Executive Summary: Linear Algebra Applications in Data Science by Jørgen Leiros

Jørgen Maurstad Leiros, as part of his MSc in Business Administration and Data Science, demonstrated **practical applications of linear algebra in programming and data science**, particularly in **matrix operations, error correction, and text similarity analysis**. His work showcases proficiency in **numerical computing, machine learning foundations, and computational efficiency**.

Key Focus Areas:

1. **Development of a Custom NumPy-like Library ("Sub-NumPy")** –
    - Implemented fundamental **matrix operations (reshape, append, shape extraction, addition, subtraction, and dot product calculations)** without relying on external libraries.
    - Designed a **Gaussian elimination solver for linear equation systems**, including **error detection, row reduction, and matrix transformations** for efficient computation.
2. **Hamming Code for Error Detection and Correction** –
    - Built an implementation of **Hamming(7,4) coding**, a key method for **error correction in data transmission**.
    - Developed a **parity-check algorithm** to **detect and correct single-bit errors**, showcasing expertise in **binary encoding, syndrome calculations, and matrix multiplications** for digital communication reliability.
3. **Text Document Similarity Analysis Using Linear Algebra** –
    - Created an NLP-based **text similarity tool** leveraging **TF-IDF (Term Frequency-Inverse Document Frequency) vectorization**.
    - Applied **cosine similarity and Euclidean distance metrics** to **rank documents based on semantic closeness**, demonstrating advanced use of **linear algebra in information retrieval and text analytics**.
4. **Computational Optimization and Performance Considerations** –
    - Explored **computational complexity and efficiency** in matrix-based algorithms.
    - Addressed trade-offs in **algorithm design, error propagation, and performance scaling** for large datasets.

Key Takeaways:

- **Linear algebra is fundamental to numerical computing, error detection, and NLP applications.**
- **Matrix transformations and vector operations** are essential for **solving linear equations and enhancing data analysis workflows**.
- **Jørgen's work integrates mathematics, programming, and data science**, showcasing a **strong foundation in computational problem-solving**.

Through this research, Jørgen has demonstrated expertise in **mathematical modeling, algorithm development, and applied data science**, contributing valuable insights into **AI, data analytics, and error correction techniques**.

Question 1 - Sub-NumPy

Prerequisite - check_input

The 'check_input' function in 'SNumPy' is essential for making sure inputs are correct before any calculations. It checks if it's given a matrix and makes sure all rows are the same length, which is a must-have for matrix operations. If it's just a single list of numbers, it treats it as a vector. It also changes any numbers written as text into numerical values, and if it finds something that isn't a number where it should be, it raises an error. This function is key because it tidies up the data and ensures everything's ready for smooth calculations.

1: Ones

The *ones* function in SNumPy does not utilize the '*check_input*' function, as this is not needed to execute the function correctly. The *ones* function starts off by checking if the length is an integer, and additionally checks if it is positive. If the length is a valid positive integer, the function creates and returns an array with length number of ones. If the length is not an integer, the function raises a TypeError or a ValueError, depending on the nature of the input error.

2: Zeros

The *zeros* function in SNumPy does not utilize the '*check_input*' function either, as this is not needed to execute the function correctly. Similarly to the *ones* function, the *zeros* function ensures that the input is a valid positive integer and uses it to determine the size of the array filled with zeros that it then returns.

3: Reshape

The *reshape* function takes an array and converts it into the dimension specified by a tuple. This tuple includes two integers specifying the number of rows and columns that the new matrix should consist of. By initially running the '*check_input*' function, it

ensures robust data handling and transformation. If the dimensions match, the function uses nested loops and constructs the reshaped matrix. The matrix is further populated by the result of each of the completed inner loops (*new_row*, representing a single row of the reshaped matrix), and the function returns the reshaped matrix. If the matrix input is invalid, the ValueError or TypeError raises the proper exceptions in a user-friendly manner.

4: Shape

The *shape* function starts off by taking a single argument, an array, which is expected to be either a vector or a matrix. It also facilitates the check of empty arrays, returning (0, 0) if the array has no elements. Furthermore, it utilizes the '*check_input*' function to validate the input. After the validation is complete, it checks the input to see if it's a vector and, if this is the case, returns a tuple with the length of the array, and a corresponding 1 as the column number. If the input array is not captured by the 'vector check', it is treated as a matrix. Finally, if the array passes all the checks without running into errors (handled with user-friendly error messages), the function returns a tuple (*len(array), col_num*), representing the number of rows and columns in the matrix (the dimensions of the matrix).

5: Append

The *append* function takes two parameters, array1 and array2, which are the two arrays that are going to be combined. Firstly, the function checks if the first elements of both arrays are not lists. If this condition is true, it implies that they are both vectors, and the function combines the two arrays (*array1 + array2*). However, if the arrays are lists, the function treats them as matrices. Considering the arrays are matrices, the function checks if they have the same number of rows, and if this is the case, it concatenates the corresponding rows to form a *combined_row*, which is then appended to the new list *result*. After iterating through all the rows in the matrix, it returns the *result*. If the function detects an attempt to combine a vector with a matrix or vice versa, it raises a TypeError, indicating that this attempt is not supported. Further, if the matrices do not have the same number of rows, it raises a ValueError on the basis that matrices must have the same number of rows to be combined.

6: Get

The *get* function takes two parameters: array, which is the vector or matrix to be accessed, and a location, which is a tuple containing the row and column indices. Initially, the function uses the '*check_input*' function, as in the previous tasks, to ensure that the array is in a suitable format for the operations. The vector check in the function is also similar to the one used in previous tasks. If the array is a vector, the function checks if the column index is 1, which is required for a one-dimensional vector. Further, it checks whether the row index is within the range of the vectors' length, and if it is, the *get* function returns the value at that index (*array[row - 1]*), adjusting for zero-based indexing. The *else* statement is executed if the array is a matrix, and it starts by checking whether the row index is within the range of the matrix's rows (*len(array)*). If the row is valid, it further checks if the column index is within the range of the columns in that row (*len(array[0])*). If both indices are valid, the function returns the value at the specified location (*array[row - 1][column - 1]*). During the function, the appropriate errors are raised if the indices are out of range, or if an attempt is made to access a vector with an invalid column index. In conclusion, the *get* function retrieves a specific value from a vector or a matrix based on the provided row and column indices.

7 & 8: Add & Subtract

The *add* function seeks to add two vectors or matrices together. It takes two arguments, array1 and array2. After making sure the inputs are valid (by using the '*check_input*' function), the *add* function utilizes the *shape* function to ensure that the arrays have the same dimensions, which is crucial for the addition to be possible. This is because element-wise addition requires both arrays to have the same dimensions. A ValueError is raised if the dimensions don't match. Furthermore, the function checks if the array is a vector, and if it is, it runs a for loop to iterate over the elements of the vectors. In each iteration, the corresponding elements from array1 and array2 are added together and appended to an empty list called *vector_added* where the result is stored and eventually returned. If the input is caught by the *else* statement in the function, it implies that the arrays are matrices. In the case of matrix addition, a nested for loop is used. The outer loop iterates over the rows, while the inner loop iterates over the columns. For each element in the matrices, the sum of the corresponding elements (*array1[i][j] +*

array2[i][j]) is calculated and appended to an empty list called *row_sum*, which represents a row in the resulting matrix. This list is further appended to an empty list called *matrix_added*. After the iterations are completed, *matrix_added* is returned, containing the element-wise sum of the two matrices. The same procedure is done in the *subtract* function, except that this function subtracts the vector/matrix elements instead of adding them, as the *add* function does.

9: Dot Product

The *dotproduct* function seeks to compute the dot product between two arrays (vectors or/and matrices) and return an appropriate value. The function takes two arguments, array1 and array2. This function also utilizes the '*check_input*' function to ensure that the arrays are provided in a suitable format for the calculations. Additionally, the function determines the dimensions of the arrays by using the *shape* function. These dimensions are stored in *dims1* and *dims2*, which are used to determine whether the dot product can be performed. After these initial operations, the function checks whether array1 is a one-row matrix (indicating a vector), and whether array2 is not a matrix. If array1 is a vector, it further checks if the length of array1 and array2 match. For the vector dot product, the function iteratively multiplies corresponding elements of array1[0] (which is the first row, since it's treated as a one-row matrix) and array2, summing these products to calculate the dot product. If the lengths don't match, it raises a ValueError, indicating the operation is not defined for vectors of different lengths. For the dot product of matrices; if the first dimension check doesn't pass, the function checks if the number of columns in array1 (*dims1[1]*) matches the number of rows in array2 (*dims2[0]*), which is a requirement for matrix multiplication. The function then iterates over each row of array1 and each column of array2, multiplying corresponding elements and summing these products to calculate the dot product for each element of the result matrix. The results are stored in *result_row*, which are further appended to *result*, providing the final dot product matrix. If the dot product cannot be defined due to incompatible dimensions (either in the vector or matrix case), the function raises a ValueError with an appropriate message.

10 (Optional): Solver for a system of linear equations

This piece of code solves a system of linear equations using Gaussian elimination and row reduction rules. In this part of the project, we decided to use frequent printing statements during the code on the basis that the code is quite comprehensive. In doing this, we can follow each step of the gaussian elimination all the way until the final solution. This is also a good feature to have for troubleshooting in case of a loop running forever.

solver

The central method of the class is *solver*, which orchestrates the process of solving a linear system represented by matrix *A* and vector *b*. Initially, it validates *A* and *b* using the *check_input* method, ensuring the inputs are in the correct numeric format and have compatible dimensions. This step is crucial to avoid errors during the calculation process. Once validated, the method combines *A* and *b* into an augmented matrix, a standard approach in Gaussian elimination that allows for efficient manipulation of the system.

is_inconsistent

The *is_inconsistent* method checks if a given matrix represents an inconsistent linear system. It examines each row of the matrix to find any scenario where the coefficients of the variables are all zero, but the constant term is non-zero, which signifies an impossible equation.

is_reduced

The *is_reduced* method checks whether the matrix has reached this desired form. It looks for rows that are not properly reduced and also checks for inconsistencies. An inconsistency, detected when a row is entirely zeros except for the last element, indicates that the system has no solution. This check is crucial, as it prevents the algorithm from running indefinitely on unsolvable systems.

sort_gauss

However, Gaussian elimination alone does not guarantee that the matrix is in a form easy to extract solutions from. This is where *sort_gauss* comes into play. It repeatedly

adjusts the matrix, ensuring each row is properly normalized and all necessary elements are zeroed out, effectively bringing the matrix closer to reduced row echelon form.

extract_result

Once the matrix is in reduced row echelon form, *extract_result* extracts the solution. This method deals with different scenarios: it identifies whether the system has a unique solution, no solution, or infinitely many solutions. Handling these different cases is vital, as linear systems can vary widely in their nature.

transpose

The *transpose* method is designed to transpose a given matrix, which involves swapping its rows and columns (so the element at row *i*, column *j* in the original matrix moves to row *j*, column *i* in the transposed matrix). With a combination of Python's *zip* function and list comprehension, it effectively groups elements from each row of the input matrix based on their position, thus creating a new matrix where rows are transformed into columns and vice versa.

apply_gaussian_elimination

The augmented matrix undergoes Gaussian elimination through *apply_gaussian_elimination*, which systematically transforms the matrix into an upper triangular form. This process involves two key steps: forward elimination, which creates zeros below the main diagonal, and back substitution, aimed at generating zeros above the main diagonal. The result is a matrix that is much simpler to work with when extracting solutions.

Question 2 - Hamming's Code

When transmitting data, sequences of bits are sent over a communication channel, such as a wire, fiber optic cable or a wireless signal. The Hamming code operates on four bits at a time. As the bits travel through the channel, they encounter noise, which can distort the information and flip certain bits from 0 to 1 or the other way (Klein, n.d., p. 251). To deal with this, systems use error checking and correction methods.

In this context, we are utilizing the Hamming(7,4) code. The primary objective of Hamming codes is to generate a series of overlapping parity bits (Wikipedia, n.d.). This overlap allows for both the detection and correction of single-bit errors, whether in the data bits or the parity bits. The Hamming(7,4) code specifically encodes a group of four data bits into a seven-bit sequence by adding three parity bits, also known as check bits. These check bits are strategically appended to each quartet of data bits in the message, thereby facilitating error correction. The data bits in this scenario are labeled as D_1, D_2, D_3, and D_4. Concurrently, the three parity bits, P_1, P_2, and P_3, are computed based on the data bits' values and are subsequently placed in predetermined positions within the code.

- P_1 is calculated from the bits that have their 1st bit in their binary representation set (for example, bits 1,3,5,7)
- P_2 is calculated from the bits that have their 2nd bit in their binary representation set (for example, bits 2, 3, 6, 7).
- P_3 is calculated from the bits that have their 3rd bit in their binary representation set (for example, bits 4, 5, 6, 7).

These groups overlap one another. This ensures that if a single-bit error occurs, the combination of which parity bits are incorrect will point to the exact location of the error. This allows for not only detecting one-bit errors, but also for correcting errors, ensuring reliable data transmission.

Program Design

The program consists of three classes, each with a specific role in demonstrating the Hamming(7,4) error detection and correction process. The *HammingMatrices* class is responsible for storing the essential matrices used in the Hamming(7,4) code process: the generator matrix *G*; the parity-check matrix *H*, and the decoder matrix *R.* Moving on, the *HammingCode* class extends the *HammingMatrices* and includes methods for encoding, decoding, error detection, and error correction. Lastly, the *HammingProcessing* extends *HammingCode* and is designed to process sequences of binary codes using the Hamming (7,4) encoding and decoding, along with

demonstrating error introduction, detection, and correction with single- and two-bit errors.

The design choices are based on the object-oriented principles. The three classes encapsulate the data and methods related to different aspects of the Hamming(7,4) code. Abstraction is achieved as users engage with high-level methods like *encoder* in *HammingCode* or *process_sequences* in *HammingProcessing* without needing to understand the complex matrix operations underneath. The design also allows for inheritance, and it allows for future expansion if needed.

Encoding

Consider a scenario where a bit sequence from the Hamming class is transmitted through a noisy communication channel. At the beginning of the Hamming class, we define the generator matrix *G* for a linear code. *G* consists of columns that serve as generators for the codewords transmitted through the channel. From the *encoder* method, we obtain the codeword *x*, where we compute the product of *G* and a four-bit vector *p*, with all entries calculated modulo 2, which is equivalent to XOR operation in binary (Wikipedia, n.d.). This process effectively encodes the original four-bit data into a 7-bit format that can be reliably transmitted, and which allows for error-detection and correction. As in the *Hamming class*, the vector *p* '1011' is encoded, and the codeword *x* '0110011' is obtained. This is illustrated below:

$$\mathbf{x} = \mathbf{G}^{\mathbf{T}}\mathbf{p} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 2 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Figure 1. Computing the codeword *x* (Wikipedia, n.d.).

Parity Check

If the received codeword, denoted as *r*, is identical to the transmitted codeword, *x*, it indicates that there have been no errors during the data transmission (Wikipedia, n.d.). To determine this, we must calculate the syndrome vector, denoted as *z*, which serves as

an indicator of whether an error has occurred and, if so, identifies the specific bit affected by the error.

In the program, the *parity_check* method processes a given codeword by calculating the syndrome vector. This is achieved by multiplying the codeword with the parity-check matrix, denoted as *H*, which is predefined in the class (Wikipedia, n.d.). The resulting syndrome vector is a binary representation of the error's position in the codeword. The syndrome is calculated as a binary vector with the least significant bit (LSB) on the right. However, when we convert this binary vector to a string for interpretation, we reverse it to match the conventional binary numbering system, where the LSB is at the end of the string. The *int* function converts the binary string to its integer equivalent, which represents the position of the error. If the syndrome vector is a null vector (all zeros), it indicates no errors, but if there is an error, the position is returned.

Following the encoding process of the bit sequence '1011', the codeword received is '0110011'. To ensure the integrity of the data, the parity check is performed. The process is illustrated below:

$$\mathbf{z} = \mathbf{Hr} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Figure 2. Computing the syndrome vector *z* (Wikipedia, n.d.).

Decoder

Once the received codeword *r* has been determined by the *parity_check* method (and corrected if necessary), the received data has to be decoded back to the original four bits, p_r. This is done in the *decoder* method. The process involves multiplying *r* by the decoder matrix, denoted as *R* (Wikipedia, n.d.). The multiplication isolates the original data bits from the codeword and results in a four digit bit sequence representing the original data that was encoded.

In the previous example, after subjecting the codeword '0110011' to a parity check, it was determined that the received data was error-free. Consequently, the codeword needs to be decoded back into the original four bits, p_r (received value), as demonstrated below:

$$\mathbf{p_r} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Figure 3. Decoding the received codeword *r* (Wikipedia, n.d.).

Error Detection and Correction

If there is an error, we can write $r = x + e_i$ under modulo 2, where e_i is the i_{th} unit vector, that is, a zero vector with a 1 in the i_{th}, counting from 1. *x* is the transmitted codeword (Wikipedia, n.d.). Illustrated below is an example where an error is introduced at position 5:

$$\mathbf{r} = \mathbf{x} + \mathbf{e}_5 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Figure 4. Introducing a bit error in bit position 5 (Wikipedia, n.d.).

Now the received codeword *r* differs from the transmitted codeword *x*, thus $r \neq x$ (Wikipedia, n.d.) In the program, the *correct_errors* method is used to check and correct errors in a received codeword. In the method, the *parity_check* method is called, which results in this:

$$\mathbf{z} = \mathbf{Hr} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

Figure 5. Computing the syndrome vector *z* when a bit error has been introduced (Wikipedia, n.d.).

Since the syndrome vector z is non-zero, it indicates the presence of an error. To correct it, the method uses the XOR operator ('^1') to flip the affected bit. In this case, the syndrome '101' is a result of the matrix-vector product between the *H* matrix and the error vector *e*, indicating that the fifth bit was corrupted (Klein, n.d., p. 212). Once the error is corrected, the decoder method is invoked.

Two-bit errors

The Hamming(7,4) code is designed to detect and correct single-bit errors. However, when two errors occur, the code can misidentify the position of the error. In the part of the program where two bit errors are introduced at certain positions, the algorithm may suggest the wrong position of the error, and try to correct it. When two errors occur, for example at position 2 and 5, the resulting syndrome vector *z* is essentially the sum (modulo 2) of the syndromes that would have been produced by each error independently. This results in a new syndrome vector that does not point to either of the error positions, but some other, for example 3. This limitation makes the Hamming(7,4) efficient for single-bit error correction but not for multi-bit error scenarios.

Question 3 - Text Document Similarity

Text Embedding

The third question of the final project asks for a Python program which takes a corpus of text documents and creates a dictionary of unique words from said corpus. Then a search document is provided and the program computes the similarity between the documents and displays the most similar documents from the corpus in descending order. Text similarity in Python can be measured by converting text files into vectors and computing the distance between vectors. A small distance between two vectors generally means that they are similar, and the similarity score based on this measure is usually in the range of 0 to 1. This score is subjective to the methods used for vectorizing and measuring distance, and choice of methods should be a result of careful consideration, taking into account the domain and use case for the similarity check.

To be able to measure the similarity using Python, the text needs to be presented in a way that algorithms can make sense of them. Text embedding is the act of encoding words into numeric form to make them machine understandable. The objective of this process is to capture as much of the meaning, context, and semantics of words in a corpus as possible. One way to do this is to create a dictionary of unique words and create vectors by checking to see which of the words occur in a text, represented as booleans. Optionally, this can be done by counting the amount of times any of the words occur in a text, which attributes a weight to each of the words for each of the documents.

Term Frequency - Inverse Document Frequency (TF-IDF) is an algorithm that turns text into vectors by assessing the frequency of words across a corpus of texts, rather than just in one document (Karabiber, n. d.). The TF is calculated through the number of occurrences of each word in the dictionary for a document divided by the total number of words in said document. The IDF is calculated as the natural logarithm of the number of documents in the corpus divided by the number of documents in which the term occurs. The TF-IDF vector is then calculated as the product of the two numbers.

The benefit of this text embedding approach compared to the two discussed in the previous paragraph, is that the significance of words that occur often in all texts of the corpus is depreciated. In this way, the algorithm can prevent some stop words from affecting the score as much, which is why we have chosen this vectorization algorithm for our text similarity program.

Distance Metrics

To compute the similarity between two vectors, the question text suggested euclidean distance and dot product as potential metrics. Euclidean distance computes the sum of squared difference for every corresponding element in two vectors and then takes the square root of this sum .

$$d(S_a, S_b) = \sqrt{\sum_{i=1}^{n} (S_a^{(i)} - S_b^{(i)})^2}$$

Figure 6. Formula for the euclidean distance between two vectors (Wang and Dong, 2020).

The larger the distance between two vectors, the lower the similarity score. In some cases this can be a confusing measure as it can range from zero to infinity, but in the case of comparing one document to a corpus, it is as simple as the smaller distance between vectors, the more similar the document.

Cosine similarity computes the cosine of the angle between two vectors, determining if the vectors are pointing in the same direction. This value is derived from the dot product of two vectors divided by the product of their magnitudes. The result of this calculation will be 1 if the vectors are pointing in exactly the same direction.

$$\mathrm{Sim}(S_a, S_b) = \cos\Theta = \frac{\vec{S_a}\cdot\vec{S_b}}{\|S_a\|\cdot\|S_b\|}$$

Figure 7. Formula for the euclidean distance between two vectors (Wang and Dong, 2020)

Each of these metrics have unique qualities, making them well suited for each of their own purposes. The major difference between euclidean distance and cosine similarity is that cosine similarity does not consider the length of the vector or the size of the text. Additionally, euclidean distance is not well suited for sparse matrices, usually created through text embeddings. For this reason, cosine similarity is often the preferred metric for comparing textual data while euclidean distance can be beneficial in tasks like text classification (Baeldung, 2022). We have chosen to rank the similarity of the documents based on the cosine similarity and sort by this ranking to provide the list of documents most similar to the search document.

Program Design

We have decided to use classes for this program as it provides higher reusability, decreases the need for global variables and provides natural modularity which can help improve maintainability. Creating this program without using classes would be enough to answer the question, but if you imagine that such a similarity check would be a part of a larger system, using classes is beneficial due to the aforementioned qualities.

The program starts by importing the necessary libraries before defining the class for the similarity check. The *__init__* method runs as soon as the class is used to instantiate an

object and organizes the methods in the program in a logical order. It loads the corpus and stores the corpus and file names in two different variables before invoking the *word_list* function. The method goes on to append the preprocessed text of the search document to the corpus before vectorizing the texts through the *tf_idf* method, passing it both the corpus and dictionary obtained from the *word_list* function.

The *load_doc* method takes a string in the form of a file path as an argument, reads the document and stores it in the variable *doc_text*. The text is then processed using a *re.sub* function that takes any characters that are not alphanumeric and replaces them with an empty space as well as turning all characters into lower case. The preprocessed text string is then returned and stored in a variable in the *load_corpus* method where the *load_doc* method is first invoked. The *load_corpus* method uses list comprehension to create a list of all the filenames that are actual files, excluding hidden files and only including txt files. The file names are then used to create a list by invoking *doc_load* for all the documents in the corpus directory. Both the list of *file_names* and the corpus of documents are then returned. The last method that is used to prepare the program for text embedding is *load_word_list*. This method takes the previously loaded corpus as an argument, joins the tuples of the corpus list and returns a set containing all the unique words in the corpus.

The next part of the program is dedicated to turning each text file into vectors before calculating similarity. The mathematical calculations have been explained above so we will only comment on the logic of the code here. The *compute_tf_idf* method takes the corpus and dictionary of unique words as arguments and invokes the *computing_tf* and *computing_idf* methods. Based on the returned values, the method computes the product of the Term Frequency (TF) and Inverse Document Frequency (IDF) and the resulting data frame is returned. *Computing_tf* creates variables containing the length of each list and creates an empty pandas dataframe with columns for each word and rows for each document. Using a for loop, the program computes the TF for each word in each document and stores the values in the pandas dataframe before returning the dataframe. In the loop of this method we have added an if statement for new words introduced by the search document that are not part of the dictionary. These words are simply added to the dictionary as they occur and are provided a count of zero as they

don't occur in any of the documents in the corpus. *Computing_idf* creates an empty dictionary and stores the length of the corpus in a variable. Using a for loop, the method iterates through the words in the dictionary for every document in the corpus to see how many documents each word is included in before computing the IDF, adding it to the empty dictionary and returning the complete dictionary to the *compute_tf_idf* method.

Two different methods have been defined to measure the similarity between the documents in the corpus and the search document, the math for which have been explained earlier in this chapter. Both of these methods take two different vectors as arguments, one representing the search document, and one representing one of the documents in the corpus. The methods each carry out every step of the calculations explained earlier and return the similarity score.

Finally, the *analyze_similarity* method stores the index of the search document in a variable. This is used to distinguish its vector from the vectors of the original documents of the corpus. Then list comprehension is used to create a list of similarity scores between the search document vector and every other vector in the *tf_idf_df*. By adding an if statement to both of these function calls, we avoid computing the similarity between the search document and itself. After computing these similarity scores, the results are added into a dataframe, then they are ranked and sorted by cosine similarity score before displaying the results.

The DocumentSimilarityAnalyzer class is instantiated in a *if __name__ == "__main__"* statement containing the path to the corpus directory and the path to the search document. This ensures that the program is executed only if the script is run as the main program. Within said if statement, the objects *analyze_similarity* method is invoked so that the program displays a dataframe with the most similar documents to the search document.

References

Baeldung (2022) Math and Logic: *Euclidean Distance vs Cosine Similarity.* Retrieved 11. 12. 2023 from https://www.baeldung.com/cs/euclidean-distance-vs-cosine-similarity

Fatih Karabiber. (n.d.). TF-IDF — Term Frequency-Inverse Document Frequency. Learndatasci. Retrieved 12. 12. 2023 from https://www.learndatasci.com/glossary/tf-idf-term-frequency-inverse-document-frequency/

Klein, P. N. (n.d.). Coding the matrix: Linear Algebra Through Applications to Computer Science (1st ed.). Brown University.

Wang, J., & Dong, Y. (2020). Measurement of Text Similarity: A Survey. *Information*, *11*(9). https://doi.org/10.3390/info11090421

Wikipedia contributors. (n.d.). Hamming(7,4). In *Wikipedia*. Retrieved December 1st, 2023, from https://en.wikipedia.org/wiki/Hamming(7,4)