Final Assignment Foundations of Programming and Linear Algebra

Group MAFL:
Marlon Alexander Braun (158306)
Leonard Brenk (158287)
Fabian Siedler (158290)
Aaron Wolf (158297)

Examiners:
Raghava Rao Mukkamala
Ashutosh Dhar Dwivedi

Study line:
M.Sc. Business Administration and Data Science

Date of submission: 11.12.22

Marlon Alexander Braun
Leonard Brenk
Fabian Siedler
Aaron Wolf

Table of Contents

Marlon Alexander Braun
Leonard Brenk
Fabian Siedler
Aaron Wolf

Question 1 Sub-Numpy

In the first part of the final assignment, we were tasked to replicate some of the functionalities of the popular NumPy library without using the actual library. Therefore, we created a new implementation called SNumPy and defined several functions within that class, ranging from reshaping an array to the calculation of the dot product. We also provided relevant test cases for our functions and compared our output to the output of the NumPy library.

For the sake of this final assignment, it is important to note that we think of a vector as a row-vector, for example x = [1,2,3] would have one row and three columns. On a similar note, y = [[3],[3],[3]] would be a matrix with three rows and one column.

In the following paragraphs, we provide a brief overview of the functions. Note that our explanations here are rather high-level. Please see the attached code and the comments for a more fine-grained/step-by-step perspective.

Subtask 1: SNumPy.ones(Int)

After having defined the class itself, the first function we defined was SNumPy.ones(Int). This function takes any integer as input and returns a list containing int times "1".

Subtask 2: SNumPy.ones(Int)

SNumPy.zeros(Int) does the same as SNumPy.ones(Int) with only one little difference: instead of "1"s, the output list now contains only "0"s.

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

The reshape function takes an input array and converts it to the dimensions specified by the tuple (row, column). So, it basically converts a vector into a matrix. We included two error messages in case the inserted dimension specification is not a tuple and in case the number of elements in the vector cannot be distributed evenly over the rows of a matrix as specified by the inserted dimension.

Subtask 4: SNumPy.shape(array)

The shape function takes an array as input and gives out the shape of it in tuple form (row, column). To come up with the result, we first check whether the array is a vector or a matrix. Depending on that, the shape is determined and returned.

Subtask 5: SNumPy.append(array1, array2)

The append function returns a new vector or matrix that is the combination of the two input vectors or matrices. Combination here means that the respective array is appended to the right of the initial array. For example, if we append two 2x2 matrices, the resulting first row is composed of the first row of array1 where the first row of array2 is appended to the right. We implemented error messages when trying to append a matrix to a vector and vice versa and when the number of rows of the arrays to be appended is not equal as appending then is not possible.

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

The get function can be used to retrieve a specific value within an array. It takes (row, column) as input and gives out the value of the array at that position. For both matrix and vector, it gives out an IndexError when trying to access an element that is not within the vector/matrix dimension. Furthermore, when trying to retrieve values from a (row-)vector (as specified for this final assignment), the inserted row number must be 0 (note: zero-indexing), otherwise, an IndexError will be thrown.

Subtask 7: SNumPy.add(array1, array2)

The add function is used to add on matrices or vectors. It calculates the sum of all the individual elements that are at the same position regarding row and column number. Arrays can only be added if they have the same dimension. When this condition is not fulfilled, our function throws out a ValueError.

Subtask 8: SNumPy.subtract(array1, array2)

The subtract function is very similar to the add function but is used to subtract matrices or vectors. It calculates the remainder of all the individual elements that are at the same position regarding row and column number. Arrays can only be subtracted if they have the same dimension. When this condition is not fulfilled, our function throws out a ValueError.

Subtask 9: SNumPy.dotproduct(array1, array2)

The last function in our SNumPy implementation is concerned with calculating the dot product of two arrays for the cases where both arrays are matrices, both arrays are vectors, or we have mixed types of arrays. ValueError messages are thrown out when the dimensions do not allow the calculation of the dot product.

Marlon Alexander Braun
Leonard Brenk
Fabian Siedler
Aaron Wolf

Question 2 Hamming's Code

In theory, the Hamming error-correction code is a technique to check whether a transmission error in the form of up to two-bit flips happened in a message transmission. The most intuitive and basic way to determine if a bit in a message flipped is to save the information three times so any inconsistency within the three messages can be fixed by choosing the number that occurs twice within the three messages. However, that means that two-thirds of the storage is used for redundancies. The Hamming Code in contrast only uses $log_2(n)$ bits out of n bits for correction. Whereas 256 bits would result in 512 additional bits to save the bits three times, the Hamming code only uses 8 additional bits. Also, if only one bit is flipped, the algorithm can even detect which bit was flipped. The heart of the technique is the parity bits positioned at powers of two that count the number of 1s in a certain set of bits. A parity bit is 0 if the number of 1s in the considered set of bits is even and 1 if the number of 1s in the considered set of bits is odd. The considered sets of bits overlap each other. With performing a parity check for the parity bits a bit flip can be identified and located using a binary-search-based approach that always divides the possible solution set in half until only one bit remains - the one that is an element of the intersection of all sets that have an odd number of 1s. For a message to be transmitted correctly, the sum of all bits in a certain dedicated set of bits - including the parity bits that represent that set - needs to be even. If it is odd, there has been at least one bit flip in that set. For identifying a two-bit error the first bit of the message comes into play - called the extended Hamming Code. It represents the overall number of 1s and evens it out if it is odd. If there is an even parity overall but odd parities at the other bits one can see that there had been at least two-bit flips. If it was only one, the overall parity bit in position 0 would have flipped too. However, as we are only using (7,4) pairs and a Generator Matrix with only 7 rows, we do not have that bit and thus can only detect max. one-bit flip. To have the extended Hamming Code bit, we would need a (8,4) pair and an 8th row in the generator matrix.

Subtask 1: Encoding Parity check

By multiplying a message with the generator matrix the message is transformed into the Hamming-encoded message. That means it now carries the parity check bits. In the matrix, every row that only contains one 1 copies the respective value to one of the message bits, while rows with 3 1s create the parity values during the multiplication. If the message is to be read at the receiving end, multiplying the message with the decoding matrix will result in the original matrix given there has not been a bit flip. In order to check for bit flips the parity check matrix is used. If there has not been a flip, the result of the encoded message multiplied with the parity check matrix will result in the nullvector.

Subtask 2: Decoding

For tests see the code. In order to transform the received encoded message back to the original message, it needs to be multiplied with the decoder matrix. That matrix contains four 1s, just as many as there are message bits in the encoded vector.

Subtask 3: Test & Extend

For tests see the code. Using the given matrices only one bit flip can be detected, as the generator matrix only has 7 rows while 8 would be needed for the extended hamming code. Thus the used matrices in this task do not calculate the parity bit that evens out the overall number of 1s.

Question 3 Text Document Similarity

Introduction

For this task, it was necessary to write a program that can calculate the similarity between different text documents. The program designed by us takes 2 arguments as input: One list of strings, making up the text corpus and an input string which is the text document that should be checked for similarity.

The text corpus and input text must be assigned before running the program. Since processing the two inputs needs multiple steps, we decided to first write individual functions for each of these steps, and finally implement a function that brings everything together, referencing the previously created functions for each step. In the following, the individual functions shall be described.

Step-by-step explanation of the process

First, it was necessary to turn the text corpus, so the list of strings, into a dictionary of words that are present in the text corpus. For this, we implemented the function “createWordDict” which simply takes the text corpus as an argument. In the beginning, we initiate a dictionary to later keep track of all words and their occurrence (1 for ’occurs’, or 0 for ’does not occur’). In the second step, we iterate through all sentences in the text corpus and initially perform some cleaning and modification steps. The first cleaning procedure is to use regular expressions to remove any characters in the sentences which are not letters, numbers or whitespaces (to still be able to identify individual words). Then we turned the whole sentence to lowercase letters so that words will be recognized no matter if they are written in capital or lowercase letters. The next step was to split the sentence up into individual words which should be added to our dictionary. For this we simply used the “split” function, splitting on each occurrence of whitespaces. Our goal was to create a dictionary which only contained unique words and no duplicates so we don’t get any problems with the vectoriation of input sentences later. To ensure this, we used list comprehension to reduce the list of words from the sentence to words which are not yet present in the keys of the dictionary (representing the individual words). Finally, for each of the remaining words, we create an entry in our dictionary and set 1 as the value. These steps happen as long as there are sentences left in the list of sentences for the text corpus. In the end, the function returns the word dictionary.

Since the assignment was to supply a list of sentences that the input is similar to, our second step was to vectorise each sentence from the text corpus individually. This step is performed in the function “textToVectors”, which takes the text corpus and the word dictionary from the first function as input. First, a dictionary is initiated which keeps track of the sentences and corresponding vectors later. Next, we again iterate through the list of sentences and perform the same cleaning and modification steps as above (removing unwanted characters, turning the sentence to lowercase and finally splitting the sentence into a list of words), additionally we initiate a NumPy array. After the cleaning and modification were performed, we iterate through the dictionary of words (to ensure an equal dimensionality of all word

vectors) and if the word is occurring in the sentence, we count the occurrences of the word, multiply this with the value 1 from the dictionary and append it to the vector, if it is not occurring, we append a 0. When this loop is done, the vector is stored in the earlier created dictionary as a value, with the sentence (in the original, unmodified form for readability) as the key. This process is repeated for each sentence in the text corpus. Finally, the dictionary containing the sentences and their respective word vectors is returned.

The third step is to vectorise the input text. The procedure is exactly the same as in the function "textToVectors", the only difference is that we do not iterate through a list of sentences, but only do the process once. We also return a dictionary with the sentence (again in original form) as the key with the respective word vector as value. The functions to vectorise the text corpus and the input text could be merged, however, to gain a deeper understanding of the process steps and to make it easier for the reader to follow our logic, we decided to write two functions.

The last step is to calculate the similarity of the input text to the sentences from the text corpus. For this, we created the function "calculateSimilarity". The first part of the function is requesting input from the user to determine which distance measure should be used. Our program offers the choice between the dot product and the Euclidean distance. We request that the user inputs "1" for the Euclidean distance and "2" for the dot product, or SStop"to end the program. This happens in a while loop, which checks if the input is valid (so either "1" or "2", or SStop") and only is excited if it is valid, otherwise the loop continues to run and requests input from the user again until the input is valid.

The second part of the function is about actually calculating the distance/similarity of the input vector to the individual word vectors of the text corpus. To keep track of the results, a dictionary is initiated which stores the sentences of the text corpus and their result for the similarity/distance measure. Then we loop through the previously created dictionary containing the sentences from the text corpus and their corresponding word vectors, and first, save the vector to a new variable. Depending on the distance measure that was chosen (the procedure is determined with an if-statement), either the dot product or Euclidian distance between the vector created from the input text and the vector created from the sentence of the text corpus is calculated. The result is stored as a value in the initiated "results" dictionary, with the sentence from the text corpus as the key. This procedure is repeated for each text vector from the text corpus. Finally, the function returns the results dictionary and the chosen distance measure.

In the last step, the function "TextDocumentSimilarity" brings all the previously created functions together. So first, the text corpus is turned into a dictionary of words, then each sentence from the text corpus is vectorised, the input text is also vectorised, and finally, the distance measure is calculated between the input text and the sentences of the text corpus. Finally, the results are printed, including interpretation information depending on the chosen distance measure.

To print the individual sentences when the dot product was chosen, we iterate through the "results"dictionary and print the keys (so the sentences of the text corpus) and the results for the dot product in descending order (since a high dot product signals a high similarity). For the Euclidean distance, the keys (so the sentences of the text corpus) and the results for the Euclidean distance are printed in ascending order (since a low distance signals a high similarity).

Behavior of the different measures

Regarding the different measures chosen, it can be observed that the dot product is highly sensitive towards word repetition. In our example code, we included an exemplary sentence which has 2 words in common with the input sentence, but these 2 words occur multiple times in the sentence of the text corpus. Using the dot product, this sentence gets the highest similarity score with the input text (13), whereas a sentence where all words occur also in the input, but only once ("Most emissions are produced by industrialized countries"), gets positioned at place 2 in terms of similarity. When choosing the Euclidean distance, the sentence with 2 matching words occurring multiple times is rated as the least similar sentence, and the one where all words also occur in the input is the most similar. So depending on the preference of the user, this should be kept in mind. If the impact of repetitive words on the similarity score should be high, the dot product would be more appropriate. However, if the overall context and therefore the total number of absolute words that match should have a higher impact, the Euclidean distance might be the better choice. Either way, both measures in their original form have the problem that they are quite hard to interpret on their own since their results, in theory, can reach infinity. Therefore only in the context of multiple results, the individual result can be interpreted.

Sources:

- Grant Sanderson, 2020, How to send a self-correcting message (Hamming codes), https://www.youtube.com/watch?v=X8jsijhllIAt=281s
- Wikipedia, 2020, Hamming Code, https://en.wikipedia.org/wiki/Hamming_code
- Geeks for Geeks, 2022, Hamming Code in Computer Network, https://www.geeksforgeeks.org/hamming-code-in-computer-network/