

FINAL EXAM PROJECT

Foundations of Data Science:
Programming and Linear Algebra

Students:

Diana Laura Janikowski (143003)

George Kotrotsios (158283)

Jenő Tóth (158386)

Rasmus Kongstad Thomsen (137041)

Group number: Fri-136342-18

MSc in Business Administration and Data Science
Copenhagen Business School

Date of submission: 11. December 2022

Number of pages: 9

Number of characters: 17,078

Table of Contents

1	Question 1 - Sub-Numpy	2
1.1	Report	2
1.2	Creating the functions	4
1.3	Testing the functions	4
1.4	Exception Handling	4
2	Question 2 - Hamming's code	5
2.1	Report	5
2.2	Creating the functions	5
2.3	Testing the functions	5
2.4	Exception Handling	5
3	Question 3 - Text Document Similarity	6
3.1	Report	6
3.2	Creating the functions	8
3.3	Testing the functions	8
3.3.1	Testing on poems	8
3.3.2	Testing on random paragraphs	8
3.4	Exception Handling	8
3.5	Discussion of methods	8

1 Question 1 - Sub-Numpy

1.1 Report

For the first question we were asked to complete ten different implementations in Python without using Numpy. To complete the assignment, we have created a class called SNumPy in which we have defined ten functions to solve the ten problems. All ten functions have been made to solve problems regarding vector and matrix operations, and therefore we have implemented in our code mathematical rules based on the book Linear Algebra and its Applications (Lay et al., 2016). The overall code involves the use of basic Python functions and list comprehensions. We have tested the use of the code during the development and have created error messages for potential issues.

The ten functions we have created to solve the ten problems are:

1. **snp.ones():** This function takes one input that has to be an integer and returns an array of the number one with the length of the input integer. The function uses a for loop to run through the list in the range of the input. If the input isn't an integer the function will return a message saying, "Input has to be an integer!".
2. **snp.zeros():** This function works almost the way as the previous one – takes an integer as an input and, differing from the first it returns a list of zeros. Still loops through the list in the range of the input and gives the same error message if the input is anything other than an integer.
3. **snp.reshape():** The purpose of this function is to take a list of numbers and turn that array into a matrix based on a given number of rows and columns. The function takes three inputs, an array and two integers. We create a list for the matrix values to be stored in and run through two for loops that creates an empty list for the row and then inserts the corresponding values into the matrix. The values are put into the matrix and the row using the append function. There are a few criteria that must be met before this function can work. To ensure that the array can be transformed into a matrix the length of the array must be equal to the product of the number of rows and columns, and of course the inputs for the rows and columns must be integers.
4. **snp.shape():** This function will, based on an input, return the dimension of the input. If the input is a matrix, it will return the number of rows, and columns and if the input is a vector, it will return the dimension of the vector. The function checks whether the input is a matrix or vector by looking at whether the first element is a list or an integer. If neither is the case, then it will alert a TypeError.

5. **snp.append():** This function takes two inputs that have to be either vectors or matrices and then combines the two. The function uses the shape function from the previous question to determine the shape of the input and thereby if it is compatible. If the input is neither a vector or a matrix or not the same thing the function will return an error saying, “The given arrays do not have the same number of rows.”.
6. **snp.get():** The purpose of this function is to determine the value in a vector or matrix at a specified row and column. The function takes three inputs which are an array, and the number of the row and column.
7. **snp.add():** This function takes two inputs which are vectors or matrices and for the function to work they need to be the same shape. Therefore, the shape function from earlier is used to check if that is the case. If that isn’t the case an error message saying, “The shapes of the given arrays do not match.” will appear otherwise it will move on to add the two together. We create a list containing zeros and reshape it to be the same shape as the input. We then iterate over the rows and columns to add the appropriate values together and add them to the list of zeros.
8. **snp.subtract():** This function works almost the way as the previous one – takes two matrices or vectors as inputs and instead calculates the difference between the two. Still iterate over the rows and columns to add the appropriate values together and add them to the list of zeros.
9. **snp.dotproduct():** This function takes two arrays as inputs and uses them to calculate the dot product between the two. The function first checks the shape of the input to determine whether they are matrices or vectors and to see if the number of rows in one matches the number of columns in the other. If that is not the case it will display the error message saying, “The number of columns in array1 does not equal the number of rows in array2.”. If both inputs are vectors or matrices that match the criteria the function will calculate the dot product by iterating through all the elements.
10. **snp.gaussian():** This function takes two linear equations as arguments and solves them based on Gaussian elimination and row reduction rules. The function checks if the equations contain the same number of unknowns, if the right-hand side of equations is a vector, and whether the same number of equations is provided. Besides that the function also involves checks for data types and division by zero.

1.2 Creating the functions

See Appendix [1]

1.3 Testing the functions

See Appendix [2]-[12]

1.4 Exception Handling

See Appendix [13]-[20]

2 Question 2 - Hamming's code

2.1 Report

There are three parts for Hamming's code to work, which are the encoding, the parity check, and the decoding. We used the function `np.matmul` which returns the matrix product of two arrays, and in order to get the binary result of 0 or 1 we used the modulo 2 operation, which is also the main part of the solution we provided. The encoding works by taking a four-bit input and multiplying the message with the generator matrix. This will return an encoded seven-bit Hamming code word ([Hamming, 1950](#)).

After this we will run the parity check to ensure that the input message has been encoded correctly, and this is done by multiplying the parity check matrix with the input. If this results in a zero vector, we know that the message has been encoded correctly and we can move on to decode the message. If the result is not a zero vector then a message that the message did not pass the parity check is provided.

The decoding works by multiplying the encoded message with the decoder matrix. If the functions are correct, this should return the original four-bit message. If this is the case, the function returns "All calculations are correct." We have included a number of error messages if the functions don't work correctly, which include the parity check not being passed, if the input message is not binary, and a `ValueError` in case the input is not four-bits.

2.2 Creating the functions

See Appendix [21]-[26]

2.3 Testing the functions

See Appendix [27]-[30]

2.4 Exception Handling

See Appendix [31]-[32]

3 Question 3 - Text Document Similarity

3.1 Report

There is a number of ways to determine text document similarity. We looked at three measures: the Damerau-Levenshtein distance (Mays et al., 1991), the dot product distance, and the Euclidean distance (Jena, 2021). For every method, we had a function which would take a look at two strings to determine their distance, and another function, which had two inputs: a list of strings to rank, and a search string to compare them to. The second function would use the first function to determine every strings' distance from the search string, then return a list of these strings in descending order of similarity. The inputs for all three functions were the same: a list of lists, containing the strings to be ranked as well as their titles (such as ["String 1 title", "String 1 text", [etc.]]), and a search string that was compared to all of the other strings. Exception handling was done in the same way for all three functions. If a user input was not a string, the function would send a warning message, but it ran nonetheless, converting the input into string.

The first method, the Damerau-Levenshtein distance, considers the minimum distance between strings (Mays et al., 1991). It uses four operations: insertions, substitutions, deletions and transpositions. From these, it calculates the smallest possible number of operations which are necessary in order to convert string A into string B (Mays et al., 1991). Insertion involves adding a character, substitution involves changing a character for another, deletion means removing a character and the last operation, transposition, changes the place of two characters. This method can be improved, but it serves as a baseline for our further models (Brill and Moore, 2000). For our implementation, we used it to look at two unaltered input strings, and simply calculate their Damerau-Levenshtein distance. In the ranking function, these strings were ranked such as the higher the distance, the lower the similarity with the search string.

The second and the third methods were similar in build-up, but different in the distance calculations. Instead of simply looking at the strings, the two methods first manipulated the inputs. This is because they are not concerned with the characters within a string, but the words that make up the string. A helping function does this manipulation, that both of them used later on. The function first handles exceptions, then deletes every character that is not a letter, number, or space. After this, it turns uppercase letters into lower cases. This is done to eliminate differences between words that should be the same, but have different forms (for example James' and james). The function then creates a list of all the words in all the strings without duplicates. This gives us a dictionary of our text corpus. The

output of the function is a 2 dimensional matrix, where in one dimension are the words, on the other the strings (including the search string), and in their intersections the weights. Weights could be assigned using multiple methods. The simplest one would be to look at how many times a given word appears in a string. This, however, would give common words such as "the" and "and" larger weights, even though they are generally less useful when looking at text similarities. Because of this, we implemented a Term Frequency/Inverse Document Frequency calculation (Rathi and Mustafi, 2022; Nguyen, 2014). The calculation gives a non-negative weight to each word based on how many strings it appears in (if a word appears in every string, its weight is 0, as it cannot be used to determine similarity). We used these weights, which filled our matrix, giving us vectors for every string which we could use for our other two methods. Other weights could have also been used, which maybe went a step further, and either looked at similarities in meaning (such as "many" and "lots"), or similarities in forms (such as "cat" and "cats").

The second method we used was the dot product distance (Lay et al., 2016). The concept behind it is that we have a number of elements in the search string vector that are not zero, the words that appear in that string. When taking the dot product of the search string vector and another string vector, only these values will increase the distance (in this model, higher "distance" means that the two strings are more similar). The main issue with this model is that if we take a shorter and a longer string of random words, the longer one will have more words in common with the search string. For example, if the search string is "This house is very nice", and the two strings to compare are "This house is nice" and a very long string containing all the words in the English dictionary, the latter string would score better, even though it is less similar. Because of this, we divided the dot products by their respective string lengths (and multiplied them by 100 for better displaying). From these, we gathered the final rankings which were then displayed.

The third method, the Euclidean distance, calculates the distance between two vectors as well. It is calculated as the square root of the sum of squared differences (Lay et al., 2016). This means that it increases distance for words that are not the same, meaning that the smaller the output distance is, the larger text similarity is. Once again, the lengths of the strings proved to be an issue, however, this time we couldn't divide them by their lengths. On the one hand, longer texts may have more common words with the search string than shorter texts, which would indicate lower distance, however, on the other hand longer texts may also have a number of words that do not appear in the search string, indicating higher distance. Because of this, our hypothesis is that this model works best for strings that are similar in lengths.

While creating the functions, we also looked at a set of randomly generated short strings, to see that the functions work as intended. After this, we tested them on two sets of longer texts: a list of 10 poems, and a randomly generated list of 6 paragraphs with similar lengths. To see how useful the methods are, we ran correlations between the distances and the strings' lengths. Our assumption is that if there is high correlation, it indicates that the method does not detect text similarity of characters (in the Damerau–Levenshtein model) or similarity of words (in the dot product and Euclidean models), but only similarities of lengths.

3.2 Creating the functions

See Appendix [33]-[43]

3.3 Testing the functions

3.3.1 Testing on poems

Users can also input strings from texts. We tested the methods by using 10 poems: 3 by Edgar Allan Poe, 3 by Walt Whitman, 3 by Oscar Fingal O'Flahertie Wills Wilde, and the search string by William Shakespeare).

See Appendix [44]-[51]

3.3.2 Testing on random paragraphs

We also looked at a number of randomly generated paragraphs. See Appendix [52]-[59]

3.4 Exception Handling

The functions can only work on strings. Because any recognizable inputs (such as integers) can be converted to strings, the function still works, returning a warning to the users, then completing the calculations with the converted inputs. See Appendix [60]

3.5 Discussion of methods

As we have seen, the Damerau-Levenshtein distance is highly correlated with string lengths. This is not surprising, as the difference between two strings' lengths is added to the distance by a number of insertion operations. Even when the strings were similar in lengths, a high correlation still persisted, which means that for longer texts, the method is not very useful. Despite this, it can be a better estimator for shorter strings of one or only a few words.

When there are no similar words between texts, the other two methods cannot be used, giving room to the Damerau-Levenshtein ranks.

The dot product distance ranking performed well both when the strings were of different lengths, and when they were similar. This is once again not surprising, as we have previously accounted for string lengths when we divided the dot products by them. Because of this, the dot product distance is our best estimator for text similarities. It can be used when the texts have at least a few similar words, and it gives an accurate answer when determining ranks, taking word frequencies into consideration as well.

Finally, the Euclidean distance did not perform well in the first test, when there were large variances in string lengths. Despite this, in the second test there was a much smaller correlation between string lengths and the distances. The ranks were almost identical to the dot product ranks as well. As such, we recommend using the algorithm as an additional test to compliment the dot product method in texts which are similar in length.

References

- Brill, E. and Moore, R. C. (2000). An improved error model for noisy channel spelling correction. ACL '00, page 286–293, USA. Association for Computational Linguistics.
- Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160.
- Jena, S. (2021). Euclidean distance based similarity measurement and ensuing ranking scheme for document search from outsourced cloud data.
- Lay, D. C., Lay, S. R., and McDonald, J. J. (2016). *Linear algebra and its applications*. Pearson.
- Mays, E., Damerau, F. J., and Mercer, R. L. (1991). Context based spelling correction. *Information Processing Management*, 27(5):517–522.
- Nguyen, E. (2014). Chapter 4 - text mining and network analysis of digital libraries in r. In Zhao, Y. and Cen, Y., editors, *Data Mining Applications with R*, pages 95–115. Academic Press, Boston.
- Rathi, R. N. and Mustafi, A. (2022). The importance of term weighting in semantic understanding of text: A review of techniques. *Multimedia Tools and Applications*.

Appendix

```
[1]: class SNumPy:

    #1)
    #Define a function that takes an int parameter and returns a list of
    →length Int
    #containing the number 1 Int amount of times
    def ones(Int):
        if type(Int) != int:
            return "Input has to be integer!"
        else:
            return [1 for i in range(Int)]

    #2)
    #The same concept as the function above, the only difference is the
    →use of the number 0 instead of 1
    def zeros(Int):
        if type(Int) != int:
            return "Input has to be integer!"
        else:
            return [0 for i in range(Int)]

    #3)
    #Define a function taking the following as parameters: array, number
    →of rows and number of columns
    #And reshaping the array into a matrix with dimensions according to
    →the number of rows and columns
    #Based on: https://www.youtube.com/watch?v=FF29bm8j6kQ

    def reshape(array, row, column):
        if type(row) == int:
            if type(column) == int:
```

```

        if row * column == len(array):
            matrix = []
            for r in range(row):
                listrow = []
                for c in range(column):
                    listrow.append(array[r * column + c])
                matrix.append(listrow)
            return matrix
        else:
            return "Cannot reshape array of size " + str(len(array)) + " into shape (" + str(row) + ", " + str(column) + ")."
        else:
            return "Row and column must be integers!"
    else:
        return "Row and column must be integers!"

```

#4)

```

def is_vector(array):
    try:
        array = array[0][0]
    except TypeError:
        return True
    else:
        return False

```

*#Define a function taking an array as an argument and return its
 → number of rows and columns*

#Based on: <https://www.pythonpool.com/list-shape-python/>

```

def shape(array):
    if type(array) == int:
        rows = 1
        columns = 1
    else:
        rows = len(array)          #Take the length of the array
        if type(array[0]) == int:

```

```

        columns = 1
    else:
        columns = len(array[0])    #Take the length of the first
→item in the array
    shape = (rows, columns)
    return shape

#5
    #Define a function taking two arrays as arguments and return their
→combination
    #Based on: https://www.geeksforgeeks.org/
→python-merge-two-list-of-lists-according-to-first-element/
    def append(array1, array2):
        shape1 = SNumPy.shape(array1)    #Get the shape of array1
        shape2 = SNumPy.shape(array2)    #Get the shape of array2
        #Check if the arrays have the same number of rows
        if (shape1[0] == shape2[0]):
            if SNumPy.is_vector(array1):
                array1 = [array1]
                array2 = [array2]
            return [a + b for (a, b) in zip(array1, array2)]
        else:
            return "The given arrays do not have the same number of rows."

#6
    #Define a function taking an array as an argument and return the
→value specified by the row and column
    def get(array, row, column):
        value = [array[row - 1][column - 1]]
        return value

#7)

```

```

#Define a function taking two arrays as arguments and return their sum
#Based on: https://www.youtube.com/watch?v=-sAmg3yoVhI
def add(array1, array2):
    shape1 = SNumPy.shape(array1)      #Get the shape of array1
    shape2 = SNumPy.shape(array2)      #Get the shape of array2
    #Check if the arrays have the same shape
    if (shape1[0] == shape2[0] and shape1[1] == shape2[1]):
        if SNumPy.is_vector(array1):
            array1 = [array1]
            array2 = [array2]
            #Create an array with 0s to be used for the final calculation
→based on the shape of the arrays
            thesum = SNumPy.zeros(shape1[0] * shape1[1])
→#Create a list with 0s
            thesum = SNumPy.reshape(thesum, shape1[0], shape1[1])
→#Reshape the list
            for i in range(shape1[0]):
                for k in range(shape1[1]):
                    thesum[i][k] = array1[k][i] + array2[k][i]
            return thesum
        else:
            return "The shapes of the given arrays do not match."

#8)
#Define a function taking two arrays as arguments and return the
→difference
#Based on: https://www.youtube.com/watch?v=-sAmg3yoVhI
def subtract(array1, array2):
    shape1 = SNumPy.shape(array1)      #Get the shape of array1
    shape2 = SNumPy.shape(array2)      #Get the shape of array2
    #Check if the arrays have the same shape
    if (shape1[0] == shape2[0] and shape1[1] == shape2[1]):
        if SNumPy.is_vector(array1):
            array1 = [array1]
            array2 = [array2]

```

```

        #Create an array with 0s to be used for the final calculation
→based on the shape of the arrays
        difference = SNumPy.zeros(shape1[0] * shape1[1])
→ #Create a list with 0s
        difference = SNumPy.reshape(difference, shape1[0], shape1[1])
→ #Reshape the list
        for i in range(shape1[0]):
            for k in range(shape1[1]):
                difference[i][k] = array1[k][i] - array2[k][i]
        return difference
    else:
        return "The shapes of the given arrays do not match."

#9)
    #Define a function taking two arrays as arguments and return dot
→product
    #Based on: https://stackoverflow.com/questions/66427420/
→multiplying-matrixes-without-numpy
    def dotproduct(array1, array2):
        shape1 = SNumPy.shape(array1)        #Get the shape of array1
        shape2 = SNumPy.shape(array2)        #Get the shape of array2
        #Check if the number of columns in array1 equals the number of
→rows in array2
        if SNumPy.is_vector(array1):
            array1 = [array1]
        if SNumPy.is_vector(array2):
            array2 = [array2]
        if (shape1[1] == shape2[0]):
            dotproduct = []
            for m in range(0, len(array1)):
                rows = []
                for i in range(0, len(array2[0])):
                    columns = 0
                    for j in range(0, len(array2)):

```

```

        columns += array1[m][j] * array2[j][i]
        rows.append(columns)
        dotproduct.append(rows)
    return dotproduct
else:
    return "The number of columns in array1 does not equal the
→number of rows in array2."

#10)
#Solver for a system of linear equations using Gaussian elimination
#and row reduction rules for the functionality
#based on https://www.codesansar.com/numerical-methods/
→gauss-elimination-method-python-program.htm
def gaussian(a,b):
    for i in range(1,len(a)):
        if len(a[i]) != len(a[i-1]):
            return "All equations must contain the same number of
→unknowns. E.g. if equations are  $kx_0 + lx_1 = a$ ,  $mx_0 + nx_1 = b$ ,
→input arrays must be  $[[k,l],[m,n]]$  and  $[a,b]$ "
        if SNumPy.is_vector(b) != True:
            return "Right-hand side of the equations must be given as a
→vector. E.g. if equations are  $kx_0 + lx_1 = a$ ,  $mx_0 + nx_1 = b$ ,
→input arrays must be  $[[k,l],[m,n]]$  and  $[a,b]$ "
        elif len(b) != len(a):
            return "Same amount of equations must be given. E.g. if
→equations are  $kx_0 + lx_1 = a$ ,  $mx_0 + nx_1 = b$ , input arrays must
→be  $[[k,l],[m,n]]$  and  $[a,b]$ "
    else:
        for i in range(len(a)):
            for j in range(len(a[0])):
                try:
                    float(a[i][j])
                except ValueError:
                    return "Input must be integer or float!"

```

```

        for i in range(len(b)):
            try:
                float(b[i])
            except ValueError:
                return "Input must be integer or float!"

    m = SNumPy.reshape(snp.
→zeros(len(a)*len(a[0])+len(b)),len(a),len(a[0])+1)
    n = SNumPy.zeros(len(a))
    for i in range(len(a[0])):
        for j in range(len(a)):
            m[i][j] = a[i][j]
            m[i][j+1] = b[i]
    for i in range(len(a)):
        if m[i][i] == 0.0:
            return 'Divide by zero detected!'
        for j in range(i+1, len(a)):
            ratio = m[j][i]/m[i][i]
            for k in range(len(a)+1):
                m[j][k] = m[j][k] - ratio * m[i][k]
    n[len(a)-1] = m[len(a)-1][len(a)]/m[len(a)-1][len(a)-1]

    for i in range(len(a)-2,-1,-1):
        n[i] = m[i][len(a)]

        for j in range(i+1,len(a)):
            n[i] = n[i] - m[i][j]*n[j]

        n[i] = n[i]/m[i][i]
    return n

```

[2]: *#Shorthand reference:*
snp = SNumPy

1) SNumPy.ones(Int):

```
[3]: print(snp.ones(5))  
      print(snp.ones(10))
```

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

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

2) SNumPy.zeros(Int):

```
[4]: print(snp.zeros(3))  
      print(snp.zeros(7))
```

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

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

3) SNumPy.reshape(array, row, column):

```
[5]: arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]  
      print(snp.reshape(arr, 6, 2))  
      print(snp.reshape(arr, 3, 4))  
      print(snp.reshape(arr, 4, 3))
```

```
[[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10, 11]]
```

```
[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]]
```

```
[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]
```

4) SNumPy.shape(array):

```
[6]: arr2 = [[0, 1, 2, 3], [3, 4, 5, 6], [6, 7, 8, 9]]  
      arr3 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]  
      print(snp.shape(arr2))  
      print(snp.shape(arr3))  
      #Test with vector:  
      v = [2, 4, 6]  
      print(snp.shape(v))  
      #Test with single value:  
      print(snp.shape(5))
```

```
(3, 4)
```

```
(3, 3)
```

```
(3, 1)
```

```
(1, 1)
```

5) SNumPy.append(array1, array2):

```
[7]: #Test with 3x4 and 3x3:
print("3x4 and 3x3:")
print(arr2)
print(arr3)
print(snp.append(arr2, arr3))

#Test with 3x3 and 3x3:
print("3x3 and 3x3:")
arr4 = [[5, 1, 3], [2, 5, 5], [8, 7, 2]]
print(arr3)
print(arr4)
print(snp.append(arr3, arr4))

#Test with vector:
v2 = [3, 5, 8]
print("1x3 and 1x3:")
print(snp.append(v, v2))
```

3x4 and 3x3:

```
[[0, 1, 2, 3], [3, 4, 5, 6], [6, 7, 8, 9]]
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
[[0, 1, 2, 3, 0, 1, 2], [3, 4, 5, 6, 3, 4, 5], [6, 7, 8, 9, 6, 7, 8]]
```

3x3 and 3x3:

```
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
[[5, 1, 3], [2, 5, 5], [8, 7, 2]]
[[0, 1, 2, 5, 1, 3], [3, 4, 5, 2, 5, 5], [6, 7, 8, 8, 7, 2]]
```

1x3 and 1x3:

```
[[2, 4, 6, 3, 5, 8]]
```

6) SNumPy.get(array, row, column):

```
[8]: print(arr4)
#The indexing starts from 1
print(snp.get(arr4, 3, 1))
```

```
[[5, 1, 3], [2, 5, 5], [8, 7, 2]]
[8]
```

7) SNumPy.add(array1, array2):

```
[9]: #Test with 3x3 and 3x3:  
print(snp.add(arr3, arr4))  
  
#Test with 1x3 vectors:  
print(snp.add(v, v2))
```

```
[[5, 5, 14], [2, 9, 14], [5, 10, 10]]  
[[5], [9], [14]]
```

8) SNumPy.subtract(array1, array2):

```
[10]: #Test with 3x3 and 3x3:  
print(snp.subtract(arr3, arr4))  
  
#Test with 1x3 vectors:  
print(snp.subtract(v2, v))
```

```
[[ -5,  1, -2], [ 0, -1,  0], [-1,  0,  6]]  
[[1], [1], [2]]
```

9) SNumPy.dotproduct(array1, array2):

```
[11]: #Test with 3x3 and 3x3:  
print(snp.dotproduct(arr3, arr4))  
  
#Test with 3x4 and 4x3:  
arr5 = [[8, 2, 3], [2, 7, 1], [8, 9, 0], [5, 2, 1]]  
print(snp.dotproduct(arr2, arr5))
```

```
[[18, 19, 9], [63, 58, 39], [108, 97, 69]]  
[[33, 31, 4], [102, 91, 19], [171, 151, 34]]
```

10) SNumPy.gaussian(a, b):

```
[12]: a = [[1,3,-2], [3,5,6], [2,4,3]]  
      b = [5,7,8]  
      snp.gaussian(a,b)
```

```
[12]: [-15.0, 8.0, 2.0]
```

1) SNumPy.ones(Int):

```
[13]: print(snp.ones("test"))
```

Input has to be integer!

2) SNumPy.zeros(Int):

```
[14]: print(snp.zeros("test"))
```

Input has to be integer!

3) SNumPy.reshape(array, row, column):

```
[15]: print(snp.reshape(arr, 4, "test"))  
print(snp.reshape(arr, 3, 5))
```

Row and column must be integers!

Cannot reshape array of size 12 into shape (3,5).

5) SNumPy.append(array1, array2):

```
[16]: #Test with 3x4 and 4x3:  
arr5 = [[8, 2, 3], [2, 7, 1], [8, 9, 0], [5, 2, 1]]  
print("3x4 and 4x3:")  
print(snp.append(arr2, arr5))
```

3x4 and 4x3:

The given arrays do not have the same number of rows.

7) SNumPy.add(array1, array2):

```
[17]: #Test with 3x4 and 3x3:  
print(snp.add(arr2, arr3))  
#Test with 3x4 and 4x3:  
print(snp.add(arr2, arr5))
```

The shapes of the given arrays do not match.

The shapes of the given arrays do not match.

8) SNumPy.subtract(array1, array2):

```
[18]: #Test with 3x4 and 3x3:  
print(snp.subtract(arr2, arr3))  
#Test with 3x4 and 4x3:  
print(snp.subtract(arr2, arr5))
```

The shapes of the given arrays do not match.

The shapes of the given arrays do not match.

9) SNumPy.dotproduct(array1, array2):

```
[19]: #Test with 3x4 and 3x3:  
print(snp.dotproduct(arr2, arr3))
```

The number of columns in array1 does not equal the number of rows in array2.

10) SNumPy.gaussian(a, b):

```
[20]: #Test with  
a = [[1,3,-2], [3,5,6],[2,4,3]]  
b = [5,7,8,9]  
print(snp.gaussian(a,b))  
c = [[3,-2], [3,5,6],[2,4,3]]  
d = [5,7,8]  
print(snp.gaussian(c,d))
```

Same amount of equations must be given. E.g. if equations are $k \cdot x_0 + l \cdot x_1$

↪ = a,

$m \cdot x_0 + n \cdot x_1 = b$, input arrays must be $[[k,l],[m,n]]$ and $[a,b]$

All equations must contain the same number of unknowns. E.g. if equations

↪ are

$k \cdot x_0 + l \cdot x_1 = a$, $m \cdot x_0 + n \cdot x_1 = b$, input arrays must be $[[k,l],[m,n]]$ and $[a,b]$

```
[21]: #Importing Numpy
```

```
[22]: import numpy as np
```

```
[23]: #Generating the necessary matrixes
```

```
[24]: # Hamming's Generator Matrix  
G=np.array([[1,1,0,1],  
            [1,0,1,1],  
            [1,0,0,0],  
            [0,1,1,1],  
            [0,1,0,0],
```

```

        [0,0,1,0],
        [0,0,0,1]])

#Check array
H= np.array([[1,0,1,0,1,0,1],
             [0,1,1,0,0,1,1],
             [0,0,0,1,1,1,1]])

#decoder
R=np.array([[0,0,1,0,0,0,0],
            [0,0,0,0,1,0,0],
            [0,0,0,0,0,1,0],
            [0,0,0,0,0,0,1]])

```

[25]: *#Defining the functions*

```

[26]: def Hamming(array):
    try:
        if ((array==0) | (array==1)).all():
            encoded = np.matmul(G,array) % 2
            print("The 7 bit code is:")
            print(encoded)

            print("Parity test:")
            parity = np.matmul(H,encoded) % 2
            print(parity)
            if sum(abs(parity)) == 0:
                print("Parity check passed")

            decoded = np.matmul(R, encoded)
            print("The decoded vector:")
            print(decoded)
            if sum(array == decoded) == 4:
                return "All calculations are correct."
            else:
                return "Errors in calculation!"

```

```

        else:
            return "Parity check not passed!"
    else:
        return "Input vector is not binary!"

except ValueError:
    return "The vector has to be 4 bits!"

```

```

[27]: #Test cases
arr1 = np.array([1,0,1,1])
Hamming(arr1)

```

```

The 7 bit code is:
[0 1 1 0 0 1 1]
Parity test:
[0 0 0]
Parity check passed
The decoded vector:
[1 0 1 1]

```

```

[27]: 'All calculations are correct.'

```

```

[28]: arr2 = np.array([0,0,0,1])
Hamming(arr2)

```

```

The 7 bit code is:
[1 1 0 1 0 0 1]
Parity test:
[0 0 0]
Parity check passed
The decoded vector:
[0 0 0 1]

```

```

[28]: 'All calculations are correct.'

```

```

[29]: arr3 = np.array([1,0,1,0])
Hamming(arr3)

```

```

The 7 bit code is:

```

```
[1 0 1 1 0 1 0]
Parity test:
[0 0 0]
Parity check passed
The decoded vector:
[1 0 1 0]
```

[29]: 'All calculations are correct.'

```
[30]: arr4 = np.array([0,1,1,0])
      Hamming(arr4)
```

```
The 7 bit code is:
[1 1 0 0 1 1 0]
Parity test:
[0 0 0]
Parity check passed
The decoded vector:
[0 1 1 0]
```

[30]: 'All calculations are correct.'

```
[31]: arr6 = np.array([2,0,1,2])
      Hamming(arr6)
```

[31]: 'Input vector is not binary!'

```
[32]: arr7 = np.array([0,0,1])
      Hamming(arr7)
```

[32]: 'The vector has to be 4 bits!'

```
[33]: # First, let's create the text corpus, as well as the search string:
      # The strings of the text corpus must be given in a list of lists, with
      →their titles:
      # (Generated by https://randomwordgenerator.com/paragraph.php)
      input_corpus = [
          ["String 1", "The alarm went off at exactly 6:00 AM as it had every
          →morning for the past five years"],
```

```

["String 2", "Barbara began her morning and was ready to eat_
↳breakfast by 7:00 AM"],

["String 3", "The day appeared to be as normal as any other, but that_
↳was about to change"],

["String 4", "In fact, it was going to change at exactly 7:23 AM"],

["String 5", "It was supposed to be a dream vacation"],

["String 6", "They had planned it over a year in advance so that it_
↳would be perfect in every way"],

["String 7", "It had been what they had been looking forward to_
↳through all the turmoil and negativity around them"],

["String 8", "It had been the light at the end of both their tunnels"]
]
# Next, the search string:
input_search = "Now that the dream vacation was only a week away, the_
↳virus had stopped all air travel"

```

```

[34]: def Dam_Lev_dist(str1, str2):
    # The function uses https://en.wikipedia.org/wiki/
    ↳Damerau%E2%80%93Levenshtein_distance
    # We need to make sure that the inputs are strings:
    if type(str1) != str:
        print("Input should be string. It has been converted to string.")
        str1 = str(str1)
    if type(str2) != str:
        print("Input should be string. It has been converted to string.")
        str2 = str(str2)
    # The script goes through the two strings to look at the minimum_
    ↳number of operations in steps

```

```

oneago = None
thisrow = list(range(1, len(str2) + 1)) + [0]
for x in range(len(str1)):
    twoago, oneago, thisrow = oneago, thisrow, [0] * len(str2) + [x +
→1]

    for y in range(len(str2)):
        # Removing a character
        delcost = oneago[y] + 1
        # Adding a character
        addcost = thisrow[y - 1] + 1
        # Substituting a character
        subcost = oneago[y - 1] + (str1[x] != str2[y])
        thisrow[y] = min(delcost, addcost, subcost)
        # Switching two characters
        if (x > 0 and y > 0 and str1[x] == str2[y - 1]
            and str1[x-1] == str2[y] and str1[x] != str2[y]):
            thisrow[y] = min(thisrow[y], twoago[y - 2] + 1)
    return thisrow[len(str2) - 1]

```

```

[35]: def Dam_Lev_ranking(corpus, search_string):
    # Creating two lists of corpus: titles and strings:
    corpus_titles = [x[0] for x in corpus]
    corpus_string = [x[1] for x in corpus]
    # Checking that the inputs are strings:
    for i in range(len(corpus_string)):
        if type(corpus_string[i]) != str:
            print("User inputs should be string. We converted the values
→that aren't.")
            corpus_string[i] = str(corpus_string[i])
    if type(search_string) != str:
        print("Search string should be string. We converted it to string.
→")
        search_string = str(search_string)
    # Calculating the distances based on the above function:
    output_ranks = []

```

```

for i in range(len(corpus_string)):
    output_ranks.append(Dam_Lev_dist(corpus_string[i],search_string))
# Determining the ranks based on the distances
final_ranking = []
place = 1
tie_breaker = 0
k = 0
# Creating output titles (the first 50 characters of the strings only)
output_titles = []
for i in range(len(corpus_string)):
    if len(corpus_string[i]) > 50:
        output_titles.append((corpus_string[i][:50] + "..."))
    else:
        output_titles.append(corpus_string[i])
while k <= max(output_ranks):
    for i in range(len(output_ranks)):
        if output_ranks[i] == k:
            if output_ranks.count(k) > tie_breaker:
                # Assigning the ranks
                final_ranking.append("Rank #" + str(place) + " (" +
→str(output_ranks[i]) +
                                " distance): " +
→str(corpus_titles[i]) + ": " + str(output_titles[i]))
                tie_breaker += 1
            if tie_breaker > 0 and tie_breaker == output_ranks.
→count(k):
                place += 1
            tie_breaker = 0
            k += 1
    return final_ranking

```

[36]: Dam_Lev_ranking(input_corpus, input_search)

[36]: ['Rank #1 (60 distance): String 8: It had been the light at the end of
→both
their tun...',

```

'Rank #2 (61 distance): String 3: The day appeared to be as normal as any
other, but...',
'Rank #3 (67 distance): String 2: Barbara began her morning and was
→ready to
eat bre...',
'Rank #3 (67 distance): String 4: In fact, it was going to change at
→exactly
7:23 AM',
'Rank #4 (68 distance): String 1: The alarm went off at exactly 6:00 AM
→as it
had ev...',
'Rank #4 (68 distance): String 5: It was supposed to be a dream
→vacation',
'Rank #4 (68 distance): String 6: They had planned it over a year in
→advance so
that...',
'Rank #5 (73 distance): String 7: It had been what they had been looking
forward to ...']

```

[37]: *# We will use logarithms in the function*

```

import math
# We will use dot products
import numpy as np

```

[38]: `def word_distance_matrix(corpus, search_string):`

```

    # We use a Term Frequency / Inverse Document Frequency model based on
    # https://en.wikipedia.org/wiki/Tf%E2%80%93idf
    # Creating two lists of corpus: titles and strings:
    corpus_titles_wdm = [x[0] for x in corpus]
    corpus_string_wdm = [x[1] for x in corpus]
    # Checking that the inputs are strings:
    for i in range(len(corpus_string_wdm)):
        if type(corpus_string_wdm[i]) != str:
            print("User inputs should be string. We converted the values
→that aren't.")
            corpus_string_wdm[i] = str(corpus_string_wdm[i])

```

```

if type(search_string) != str:
    print("Search string should be string. We converted it to string.
→")

    search_string = str(search_string)
    # Removing all characters from the strings that are not letters,
→numbers, or spaces:
    whitelist = set('abcdefghijklmnopqrstuvwxyz
→ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789')

    search_string = ''.join(filter(whitelist.__contains__, search_string))
    # Turning uppercase characters into lower case:
    search_string = search_string.lower()
    for i in range(len(corpus_string_wdm)):
        corpus_string_wdm[i] = ''.join(filter(whitelist.__contains__,
→corpus_string_wdm[i]))
        corpus_string_wdm[i] = corpus_string_wdm[i].lower()
    # Separating all words in all strings:
    search_string_separated = search_string.split()
    corpus_separated = [None] * len(corpus_string_wdm)
    for i in range(len(corpus_string_wdm)):
        corpus_separated[i] = corpus_string_wdm[i].split()
    # We use an inverse document frequency to weight the words
    # First, we create a dictionary of each word in the strings:
    dictionary = []
    for i in range(len(search_string_separated)):
        dictionary.append(search_string_separated[i])
    for i in range(len(corpus_separated)):
        for j in range(len(corpus_separated[i])):
            dictionary.append(corpus_separated[i][j])
    dictionary = list(dict.fromkeys(dictionary))
    # We calculate this by looking at the number of strings a word occurs
→in:
    N = len(corpus_string_wdm) + 1
    dictionary_weights = [dictionary, [None] * len(dictionary)]
    # Getting each word's weight:
    for i in range(len(dictionary)):

```

```

    # The number of documents the word occurs in:
    C = 0
    if dictionary[i] in search_string_separated:
        C += 1
    for j in range(len(corpus_separated)):
        if dictionary[i] in corpus_separated[j]:
            C += 1

    dictionary_weights[1][i] = math.log(N / C)
    # Creating output titles (the first 50 characters of the strings only)
    output_titles = []
    for i in range(len(corpus_string_wdm)):
        if len(corpus_string_wdm[i]) > 50:
            output_titles.append((str(corpus_titles_wdm[i]) + ": " +
→str(corpus_string_wdm[i][:50] + "...")))
        else:
            output_titles.append(str(corpus_titles_wdm[i]) + ": " +
→corpus_string_wdm[i])
    if len(search_string) > 50:
        search_string_title = search_string[:50] + "..."
    else:
        search_string_title = search_string
    # Next, we need to create vectors for each string based on the
→dictionary's weights.
    # We will have an output matrix, with the strings in one dimension
→and the word values in another.
    matrix = [[0 for i in range(len(corpus_string_wdm)+1)] for j in
→range(len(dictionary)+1)]
    matrix[0][0] = search_string_title
    for i in range(len(corpus_string_wdm)):
        matrix[0][i+1] = output_titles[i]
    # Filling the matrix with the word weight values:
    for i in range(len(dictionary)):
        matrix[i+1][0] = (search_string_separated.
→count(dictionary_weights[0][i]) *
            dictionary_weights[1][i])

```

```

    for i in range(1,len(corpus_string_wdm)+1):
        for j in range(len(dictionary)):
            matrix[j+1][i] = (corpus_separated[i-1].
→count(dictionary_weights[0][j]) *
                                dictionary_weights[1][j])

    # This matrix is used in the other functions later on to calculate
→the distances between strings

    return matrix

```

```

[39]: # Calculatin distances from dot products
def dot_product_matrix(corpus, search_string):
    # Creating two lists of corpus: titles and strings:
    corpus_titles = [x[0] for x in corpus]
    corpus_string = [x[1] for x in corpus]
    # Loading the word_distance_matrix function:
    word_distances = word_distance_matrix(corpus, search_string)
    # Calculating the output distances:
    dot_product_distances = []
    for i in range(len(corpus_string)):
        dot_product_distances.append(np.dot([j[0] for j in
→word_distances][1:],
                                [j[i+1] for j in
→word_distances][1:]) /
                                len(corpus_string[i]) * 100)

    # We calculate it twice, once for the rankings and once for the
→output:
    final_ranking_distances = []
    for i in range(len(corpus_string)):
        final_ranking_distances.append(np.dot([j[0] for j in
→word_distances][1:],
                                [j[i+1] for j in
→word_distances][1:]) /
                                len(corpus_string[i]) * 100)

    # We now have our dot product results. Higher values mean more
→similarity.

```

```

final_ranking = ([word_distances[0][1:]] +
                 [final_ranking_distances])
# Determining the ranks based on the distances
total_dist = sum(dot_product_distances)
place = 1
place_vector = [None] * len(corpus_string)
while max(dot_product_distances) > 0:
    for i in range(len(corpus_string)):
        if dot_product_distances[i] == max(dot_product_distances):
            place_vector[i] = place
            for j in range(len(corpus_string)):
                if dot_product_distances[i] ==
→dot_product_distances[j] and i != j:
                    place_vector[j] = place
                    dot_product_distances[j] -= total_dist
                    if max(dot_product_distances) < 0:
                        break
            dot_product_distances[i] -= total_dist
            if max(dot_product_distances) < 0:
                break
            place += 1
final_ranking.append(place_vector)
final_ranking_ordered = []
k = 1
while k <= place:
    for i in range(len(place_vector)):
        if k == place_vector[i]:
            final_ranking_ordered.append([str(final_ranking[0][i])] +
                                         [str(final_ranking[1][i])] +
                                         [str(final_ranking[2][i])])

    k += 1
final_output = []
for i in range(len(final_ranking_ordered)):
    final_output.append("Rank #" + final_ranking_ordered[i][2] + " ("
→+

```

```

        str(final_ranking_ordered[i][1])[:7] +
        " distance): " +
    →str(final_ranking_ordered[i][0]))
    return final_output

```

```
[40]: dot_product_matrix(input_corpus, input_search)
```

```

[40]: ['Rank #1 (15.9919 distance): String 5: it was supposed to be a dream
    →vacation',
      'Rank #2 (3.64422 distance): String 7: it had been what they had been
    →looking
forward to ...',
      'Rank #3 (3.32456 distance): String 6: they had planned it over a year in
advance so that...',
      'Rank #4 (3.19901 distance): String 8: it had been the light at the end
    →of both
their tun...',
      'Rank #5 (2.99123 distance): String 3: the day appeared to be as normal
    →as any
other but ...',
      'Rank #6 (2.03231 distance): String 1: the alarm went off at exactly 600
    →am as
it had eve...',
      'Rank #7 (0.69098 distance): String 4: in fact it was going to change at
exactly 723 am',
      'Rank #8 (0.51566 distance): String 2: barbara began her morning and was
    →ready
to eat bre...']

```

```

[41]: # We will use the following package for Euclidean distances:
from scipy.spatial import distance

```

```

[42]: def Euclidean_distance(corpus, search_string):
    # Creating two lists of corpus: titles and strings:
    corpus_titles = [x[0] for x in corpus]
    corpus_string = [x[1] for x in corpus]

```

```

# Loading the word_distance_matrix function:
word_distances = word_distance_matrix(corpus, search_string)
dot_product_distances = []
for i in range(len(corpus)):
    dot_product_distances.append(distance.euclidean((
        [j[0] for j in
→word_distances[1:]],
        [j[i+1] for j in word_distances[1:]]))

# We calculate it twice, once for the rankings and once for the
→output:
final_ranking_distances = []
for i in range(len(corpus)):
    final_ranking_distances.append(distance.euclidean((
        [j[0] for j in
→word_distances[1:]],
        [j[i+1] for j in word_distances[1:]]))

# We now have our dot product results. Higher values mean more
→similarity.
final_ranking = ([word_distances[0][1:]] +
                 [final_ranking_distances])

# Determining the ranks based on the distances
total_dist = 10000
place = 1
place_vector = [None] * len(corpus)
while min(dot_product_distances) < total_dist:
    for i in range(len(corpus)):
        if dot_product_distances[i] == min(dot_product_distances):
            place_vector[i] = place
            for j in range(len(corpus)):
                if dot_product_distances[i] ==
→dot_product_distances[j] and i != j:
                    place_vector[j] = place
                    dot_product_distances[j] += total_dist
                    if min(dot_product_distances) > total_dist:
                        break
            dot_product_distances[i] += total_dist
            if min(dot_product_distances) > total_dist:

```

```

        break
    place += 1
    final_ranking.append(place_vector)
    final_ranking_ordered = []
    k = 1
    while k <= place:
        for i in range(len(place_vector)):
            if k == place_vector[i]:
                final_ranking_ordered.append([str(final_ranking[0][i])] +
                                              [str(final_ranking[1][i])] +
                                              [str(final_ranking[2][i])])

        k += 1
    final_output = []
    for i in range(len(final_ranking_ordered)):
        final_output.append("Rank #" + final_ranking_ordered[i][2] + " (" +
→+
                                str(final_ranking_ordered[i][1])[7:] +
                                " distance): " +
→str(final_ranking_ordered[i][0]))
    return final_output

```

Testing the function:

```
[43]: Euclidean_distance(input_corpus, input_search)
```

```

[43]: ['Rank #1 (7.09673 distance): String 5: it was supposed to be a dream
→vacation',
      'Rank #2 (8.58745 distance): String 4: in fact it was going to change at
exactly 723 am',
      'Rank #3 (8.98715 distance): String 8: it had been the light at the end
→of both
their tun... ',
      'Rank #4 (9.70823 distance): String 2: barbara began her morning and was
→ready
to eat bre... ',

```

```

'Rank #5 (9.74688 distance): String 3: the day appeared to be as normal_
→as any
other but ...',
'Rank #6 (9.92059 distance): String 1: the alarm went off at exactly 600_
→am as
it had eve...',
'Rank #7 (9.96508 distance): String 7: it had been what they had been_
→looking
forward to ...',
'Rank #8 (10.0596 distance): String 6: they had planned it over a year in
advance so that...']

```

[44]: *# Loading the strings:*

```

with open('C:/CBS/Foundations/final/poems/Poe_Alone.txt', 'r') as file:
    Poe_Alone = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/poems/Poe_Annabel Lee.txt', 'r') as_
→file:
    Poe_Annabel_Lee = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/poems/Poe_The Raven.txt', 'r') as_
→file:
    Poe_The_Raven = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/poems/Shakespeare_The Phoenix and the_
→Turtle.txt', 'r') as file:
    Shakespeare_The_Phoenix_and_the_Turtle = file.read().replace('\n', '_
→')
with open('C:/CBS/Foundations/final/poems/Whitman_I Hear America Singing.
→txt', 'r') as file:
    Whitman_I_Hear_America_Singing = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/poems/Whitman_I Sing The Body_
→Electric.txt', 'r') as file:
    Whitman_I_Sing_The_Body_Electric = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/poems/Whitman_When Lilacs Last in the_
→Dooryard Bloomd.txt', 'r') as file:
    Whitman_When_Lilacs_Last_in_the_Dooryard_Bloomd = file.read().
→replace('\n', ' ')

```

```

with open('C:/CBS/Foundations/final/poems/Wilde_Requiescat.txt', 'r') as file:
    Wilde_Requiescat = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/poems/Wilde_The Burden Of Itys.txt', 'r') as file:
    Wilde_The_Burden_Of_Itys = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/poems/Wilde_The Garden of Eros.txt', 'r') as file:
    Wilde_The_Garden_of_Eros = file.read().replace('\n', ' ')

```

```

[45]: poem_corpus = ["Poe_Alone.txt", Poe_Alone],
                    ["Poe_Annabel Lee.txt", Poe_Annabel_Lee],
                    ["Poe_The Raven.txt", Poe_The_Raven],
                    ["Whitman_I Hear America Singing.txt",
                    Whitman_I_Hear_America_Singing],
                    ["Whitman_I Sing The Body Electric.txt",
                    Whitman_I_Sing_The_Body_Electric],
                    ["Whitman_When Lilacs Last in the Dooryard Bloomd.txt",
                    Whitman_When_Lilacs_Last_in_the_Dooryard_Bloomd],
                    ["Wilde_Requiescat.txt", Wilde_Requiescat],
                    ["Wilde_The Burden Of Itys.txt", Wilde_The_Burden_Of_Itys],
                    ["Wilde_The Garden of Eros.txt", Wilde_The_Garden_of_Eros]]
poem_search = Shakespeare_The_Phoenix_and_the_Turtle

```

```

[46]: # The Damerau-Levenshtein distances:
Dam_Lev_ranking(poem_corpus, poem_search)
# (Takes a while to load)

```

```

[46]: ['Rank #1 (1449 distance): Poe_Annabel Lee.txt: It was many and many a
    year ago,
    In a kingdom by t...',
    'Rank #2 (1544 distance): Whitman_I Hear America Singing.txt: I hear
    America
    singing, the varied carols I hear; ...',
    'Rank #3 (1605 distance): Poe_Alone.txt: From childhoodâ€™s hour I have
    not

```

```

been As others ...',
  'Rank #4 (1712 distance): Wilde_Requiescat.txt: Tread lightly, she is
    ↪near
Under the snow, Speak g...',
  'Rank #5 (4926 distance): Poe_The Raven.txt: Once upon a midnight
    ↪dreary, while
I pondered, wea...',
  'Rank #6 (10438 distance): Whitman_When Lilacs Last in the Dooryard
    ↪Bloomd.txt:
1 When lilacs last in the dooryard bloomâ€™d, And ...',
  'Rank #7 (10663 distance): Wilde_The Garden of Eros.txt: It is full
    ↪summer now,
the heart of June; Not yet ...',
  'Rank #8 (10836 distance): Whitman_I Sing The Body Electric.txt: I sing
    ↪the
Body electric; The armies of those I lo...',
  'Rank #9 (13978 distance): Wilde_The Burden Of Itys.txt: This English
    ↪Thames is
holier far than Rome, Those...']

```

```

[47]: # To see the issue with the Damerau-Levenshtein distance in such long
    ↪documents,
# let's see how the distances correlate with the string lengths
distances = [1605, 1449, 4926, 1544, 10836, 10438, 1712, 13978, 10663]
string_lengths = []
poem_corpus_strings = [x[1] for x in poem_corpus]
for i in range(len(distances)):
    string_lengths.append(len(poem_corpus_strings[i]))
print(np.corrcoef(distances, string_lengths))

```

```

[[1.          0.99581866]
 [0.99581866 1.          ]]

```

```

[48]: dot_product_matrix(poem_corpus, poem_search)

```

```
[48]: ['Rank #1 (3.22161 distance): Poe_Annabel Lee.txt: it was many and many a
→year
ago in a kingdom by th...',
'Rank #2 (2.50333 distance): Wilde_Requiescat.txt: tread lightly she is
→near
under the snow speak gen...',
'Rank #3 (1.69688 distance): Wilde_The Garden of Eros.txt: it is full
→summer
now the heart of june not yet th...',
'Rank #4 (1.64635 distance): Poe_Alone.txt: from childhoods hour i have
→not
been as others wer...',
'Rank #5 (1.44305 distance): Whitman_I Sing The Body Electric.txt: i
→sing the
body electric the armies of those i lov...',
'Rank #6 (1.42301 distance): Wilde_The Burden Of Itys.txt: this english
→thames
is holier far than rome those ...',
'Rank #7 (1.34545 distance): Whitman_When Lilacs Last in the Dooryard
Bloomd.txt: 1 when lilacs last in the dooryard bloomd and the ...',
'Rank #8 (1.28547 distance): Poe_The Raven.txt: once upon a midnight
→dreary
while i pondered weak ...',
'Rank #9 (0.80509 distance): Whitman_I Hear America Singing.txt: i hear
→america
singing the varied carols i hear th...']
```

```
[49]: # Now let's see if these distances correlate with string lengths:
dot_product_distances = [1.64635, 3.22161, 1.28547, 0.80509, 1.44305, 1.
→34545, 2.50333, 1.42301, 1.69688]
print(np.corrcoef(dot_product_distances, string_lengths))
# They are not correlated, because we already accounted for string
→lengths in the function.
```

```
[[ 1.          -0.36423634]
 [-0.36423634  1.          ]]
```

```
[50]: Euclidean_distance(poem_corpus, poem_search)
```

```
[50]: ['Rank #1 (32.4426 distance): Wilde_Requiescat.txt: tread lightly she is
      ↪near
      under the snow speak gen...',
      'Rank #2 (33.4728 distance): Poe_Alone.txt: from childhoods hour i have
      ↪not
      been as others wer...',
      'Rank #3 (42.5969 distance): Whitman_I Hear America Singing.txt: i hear
      ↪america
      singing the varied carols i hear th...',
      'Rank #4 (43.4422 distance): Poe_Annabel Lee.txt: it was many and many a
      ↪year
      ago in a kingdom by th...',
      'Rank #5 (88.7217 distance): Wilde_The Garden of Eros.txt: it is full
      ↪summer
      now the heart of june not yet th...',
      'Rank #6 (89.1769 distance): Poe_The Raven.txt: once upon a midnight
      ↪dreary
      while i pondered weak ...',
      'Rank #7 (101.469 distance): Whitman_When Lilacs Last in the Dooryard
      Bloomd.txt: 1 when lilacs last in the dooryard bloomd and the ...',
      'Rank #8 (103.808 distance): Whitman_I Sing The Body Electric.txt: i
      ↪sing the
      body electric the armies of those i lov...',
      'Rank #9 (107.160 distance): Wilde_The Burden Of Itys.txt: this english
      ↪thames
      is holier far than rome those ...']
```

```
[51]: # Once again, let's see if these results depend on the strings' lengths:
      Euclidean_distances = [33.4728, 43.4422, 88.1769, 42.5969, 103.808, 101.
      ↪469, 32.4426, 107.160, 88.7217]
      print(np.corrcoef(Euclidean_distances, string_lengths))
      # There is a large correlation between the values.
```

```
[[1.          0.95721818]
```

```
[0.95721818 1.      ]]
```

```
[52]: # Loading the strings:
with open('C:/CBS/Foundations/final/same_length/string1.txt', 'r') as f:
    file:
        string1 = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/same_length/string2.txt', 'r') as f:
    file:
        string2 = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/same_length/string3.txt', 'r') as f:
    file:
        string3 = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/same_length/string4.txt', 'r') as f:
    file:
        string4 = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/same_length/string5.txt', 'r') as f:
    file:
        string5 = file.read().replace('\n', ' ')
with open('C:/CBS/Foundations/final/same_length/same_length_search_string.
    txt', 'r') as f:
    same_length_search_string = file.read().replace('\n', ' ')
```

```
[53]: same_length_corpus = [{"string1.txt", string1},
                            ["string2.txt", string2],
                            ["string3.txt", string3],
                            ["string4.txt", string4],
                            ["string5.txt", string5]]
```

```
[54]: # The Damerau-Levenshtein distances:
Dam_Lev_ranking(same_length_corpus, same_length_search_string)
```

```
[54]: ["Rank #1 (270 distance): string5.txt: There was a time when this
    wouldn't have
bothered ...",
    'Rank #2 (285 distance): string1.txt: He slowly poured the drink over a
    large
```

```

chunk of i...',
    'Rank #3 (287 distance): string4.txt: The shades were closed keeping the
    ↳room
dark. Pete...',
    'Rank #4 (289 distance): string2.txt: She never liked cleaning the sink.
    ↳It was
beyond h...',
    "Rank #5 (300 distance): string3.txt: You're going to make a choice
    ↳today that
will have..."

```

```

[55]: # Once again, let's see the correlation between distances and lengths:
same_D_L_distances = [285, 289, 300, 287, 270]
same_string_lengths = []
same_length_corpus_strings = [x[1] for x in same_length_corpus]
for i in range(len(same_D_L_distances)):
    same_string_lengths.append(len(same_length_corpus_strings[i]))
print(np.corrcoef(same_D_L_distances, same_string_lengths))
# Even at small differences in lengths, these small differences are
    ↳significant

```

```

[[1.          0.88362546]
 [0.88362546 1.          ]]

```

```

[56]: # The dot-product ranks:
dot_product_matrix(same_length_corpus, same_length_search_string)

```

```

[56]: ['Rank #1 (5.47733 distance): string1.txt: he slowly poured the drink
    ↳over a
large chunk of i...',
    'Rank #2 (3.88784 distance): string4.txt: the shades were closed keeping
    ↳the
room dark peter...',
    'Rank #3 (2.01467 distance): string5.txt: there was a time when this
    ↳wouldnt
have bothered h...',

```

```
'Rank #4 (1.30156 distance): string2.txt: she never liked cleaning the
↳sink it
was beyond he...',
'Rank #5 (0.90745 distance): string3.txt: youre going to make a choice
↳today
that will have ...']
```

```
[57]: # Their correlations:
same_dot_product_distances = [5.47733, 1.30156, 0.90745, 3.88784, 2.01467]
print(np.corrcoef(same_dot_product_distances, same_string_lengths))
# There is no correlation between the dot product distances and the
↳string lengths
```

```
[[ 1.          -0.07805612]
 [-0.07805612  1.          ]]
```

```
[58]: # The Euclidean distances:
Euclidean_distance(same_length_corpus, same_length_search_string)
```

```
[58]: ['Rank #1 (16.4607 distance): string1.txt: he slowly poured the drink
↳over a
large chunk of i...',
'Rank #2 (16.7552 distance): string4.txt: the shades were closed keeping
↳the
room dark peter...',
'Rank #3 (16.9688 distance): string2.txt: she never liked cleaning the
↳sink it
was beyond he...',
'Rank #4 (17.0178 distance): string5.txt: there was a time when this
↳wouldnt
have bothered h...',
'Rank #5 (22.7794 distance): string3.txt: youre going to make a choice
↳today
that will have ...']
```

```
[59]: # Their correlations:
same_Euclidean_distances = [16.4607, 16.9688, 22.7794, 16.7552, 17.0178]
print(np.corrcoef(same_Euclidean_distances, same_string_lengths))
# Now there is only a weak correlation between Euclidean distances and
→string lengths
# It is also interesting to see, that now the results are almost the same
→for the
# dot product and Euclidean ranks!
```

```
[[1.          0.56050713]
 [0.56050713  1.          ]]
```

```
[60]: test_corpus = [{"Test string", 55}]
test_search = 115
Dam_Lev_ranking(test_corpus, test_search)
```

User inputs should be string. We converted the values that aren't.
Search string should be string. We converted it to string.

```
[60]: ['Rank #1 (2 distance): Test string: 55']
```