 0, 0, 0],  # user 4
               [0, 0, 0, 0, 1, 0, 1, 1, 0, 0],  # user 5
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],  # user 6
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0],  # user 7
               [0, 0, 0, 0, 0, 0, 1, 1, 0, 1],  # user 8
               [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]]  # user 9

friendships[0][2] == 1 # True, user 0과 2는 친구이다
friendships[0][8] == 1 # False, user 0과 8은 친구가 아니다

# only need # to look at # one row
friends_of_five = [i for i, is_friend in enumerate(friendships[5]) if is_friend]
print(friends_of_five)
```

Out[80]:

```
True
```

Out[80]:

```
False
```

```
[4, 6, 7]
```

행렬 덧셈

In [83]:

```
def matrix_add(A, B):
    if shape(A) != shape(B):
        raise ArithmeticError("cannot add matrices with different shapes")

    num_rows, num_cols = shape(A)
    def entry_fn(i, j): return A[i][j] + B[i][j]

    return make_matrix(num_rows, num_cols, entry_fn)

A = [[ 1., 0., 0.], [ 0., 1., 2.]]
B = [[ 5., 4., 3.], [ 2., 2., 2.]]

matrix_add(A,B)
```

Out[83]:

```
[[6.0, 4.0, 3.0], [2.0, 3.0, 4.0]]
```

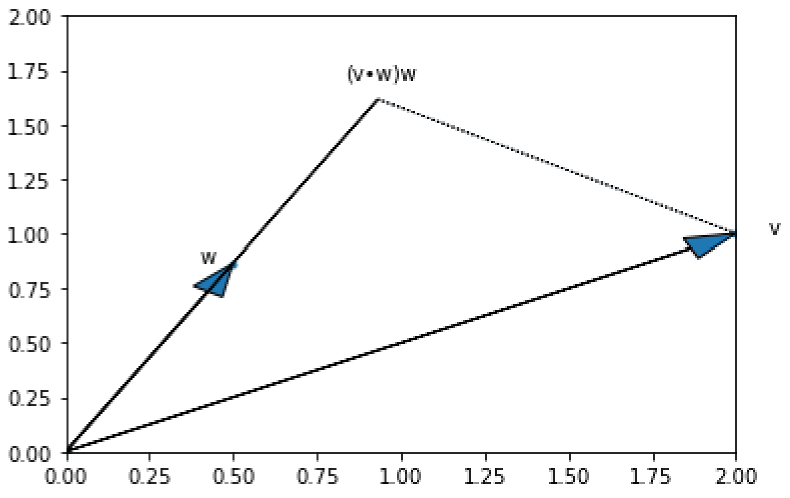
벡터 점곱 그래프

In [85]:

```
def make_graph_dot_product_as_vector_projection(plt):
    v = [2, 1]
    w = [math.sqrt(.25), math.sqrt(.75)]
    c = dot(v, w)
    vonw = scalar_multiply(c, w)
    o = [0,0]

    plt.arrow(0, 0, v[0], v[1],
              width=0.002, head_width=.1, length_includes_head=True)
    plt.annotate("v", v, xytext=[v[0] + 0.1, v[1]])
    plt.arrow(0 ,0, w[0], w[1],
              width=0.002, head_width=.1, length_includes_head=True)
    plt.annotate("w", w, xytext=[w[0] - 0.1, w[1]])
    plt.arrow(0, 0, vonw[0], vonw[1], length_includes_head=True)
    plt.annotate(u"(v•w)w", vonw, xytext=[vonw[0] - 0.1, vonw[1] + 0.1])
    plt.arrow(v[0], v[1], vonw[0] - v[0], vonw[1] - v[1],
              linestyle='dotted', length_includes_head=True)
    plt.scatter(*zip(v,w,o),marker='.')
    plt.axis([0,2,0,2]) # 잘리는 부분이 있어서 변경
    plt.show()

%matplotlib inline
make_graph_dot_product_as_vector_projection(plt)
```



Lab5 (1)

행렬의 점곱연산

In [117]:

```
def my_matrix_dot(A, B):
    rowsA = len(A)
    colsA = len(A[0])
    rowsB = len(B)
    colsB = len(B[0])
    if rowsA != colsB: # 앞의 행렬의 행과 뒤 행렬의 열의 개수가 같지 않으면 에러 출력
        raise ArithmeticError(
            'Number of A columns must equal number of B rows.')

    C = make_matrix(rowsA, colsB, is_diagonal)
    for i in range(rowsA):
        for j in range(colsB):
            total = 0
            for ii in range(colsA):
                total += A[i][ii] * B[ii][j]
            C[i][j] = total

    return C

A = [[1,2,3], [4,5,6]]
B = [[1,2],[3,4],[5,6]]

my_matrix_dot(A,B)

# Numpy version
np.dot(A,B) # vector와 마찬가지로 크기 같은 matrix 형태의 list가 들어오면 자동으로 변환함
```

Out[117]:

```
[[22, 28], [49, 64]]
```

Out[117]:

```
array([[22, 28],
       [49, 64]])
```

Lab5 (2)

전치행렬

In [132]:

```
def my_matrix_tranpose1(M): # 방법1
    return list(zip(*M))

def my_matrix_tranpose2(M): # 방법2
    return [[row[i] for row in M] for i in range(len(M[0]))]

my_matrix_tranpose1(A)
my_matrix_tranpose1(B)
my_matrix_tranpose2(A)
my_matrix_tranpose2(B)

# Numpy version
np.transpose(A) # vector와 마찬가지로 크기 같은 matrix 형태의 list가 돌아오면 자동으로 변환
np.transpose(B) # vector와 마찬가지로 크기 같은 matrix 형태의 list가 돌아오면 자동으로 변환
```

Out[132]:

```
[(1, 4), (2, 5), (3, 6)]
```

Out[132]:

```
[(1, 3, 5), (2, 4, 6)]
```

Out[132]:

```
[[1, 4], [2, 5], [3, 6]]
```

Out[132]:

```
[[1, 3, 5], [2, 4, 6]]
```

Out[132]:

```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Out[132]:

```
array([[1, 3, 5],
       [2, 4, 6]])
```

201700949 설재혁

In [ ]: