SEAS 8510 - Analytical Methods for ML

Homework 1

Due Date: March 30, 2024 (9:00am EST)

For each of the following exercises, solve using Python in the Google Colab environment. Your homework 1 should be submitted as a pdf containing your responses, your code, and your printed results where applicable.

All of the imports

In [ ]:
```
import pandas as pd
import numpy as np
import datetime as dt
import matplotlib.pyplot as plt
%matplotlib inline
```

Exercise 1.

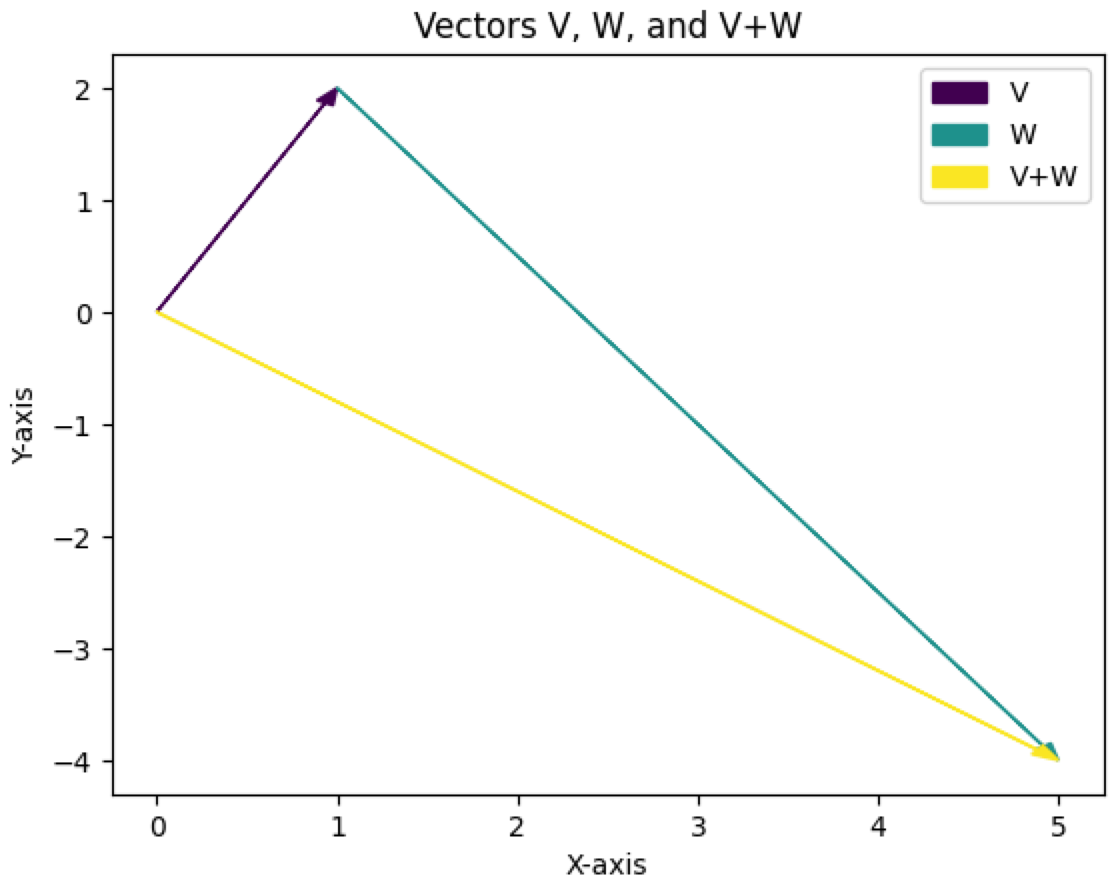
Write your own code to produce and graph the vectors in the following figure:

In [ ]:
```
######################################################################################
#
# plot_vectors - Plots vectors on a single chart.
#
#  Args:
#      vectors: A list of lists or numpy arrays, where each sublist represents
#               a vector (x, y).
#      labels: A list of strings, optional, to label each vector.
#      operation: An operation to use between the vectors
#  Returns:
#      The calculated vector
#
#  Operation
#      Uses matplotlib to plot the vectors. After initial setup, and validating
#      parameters, the provided vectors are processed. This is done by iterating
```

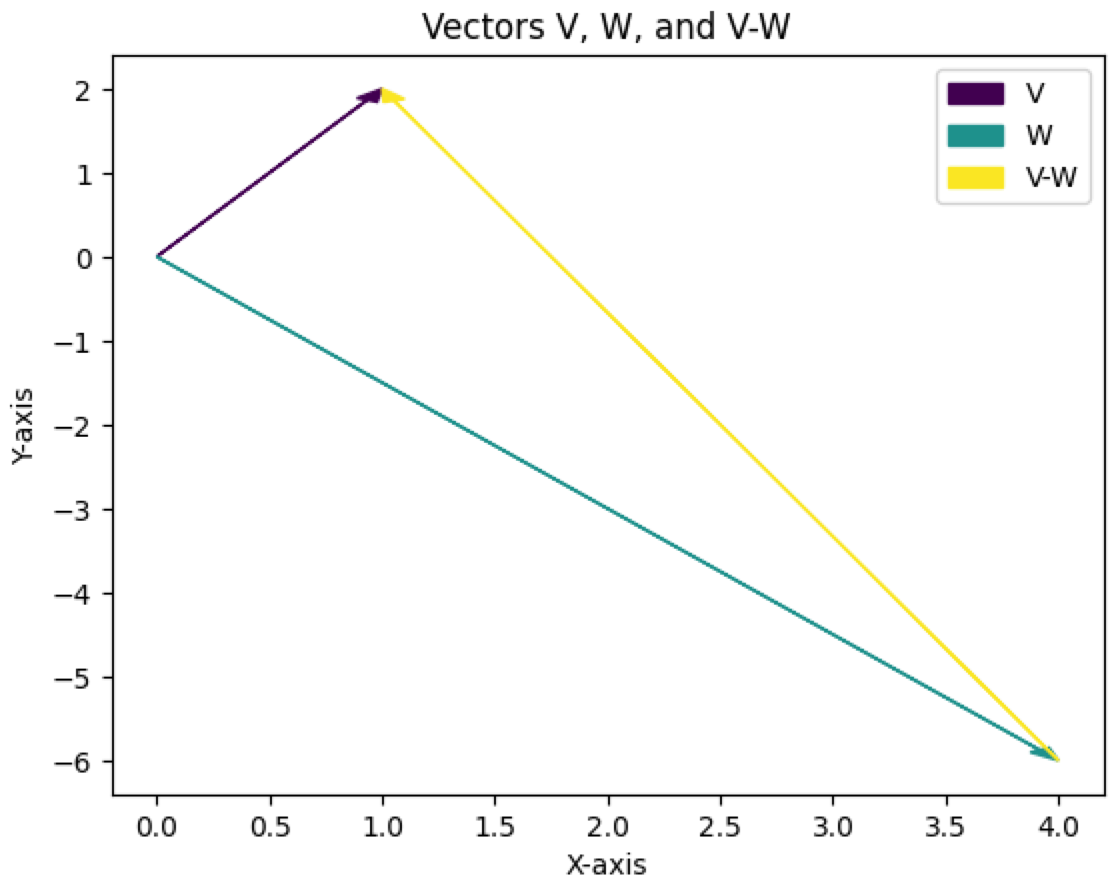
```
#      over them and plotting each one while maintaining the information that is
#      needed to plot the resultant vector.
####################################################################################
def plot_vectors(vectors, labels=None, operation="+"):
  plt.figure()

  vectors = np.array(vectors)  # Ensure vectors is a numpy array

  # Set up the color map
  # Get the number of vectors plus one for the result
  num_vectors = vectors.shape[0] + 1
  # currently only supports two vectors and one result
  if num_vectors > 3:
    raise ValueError(f"Currently only supports two vectors, you supplied {num_vectors - 1}.")

  # Create a colormap object
  colormap="viridis"
  cmap = plt.colormaps[colormap]
  # Normalize color values based on number of vectors
  norm = plt.Normalize(vmin=0, vmax=num_vectors - 1)

  # Check if labels are provided
  if labels is None:
    labels = [f"Vector {i+1}" for i in range(len(vectors))]

  # Check operation (+ or -)
  if operation not in ["+", "-"]:
    raise ValueError("Invalid operation. Use '+' for addition or '-' for subtraction.")

  startX = 0
  startY = 0
  title = "Vectors "
  # Plot each vector with its label
  for i, vector in enumerate(vectors):
    title = title + labels[i] + ", "
    color = cmap(norm(i))  # Get color based on index and normalization
    if i > 0:
      resultLabel = resultLabel + operation + labels[i]
      if operation == "-":
        rvec = rvec - vector
        spX = vector[0]
        spY = vector[1]
```

```
        epX = rvec[0]
        epY = rvec[1]
      else:
        rvec = rvec + vector
        spX = 0
        spY = 0
        epX = rvec[0]
        epY = rvec[1]
    else:
      resultLabel = labels[i]
      rvec = vector
    plt.arrow(startX, startY, vector[0], vector[1], length_includes_head=True, head_width=0.1, label=labels[i], color
    if operation == "-":
      startX = 0
      startY = 0
    else:
      startX = startX + vector[0]
      startY = startY + vector[1]
  title = title + "and " + resultLabel
  # Plot the resultant vector
  color = cmap(norm(num_vectors))  # Get color for the result
  plt.arrow(spX, spY, epX, epY, length_includes_head=True, head_width=0.1, label=resultLabel, color=color)

  # Set labels and title
  plt.xlabel("X-axis")
  plt.ylabel("Y-axis")
  plt.title(title)

  # Add legend if labels are provided
  if labels is not None:
    plt.legend()

  # Show the plot
  plt.show()
  return rvec

# Test with two vectors
vectors = np.array([[1, 2], [4, -6]])
labels = ["V", "W"]
operation = "+"
rvec = plot_vectors(vectors, labels, operation)
print(f"For {vectors[0]} + {vectors[1]} the result is {rvec}.")
```

```
operation = "-"
rvec = plot_vectors(vectors, labels, operation)
print(f"For {vectors[0]} - {vectors[1]} the result is {rvec}.")
```




```
For [1 2] + [ 4 -6] the result is [ 5 -4].
```




```
For [1 2] - [ 4 -6] the result is [-3  8].
```

Exercise 2.

Write an algorithm that computes the norm of a vector by translating the following equation into code.

$$\|V\| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

Confirm, using random vectors with different dimensionalities and orientations, that you get the same result as np.linalg.norm(). This exercise is designed to give you more experience with indexing NumPy arrays and translating formulas into code; in practice, it's often easier to use np.linalg.norm().

In [ ]:
```
####################################################################################
#
# manual_norm - Calculates the Euclidean norm (L2 norm) of a vector.
#
# Args:
#      vector: A numpy array, list or tuple representing the vector (x, y, ..., z).
#
#  Returns:
#      The Euclidean norm of the vector.
#
#  Operation
#     Run through each element in the vector. Accumulate the square of the element
#     values. When done return the square root of the accumulated squares.
####################################################################################
def manual_norm(vector):
  vector = np.array(vector)  # Ensure vector is a numpy array
  sum_of_squares = 0
  for element in vector:
    sum_of_squares += element**2
  return sum_of_squares**0.5

# create 10 random vectors with dimension from 2 to 10 and values from 0 to 10. Show
# the vector, the norm from the manual calculation, and the built in function.
# Calculate the difference to show that it is zero.
for i in range(1, 11):
    dimension = np.random.randint(2, 10)
    # Generate random numbers between 0 and 10
    vector = np.random.default_rng().uniform(0,10,dimension)
    print("")
    print(f"The vector has dimension {len(vector)} and the values are {np.array2string(vector, precision=2, floatmode
    n1 = manual_norm(vector)
    n2 = np.linalg.norm(vector)
    print(f"The manual norm is {n1:,.2f} and the function norm is {n2:,.2f}. The difference is {n1-n2:,.2f}.")
```

```
The vector has dimension 4 and the values are [7.71 2.25 9.47 0.64]
The manual norm is 12.44 and the function norm is 12.44. The difference is 0.00.

The vector has dimension 8 and the values are [9.64 4.96 6.12 8.39 8.18 0.85 1.74 5.30]
The manual norm is 18.01 and the function norm is 18.01. The difference is 0.00.

The vector has dimension 7 and the values are [4.39 5.19 8.91 3.57 2.24 9.25 5.95]
The manual norm is 16.26 and the function norm is 16.26. The difference is 0.00.

The vector has dimension 7 and the values are [4.95 7.34 3.77 4.36 5.51 6.72 1.73]
The manual norm is 13.79 and the function norm is 13.79. The difference is -0.00.

The vector has dimension 3 and the values are [4.24 6.31 0.66]
The manual norm is 7.63 and the function norm is 7.63. The difference is 0.00.

The vector has dimension 4 and the values are [2.29 6.63 8.23 7.29]
The manual norm is 13.04 and the function norm is 13.04. The difference is 0.00.

The vector has dimension 5 and the values are [5.44 3.06 0.31 4.83 1.38]
The manual norm is 8.02 and the function norm is 8.02. The difference is 0.00.

The vector has dimension 5 and the values are [0.02 7.43 6.65 0.45 7.83]
The manual norm is 12.69 and the function norm is 12.69. The difference is 0.00.

The vector has dimension 5 and the values are [3.91 4.63 5.71 8.43 6.83]
The manual norm is 13.68 and the function norm is 13.68. The difference is 0.00.

The vector has dimension 4 and the values are [3.77 4.24 5.92 8.59]
The manual norm is 11.88 and the function norm is 11.88. The difference is 0.00.
```

Exercise 3.

Create a Python function that will take a vector as input and output a unit vector in the same direction. What happens when you input the zeros vector?

In [ ]:

```
####################################################################################
#
# cuv -  Calculates the unit vector in the same direction as the given vector.
#
```

```
# Args:
#       vector: A numpy array, list or tuple representing the vector (x, y, ..., z).
#
#  Returns:
#       The unit vector in the same direction as the input vector, or None if the
#       input vector has a magnitude of 0.
####################################################################################
def cuv(vector):

  vector = np.array(vector)  # Ensure vector is a numpy array

  norm = np.linalg.norm(vector)  # Calculate the vector's magnitude

  # Handle zero vector case, if we did not do this, we would have a divide by zero
  if norm == 0:
    return None

  return vector / norm  # Normalize the vector to obtain the unit vector

# For the 3, 4 vector:
vector = [3, 4]
unit_vector = cuv(vector)
print(f"For the vector {vector} the unit vector is {unit_vector}.")

# For the 8, 8 vector:
vector = [8, 8]
unit_vector = cuv(vector)
print(f"For the vector {vector} the unit vector is {unit_vector}.")

# For the 0, 0 vector:
vector = [0, 0]
unit_vector = cuv(vector)
print(f"For the vector {vector} the unit vector is {unit_vector}.")
```

```
For the vector [3, 4] the unit vector is [0.6 0.8].
For the vector [8, 8] the unit vector is [0.70710678 0.70710678].
For the vector [0, 0] the unit vector is None.
```

Exercise 4.

You know how to create unit vectors; what if you want to create a vector of any arbitrary magnitude? Write a Python function that will take a vector and a desired magnitude as inputs and will return a vector in the same direction but with a magnitude corresponding to the second input.

In [ ]:
```
################################################################################
#
# scale_vector -  Scales a vector to a desired magnitude while maintaining
#                 its direction.
#
# Args:
#      vector: A numpy array, list or tuple representing the vector (x, y, ..., z).
#      magnitude: The desired magnitude for the scaled vector.
#
#  Returns:
#      A NumPy array representing the scaled vector.
################################################################################
def scale_vector(vector, magnitude):

  vector = np.array(vector)  # Ensure vector is a NumPy array

  # Handle zero vector case
  if np.linalg.norm(vector) == 0:
    return vector  # Return the zero vector itself

  # Calculate unit vector in the same direction
  unit_vector = vector / np.linalg.norm(vector)

  # Scale the unit vector by the desired magnitude
  scaled_vector = magnitude * unit_vector

  return scaled_vector

# For the 3, 4 vector with magnitude 5:
vector = [2, 3]
magnitude = 5
scaled_vector = scale_vector(vector, magnitude)
print(f"For vector {vector} and scale {magnitude} the scaled vector is {scaled_vector}.")

# For the 8,  vector with magnitude 5:
vector = [8, 8]
magnitude = 5
```

```
scaled_vector = scale_vector(vector, magnitude)
print(f"For vector {vector} and scale {magnitude} the scaled vector is {scaled_vector}.")

# For the 0, 0 vector with magnitude 5:
vector = [0, 0]
magnitude = 5
scaled_vector = scale_vector(vector, magnitude)
print(f"For vector {vector} and scale {magnitude} the scaled vector is {scaled_vector}.")
```

```
For vector [2, 3] and scale 5 the scaled vector is [2.77350098 4.16025147].
For vector [8, 8] and scale 5 the scaled vector is [3.53553391 3.53553391].
For vector [0, 0] and scale 5 the scaled vector is [0 0].
```

Exercise 5.

Write a for loop to transpose a row vector into a column vector without using a built-in function or method such as np.transpose() or v.T. This exercise will help you create and index orientation-endowed vectors.

In [ ]:
```
# Sample row vector
row_vector =  np.array([[1, 2, 3]])

# Create an empty list to store the column vector
column_vector_a = np.zeros((3,1),dtype=int)

# Iterate through the rows, should always be 1
for i, aRow in enumerate(row_vector):
  # Iterate through the elements
  for j, element in enumerate(aRow):
    # By switching row, i, and column, j, we
    # transpose the vector
    column_vector_a[j][i] = row_vector[i][j]

column_vector_b = np.transpose(row_vector)

# Print the column vector
print(f"The row vector {row_vector} with shape {row_vector.shape} transposed to columns is:")
print(f"{column_vector_a}")
print(f"with shape {column_vector_a.shape} using a for loop.")
print(f"The row vector {row_vector} with shape {row_vector.shape} transposed to columns is:")
```

```
print(f"{column_vector_b}")
print(f"with shape {column_vector_b.shape} using the numpy transpose function.")
```

```
The row vector [[1 2 3]] with shape (1, 3) transposed to columns is:
[[1]
 [2]
 [3]]
with shape (3, 1) using a for loop.
The row vector [[1 2 3]] with shape (1, 3) transposed to columns is:
[[1]
 [2]
 [3]]
with shape (3, 1) using the numpy transpose function.
```

Exercise 6.

Here is an interesting fact: you can compute the squared norm of a vector as the dot product of that vector with itself. Look back to the equation in Exercise 2 to convince yourself of this equivalence. Then confirm it using Python.

The formula from Exercise 2 is:

$$||V|| = \sqrt{\sum_{i=1}^{n} v_i^2}$$

When squared, this is:

$$||V||^2 = \sum_{i=1}^{n} v_i^2$$

The formula for a dot product is:

$$\delta = \sum_{i=1}^{n} a_i b_i$$

If we take the dot product of a vector with itself this becomes:

$$\delta = \sum_{i=1}^{n} a_i a_i$$

Which can be written as:

$$\delta = \sum_{i=1}^{n} a^2$$

Which is equivalent to the squared norm.

In [ ]:
```
##################################################################################
# Create 10 random vectors with dimension from 2 to 10 and values from 0 to 10.
# Show the squared norm and dot product with itself are the same.
##################################################################################
for i in range(1, 11):
    dimension = np.random.randint(2, 10)
    # Generate random numbers between 0 and 10
    vector = np.random.default_rng().uniform(0,10,dimension)
    print("")
    print(f"The vector has dimension {len(vector)} and the values are {np.array2string(vector, precision=2, floatmode
    squarednorm = np.linalg.norm(vector)**2
    dotproduct = np.dot(vector,vector)
    print(f"The squared norm is {squarednorm:,.2f} and the dot product is {dotproduct:,.2f}. The difference is {squar
```

```
The vector has dimension 8 and the values are [3.23 9.57 7.90 1.56 8.66 9.63 9.59 0.35]
The squared norm is 426.81 and the dot product is 426.81. The difference is 0.00.

The vector has dimension 7 and the values are [2.83 3.84 4.81 5.71 0.08 4.23 1.26]
The squared norm is 97.95 and the dot product is 97.95. The difference is 0.00.

The vector has dimension 9 and the values are [5.69 1.93 1.61 5.15 1.69 7.05 7.33 3.82 3.22]
The squared norm is 196.45 and the dot product is 196.45. The difference is 0.00.

The vector has dimension 9 and the values are [0.44 2.84 3.49 7.83 6.34 9.88 0.89 9.74 3.47]
The squared norm is 327.24 and the dot product is 327.24. The difference is 0.00.

The vector has dimension 8 and the values are [3.85 6.76 7.82 7.33 6.12 8.64 2.51 2.22]
The squared norm is 298.67 and the dot product is 298.67. The difference is -0.00.

The vector has dimension 4 and the values are [6.09 8.96 8.37 5.87]
The squared norm is 221.83 and the dot product is 221.83. The difference is 0.00.

The vector has dimension 7 and the values are [7.00 4.28 4.20 9.42 2.20 0.20 4.21]
The squared norm is 196.46 and the dot product is 196.46. The difference is 0.00.

The vector has dimension 6 and the values are [5.37 4.17 7.81 1.87 6.65 2.75]
The squared norm is 162.59 and the dot product is 162.59. The difference is -0.00.

The vector has dimension 5 and the values are [7.26 5.48 4.59 7.05 8.26]
The squared norm is 221.73 and the dot product is 221.73. The difference is 0.00.

The vector has dimension 6 and the values are [6.76 4.70 6.69 2.23 8.58 3.94]
The squared norm is 206.70 and the dot product is 206.70. The difference is -0.00.
```

Exercise 7.

Write code to demonstrate that the dot product is commutative. Commutative means that $ab=ba$, which, for the vector dot product, means that a^T b=b^T a. After demonstrating this in code, use the following equation to understand why the dot product is commutative.

$$\delta = \sum_{i=1}^{n} a_i b_i$$

The formula for the dot product is clearly commutative. This is because multiplication is commutative. So, multiplying $a_i b_i$ or $b_i a_i$ gives the same value for each element of the sum. Therefore the result of the sum is the same.

In [ ]:
```
################################################################################
# Create 10 pairs of random vectors with dimension from 2 to 10 and values
# from 0 to 10. Calculate the dot product both ways and show the difference
# between the two ways. This shows that the dot product is commutative.
################################################################################
for i in range(1, 11):
    dimension = np.random.randint(2, 10)
    # Generate random numbers between 0 and 10
    vector1 = np.random.default_rng().uniform(0,10,dimension)
    vector2 = np.random.default_rng().uniform(0,10,dimension)
    print("")
    print(f"The vectors have dimension {len(vector1)} and the values are vector 1 {np.array2string(vector1, precision
    dotproduct1 = np.dot(vector1,vector2)
    dotproduct2 = np.dot(vector2,vector1)
    print(f"The dot product (vector 1, vector 2) is {dotproduct1:,.2f} and the dot product (vector 2, vector 1) is {d
```

```
The vectors have dimension 7 and the values are vector 1 [6.71 6.38 2.84 8.78 4.63 9.44 5.84] and vector 2 [3.08 1.41
5.19 9.00 8.84 2.64 5.30]
The dot product (vector 1, vector 2) is 220.12 and the dot product (vector 2, vector 1) is 220.12. The difference is
0.00.

The vectors have dimension 4 and the values are vector 1 [1.46 9.51 6.58 5.83] and vector 2 [0.32 0.95 7.02 9.23]
The dot product (vector 1, vector 2) is 109.45 and the dot product (vector 2, vector 1) is 109.45. The difference is
0.00.

The vectors have dimension 3 and the values are vector 1 [6.42 8.49 2.34] and vector 2 [9.29 6.44 1.92]
The dot product (vector 1, vector 2) is 118.81 and the dot product (vector 2, vector 1) is 118.81. The difference is
0.00.

The vectors have dimension 3 and the values are vector 1 [5.43 4.19 3.67] and vector 2 [2.84 5.96 1.96]
The dot product (vector 1, vector 2) is 47.57 and the dot product (vector 2, vector 1) is 47.57. The difference is 0.
00.

The vectors have dimension 5 and the values are vector 1 [8.63 0.66 6.19 6.51 2.80] and vector 2 [2.63 6.14 5.39 5.58
1.28]
The dot product (vector 1, vector 2) is 100.09 and the dot product (vector 2, vector 1) is 100.09. The difference is
0.00.

The vectors have dimension 6 and the values are vector 1 [6.60 4.95 5.20 3.31 8.81 5.51] and vector 2 [0.07 3.50 0.76
3.48 8.90 5.16]
The dot product (vector 1, vector 2) is 140.09 and the dot product (vector 2, vector 1) is 140.09. The difference is
0.00.

The vectors have dimension 2 and the values are vector 1 [5.70 2.02] and vector 2 [1.09 8.03]
The dot product (vector 1, vector 2) is 22.44 and the dot product (vector 2, vector 1) is 22.44. The difference is 0.
00.

The vectors have dimension 8 and the values are vector 1 [4.66 0.55 8.89 6.76 0.86 1.35 3.97 3.71] and vector 2 [1.44
2.53 4.48 8.04 8.83 3.95 9.08 5.62]
The dot product (vector 1, vector 2) is 172.07 and the dot product (vector 2, vector 1) is 172.07. The difference is
0.00.

The vectors have dimension 4 and the values are vector 1 [6.97 4.83 4.74 0.12] and vector 2 [4.02 9.70 3.94 0.99]
The dot product (vector 1, vector 2) is 93.67 and the dot product (vector 2, vector 1) is 93.67. The difference is 0.
00.

The vectors have dimension 3 and the values are vector 1 [7.16 6.83 4.06] and vector 2 [5.30 1.67 2.26]
```

```
The dot product (vector 1, vector 2) is 58.55 and the dot product (vector 2, vector 1) is 58.55. The difference is 0.
00.
```