MSc in Business Administration and Data Science

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

Date: 12.12.2023

Pages: 15

Characters: 27,881

Contents

1 Question 1

The initial task involves constructing a Python sub-library similar to *Numpy*, specialised in converting arrays into vectors and matrices, as well as performing mathematical operations such as dot products or matrix multiplication. Unlike *Numpy*, this library operates with Python lists instead of creating its own objects like *ndarrays*.

For the subsequent parts of the question, we assume that a vector can be denoted as a list of integers (e.g., $[1, 2, 3, 4, 5]$), and a matrix as lists within lists, where each list symbolises a row of the matrix (e.g., $[[1, 2], [3, 4]]$). It's important to note that a vector will be horizontally represented for dot product operations, resembling a matrix with a single row. Additionally, only integers and floats are permissible as inputs within the array and empty vectors and matrices are not allowed.

1.1 Auxiliary functions

is_valid_array(array)

This function is responsible for ensuring that the submitted array is a valid vector or matrix. The checks include disallowing empty arrays, verifying that the array is a list, ensuring each element is numeric, and for matrices, confirming that every row has the same number of elements. If any of the tests tails, this function raises exceptions with detailed explanations of the errors. Given that this function will be called for every input array in all subsequent functions, there is no necessity to address non-valid arrays in any other manner.

is_vector(array)

This function returns *True* if the array is a vector and *False* if it is a matrix. The check involves examining whether the first entry of the list is another list (indicating a matrix) or a number (indicating a vector).

1.2 Core functions

ones(length)

The *ones* function generates a list of size *length*, setting all its elements to the number 1. Prior to the creation of the list, it verifies that the input *length* is an integer and raises an exception if it is not.

zeros(length)

The *zeros* function has the same functionality as the *ones* function but instead, returns a list of zeros of size "length".

reshape(array, dimensions)

The *reshape* function is designed to transform the structure of an input array into a new shape, which is specified by the *dimensions* parameter. Initially, the function verifies the array's validity through the *is_valid_array* function. Then, it checks if the total number of elements in the current array matches the total number of elements specified by the input dimensions. It does this by comparing the product of the rows and columns of the input array's shape (*shape(array)*) with the product of the rows and columns in the input dimensions. Additionally, it checks if the tuple provided is valid. Once these checks are completed, if the input array is a matrix, it appends its elements to a list shaped like a vector. If the target shape is a vector, the function returns this new array. However, if the target shape is a matrix, the function iterates through the elements of the array and appends them row by row to an array with the correct shape specified by *dimensions.*

shape(array)

The *shape* function returns a tuple representing the dimensions of a given array, which could either be a vector or a matrix. First, the *is_valid_array* function is called to validate that the array provided is properly structured as a matrix or a vector. It uses *is_vector(array)* to check whether it is a matrix or a vector, and if it is a matrix, returns a tuple where the first element is the number of rows and the second is the number of columns. If the array is a vector, the function returns *(1,len(array))* where *1* represents the single row and *len(array)* is the number of elements (columns) in the vector.

append(array1, array2)

The *append* function merges two arrays, which can either be vectors or matrices, resulting in a single vector or matrix. The function utilizes the *is_valid_array* function to ensure the validity of both input arrays. Additionally, it verifies that both provided arrays are of the same type, i.e., both matrices or both vectors. Finally, in the case of both arrays being matrices, it checks that they have the same number of columns. If the validation checks pass, the combined vectors or matrices are returned. Specifically, for vectors, the function appends the second array to the first one, and for matrices, it adds the rows of array 2 as new rows for array 1.

get(array, position)

The *get* function is designed to retrieve an element from the provided array, which can be either a vector or a matrix, at a specified position. It takes an array and the position of the desired element as input. First, it checks if the tuple provided is valid. Then, the function calls the *is_vector* function and checks whether the array is a vector or a matrix. If the array is a vector it retrieves the element at the index specified by the second element of the position tuple. If the array is a matrix, it retrieves the element using both indices from the *position* tuple, where *position[0]* is the row index, and *position[1]* is the column index. It's important to note that the position coordinates follow standard Python indexing, where addressing an element in the first row and column is achieved with *get(0,0)*. In case of a non-valid index it will raise an exception.

add(array1, array2)

The *add* function adds the elements of the vector or matrix, element by element. First, it checks whether the shapes of the corresponding matrices or vectors are the same and if not it raises an exception. Then, the function performs element-wise sums and appends the sums to a new array of the same size as the original one, both for matrices and vectors.

subtract(array1, array2)

The *subtract* function works in the same way as the *add* function works, however, where the *add* function adds the elements of the vectors or matrices, the *subtract* function subtracts them.

dot_product(array1, array2)

Before computing the dot product, the function checks the validity of both arrays and ensures that their shapes allow for dot product calculation. For vectors, it verifies if they have the same length, and for matrices, it checks if the number of columns in array_1 is equal to the number of rows of array_2. To address the dot product operation, the problem is divided into three cases, each initially checked. The first case arises when both arrays are vectors; in this scenario, the result is a list containing the sum of the respective entry-by-entry multiplications. For the other two cases (dot product of a vector with a matrix or two matrices), each row of the first matrix/vector is multiplied by the corresponding column of the second one, and the sum is appended to a new array list. This array is then reshaped into the correct matrix, having the number of rows of the first array and the number of columns of the second, before being returned. It's worth noting that the operation *dot_product(matrix, vector)* is not allowed, as vectors are represented in a horizontal manner. Therefore, they should be transformed first into a matrix with one column first.

gaussian_solver(array_a, array_b)

The initial step involves verifying that array_a is a matrix, array_b is a vector, and ensuring that the number of rows in the matrix is equal to the number of elements in the vector. Subsequently, an augmented matrix is created by incorporating array_b as a new column in array_a. The process then identifies, column by column, a leading row and transforms the other entries in the same column to zero by executing elementary row operations and dividing the leading row by its leading element, ensuring the pivotal element is equal to 1. Once a row serves as the pivotal one, it is excluded from the potential candidates for pivotal (non-leading rows), and the process continues with the next column. After executing these operations, the augmented matrix attains Row Echelon Form, simplifying the extraction of solutions. If a column has more than one non-zero entry or no non-zero entries, an exception is raised, indicating a Singular Matrix. Otherwise, the method identifies the solutions in the last column and returns them to the user. It's important to note that this method doesn't distinguish between no-solutions and infinite solutions, similar to the behavior of *np.linalg.solve* that it aims to replicate.

1.3 Tests

In our library, we have implemented a comprehensive set of tests for each function to ensure its correct functionality. Despite some instances where it may seem unnecessary, it is standard practice to verify that even if internal processes within functions are modified, the ultimate output remains consistent. Due to the restriction of not using external libraries (e.g., *unittest*), we adapted the implementation. Using assert statements, we have validated corner cases, and with try and except statements, we have confirmed that exceptions are handled appropriately. These tests ensure that each function performs correctly and can be modified in the future, as long as it continues to pass the defined tests.

2 Question 2

The focus of the second question is the Hamming Code and its use in the process of coding and decoding. Coding theory is often concerned with the reliability of communication over noisy channels where errors can occur and therefore error correcting codes are used in a wide range of communication settings (Nuh, 2007). The Hamming codes are a family of linear error-correcting codes that can detect one-bit errors and two-bit errors. They can also be used to correct one-bit errors without detection of uncorrected errors because the program is not able to identify more than one error position, i.e., 2-bit errors, and is not able to give information about other bits in error. The most common use of the Hamming Code is for the (7,4) algorithm, which involves encoding four bits of data into seven bits by adding three parity bits (Epp, 2011).

The Hamming Code works on binary digits, so there can be only 0s and 1s. This was the starting point for our program; to make sure the number inputted could be converted to a four-bit binary value. The program starts by importing the Numpy library and initialising the class called *HammingCode*. Firstly, all necessary variables were defined, and the function *convert_number_to_binary()* is called during the initialisation. This way, in the case of a wrong number being inserted, the operation will be stopped from the beginning without the need to actually call the function. The function *convert_number_to_binary()* makes sure the number inserted is between 0 and 15 and thus can be converted into a four-digit binary value. If this condition is not respected, then an exception will be raised suggesting that the number provided is not in the range and therefore cannot be transformed. Then with the function *checknumber()*, a vector is created from the binary value digits and double-checks the digits are four and binary, after that, we continue by implementing the function for encoding.

2.1 Encoding

Encoding with the Hamming Code requires the use of a matrix called the G (Generator) matrix, whose rows form a basis for a linear code. The G matrix is used to generate the codeword so:

$$w = s \cdot G$$

where w is the codeword, and it's expected to be a row vector, and s is the vector we input from the four-digit value. The standard form of a generator matrix is given by the Identity Matrix and the Parity Matrix (Ling and Xing, 2004).

$$G = \left[I_k | P \right]$$

In our case, the G matrix was given so we proceeded to do the multiplication in the *encode()* function:

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Because the vector is shaped as (1,4) the transpose of G is used to complete the operation, but the result is the same independently of how the matrix and vector are set. It is only important that columns of the first element and rows of the second element match, in order to be able to execute the calculation.

2.2 Introducing Errors

At this point the possibility to simulate errors was introduced. As previously mentioned, the Hamming Code is used to correct the errors that might corrupt data while it's being transmitted or stored. Since while executing the code in the Python environment it is unlikely for the output to be corrupted, a function that simulates a communication error has been created. In the function *introduce_error()* there's the option to introduce either 1-bit or 2-bit errors and to decide the position where to insert them. The process involved copying the encoded message, referred to as "wrongcoded." Subsequently, one or two digits in this wrongcoded message were altered by switching them from 0 to 1 or vice versa.

The same process is applied for the introduction of the 2-digit error. In this case, an additional exception was introduced in case the two positions were identical, as flipping the same digit two times would result in the original matrix. The decoding part begins by checking errors through parity checking and the eventual correction of the 1-bit error. Considering that usually the encoding and decoding are handled by different parties, the functions start by checking if the message can be decoded; therefore validating that it has 7 bits and it is binary. With the function *checkencoded()* these requirements are checked, and the check_wrongcoded Boolean is given so that the function can identify whether to use the rightly encoded or the wrongly encoded message. This is simply based on the fact that it was our choice not to input the value but to use the one created from earlier functions; if the choice was to have a value re-inputted then there would be no need to specify this distinction when decoding.

2.3 Parity Check

Subsequently, the Parity Check is implemented and it detects up to 2-bit errors, but it is only possible to return the position of the wrong digit for 1-digit errors and therefore correct them;

consequently, whenever there's an error the assumption is always that it is a 1-bit error. To perform the parity check it is necessary to calculate the dot product between the message vector and the parity check matrix.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} * \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

When applying the parity check the value obtained is saved in *self.checkvector_wrongcoded*, which is an attribute of the HammingCode class and will be used later in the correction.

Proceeding with the function *correct_error()*, we have created a paritycheck dictionary where all the possible combinations of non-zero parity check matrices and their paired one-bit error positions are stored; the position is the key and the vector showing the possible result from the parity check is the value. Considering the limitations of the Hamming code, to perform the correction we assume the error was a 1-digit error. The function compares the parity check vector we stored as *self.checkvector_wrongcoded* with all the possible values in the dictionary. If it finds a correspondence, it will return the index associated with the position of the one-bit error, and it will proceed to flip the wrong bit to correct the error. Otherwise, it will raise an error saying that no one-bit error was found.

2.4 Decoding

Finally, the *decode()* function is implemented to decode the encrypted message and obtain the original message. The function only decodes the encoded message that had no error because decoding a message with errors would return a wrong result. To decode it is necessary to perform the dot product between the codeword and the R Matrix. The R Matrix in this case was given as:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Finally, the function fetches the initial number inputted using the expression:

original_number=int(''.join(map(str, decoded.flatten())),2).

2.5 Tests

We encoded and decoded four different 4-bit vectors without introducing errors, and we managed to retrieve the initial 4-bit vector. As expected, when performing the parity check on encoded vectors without errors the vector obtained is [[0,0,0]]. After this, we tested our code with one or two bit errors. We started by testing 1-bit errors and as expected, the parity check did not return a zero vector. As it is possible to correct 1-bit errors, the original codeword was restored and thus the initial 4-bit vector and number were retrieved. The same is not true for two-digit errors, in which case it is not possible to restore the original codeword.

2.6 Limitations

When implementing the Hamming Code the limitation of this linear error-correcting method became evident. The Hamming Code can detect errors in up to two digits, but the parity check does not specify whether there is a one-digit or a two-digit error. If the wrong digits were two, then we would not be able to identify or correct both positions. Therefore to correct errors, it is assumed that the error is always a one-digit error. This assumption allowed us to use the Hamming Code to identify the position where the error was located and to correct it. Another limitation of the Hamming code is that it fails to identify errors of more than 2 digits. If for example 3 different digits are flipped in a valid codeword, the resulting vector could be a valid codeword; implying that if a parity check is performed on a 3-digit error, the result may be a zero vector and no error may be detected.

3 Question 3

Question 3 requires the development of a Python program to compute the similarity between a given set of documents. A class named *DocumentSimilarity* has been created for loading and comparing documents (.txt files). When presented with a text name, the class outputs a data frame containing similarity scores to the rest of the documents already present in the Corpus.

To calculate similarity, four distinct methods have been implemented: dot product, distance norm, and two variations of cosine similarity. Users are given the flexibility to select a specific method or compare all methods simultaneously. Regardless of the chosen method, the common approach involves comparing the words across documents by transforming them into vectors. The following variables are consistently used throughout all functions to address this question.

3.1 Key Variables

- **word_bank**: A set containing every unique word present in at least one of the documents. This set updates whenever a new document is added.
- **words_in_each_document**: A dictionary with the document name as the key and the list of its words as the value. It is updated only the first time a document is added.
- **binary_vectors**: A dictionary with the document name as the key and the vector used for calculating similarity metrics as the value. It requires updating each time a new document is added due to changes in the set of all words.
- **frequency_vectors**: Identical to *binary_vectors* in structure, but the vectors within are computed by counting the frequency of word occurrences rather than using a binary representation.

3.2 Document Loading

The initial step involves adding documents to the corpus, using the *read_all_files(folder_path)* function. This function first validates the folder path, raising specific exceptions for different error types. It then identifies every .txt file in the folder and calls the *add_doc(filepath)* function for each.

The *add_doc()* function manages the process from reading the file to storing the clean list of words by using some auxiliary functions. Firstly, the *read_doc()* function reads the document, converting its text to a string and raising an exception if an error occurs. Next, the string is transformed into a list of words, excluding unwanted characters such as spaces, line breaks, or numbers. This is achieved using the *clean_words(doc_as_string)* function, which utilises the *Regex* package to substitute consecutive sets of non-letter characters with empty spaces. This makes it easy to split the string by spaces, resulting in a list of clean words.

The cleaned word list is then stored in the *words_in_each_document* dictionary, where the document name (extracted from the filename before .txt) serves as the key, and the list of words as the value. Additionally, the set of all unique words (*word_bank*) is updated by performing a union with the existing set. Using a set is advantageous as it ensures there are only unique words, and order is not crucial. The last step of *read_all_files* is to call the *update_word_vectors* function which is explained in the next section.

3.3 Update Word Vectors

Once we have already executed the *read_all_files(folder_path)* function, the second step would be to choose a document, compute and return the similarity scores of the document against all other documents in the corpus. This is all done through the *add_doc_compute_similarity()* function. This process can be executed in one of two ways; either by selecting a pre-existing document in the corpus to calculate similarities, or by adding a new document to the corpus, which will involve updating the corpus and the document's vectors before performing the similarity calculations.

The function begins by validating that the requested similarity methods are supported by our functions, otherwise raising an error for invalid methods. It then extracts the file name to check if the document is already in the *words_in_each_document* dictionary. If not, the document is added to the corpus using "add_doc()" and the vectors are updated.

The function *update_word_vectors* is used to update the vector representations of words in each document. It iterates through every unique word in the corpus set, checking its presence in each document, and appends 1 if the word is present and 0 if it isn't. This ensures that each entry in the vector corresponds to the same word for all documents. In addition to this binary approach, the function also creates and updates the *frequency_vectors* dictionary of vectors. This process involves calculating the frequency of each word's appearance, therefore accounting for the significance of the word in the text. The binary and frequency approaches offer distinct perspectives, capturing presence and frequency information, respectively.

It is worth noting that we need to update the word vectors every time we add a new file because the set of unique words in the corpus can change. Once we have these new updated vectors we call and return *get_similarity_all()* and proceed to the last step of our program.

3.4 Similarity Metrics Calculation

The function then calculates the similarity scores of the given document against all others in the corpus by calling the *get_similarity_all* function which then returns a *Pandas DataFrame* containing the similarity scores for all chosen methods.

This function iterates over a list of methods provided and for each of them it iterates over all documents in the corpus, comparing each document to the chosen one using the methods provided. For each document pair, the function calculates the similarity score and stores it in a sub-dictionary. If the document is already in the loop, the function skips this document to avoid comparing it with itself. After computing the similarities for all document pairs and for each method, the function converts the *dict_scores* dictionary into a *Pandas DataFrame* for easier analysis and visualisation.

To compute the similarity between two documents we have implemented the following methods: dot product, distance norm (Euclidean distance) and cosine similarity methods. For all functions that compute similarity Python's *Numpy* library has been imported and used.

Dot product

We have created a function called *dot_product_similarity*, which uses *Numpy's* library to calculate the dot product between two vectors.

Euclidean distance (norm)

To calculate the Euclidean distance we have created the *distance_norm_similarity* function which uses *Numpy's linalg.norm* function. Norm(vector_1 - vector_2) calculates the Euclidean norm (length) of the vector formed by the element-wise difference between vector_1 and vector_2. In mathematical terms the Euclidean norm is expressed as follows:

$$d(p, q) = \|p - q\|_2$$

Cosine similarities

Finally, for the calculation of cosine similarity, we have created two different functions. The first function uses the *binary_vectors* dictionary to calculate the cosine similarity between two vectors. This is done by calculating the dot product of the vectors and then dividing by the product of the Euclidean norm (distance) of each vector. The second function is the same in mathematical terms, however, it uses the *frequency_vectors* dictionary. So *get_cosine_similarity_frequency* calculates the cosine similarity between two text documents but also takes into account the frequency each word appears in the text.

$$similarity(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|}$$

3.5 Limitations

Dot-product similarity fails to effectively measure the similarity between texts of different lengths because it relies on the product of the elements of the two vectors. Consequently, if one document contains more words, there's a higher probability of obtaining a higher dot product, even if the documents discuss different topics. Euclidean distance does not normalise for the text length and it is interpreted in a different way; a value closer to zero indicates higher document similarity. Finally, both dot-product similarity and Euclidean distance do not consider the angle between vectors, useful for understanding contextual relationships between documents.

To address these limitations, we introduced cosine similarity, which normalises each document independently. Cosine similarity is better at comparing documents and providing a score independent of the length of the document. Additionally, it considers the angle between vectors, capturing themes within documents. Frequency-based cosine similarity further improves the capture of thematic differences between documents. For varying document sizes, either form of cosine similarity is more effective, with the frequency-based variant having improved performance at identifying themes within text.

Additionally, to the methods that we have provided, there are other more precise ways to calculate the similarity between documents. One way is removing stop words (words that are most commonly used in language without adding much meaning to the text such as the, a, to). Another way could be to implement stemming; the process of stemming provides the root of words and thus, makes it possible for words with the same meaning but used in different contexts, to be considered as the same word. For example running, runs and run will all be used as run in the calculation of the similarity.

4 References

Epp S., S. (2011). Discrete Mathematics with Applications. De Paul University, pag 389-390.

Ling S., Xing C. (2004). Coding Theory/ A First Course. Cambridge University Press. ISBN 0-521-52923-9.

Nuh A. (2007). An Introduction to Coding Theory via Hamming Codes: A Computational Science Model. Faculty Publications. Paper 4.