Final Exam project

Final Examination Paper - Report
Foundations of Data Science

Submitted by	Natalie Schober Kitti Kresznai Anastasiya Vitaliyivna Strohonova
Study Programme	M.Sc. Business Administration & Data Science
Submitted on	21 December 2020
Number of pages	15
Number of characters	31534

Contents

1 Introduction

For our final exam project, we were tasked with three question to answer using Python 3 and the mathematical foundations from this course. More specifically, in this project assignment we refer to topics within the field of programming in Python, including object-oriented programming, the python library Numpy as well as arrays and matrices in Python. On the other hand, we required general Linear Algebra topics such as operations with matrices and vectors. In our report, we will be explaining each implementation in Python in detail, following the structure of the questions. The Linear Algebra foundations we required are attached in the appendix.

2 Question 1: Sub-Numpy

We were tasked to create a Python class that is similar to the Python library Numpy and define some methods within this Snumpy Class. These ten functions use methods from the mathematical field of Linear Algebra to carry out calculations with matrices and vectors.
Generally, a class is an abstract data type which binds together a set of objects and the operations on those objects [1, p.91]. Classes can have functions attached to them which are, then, called methods or method attributes [1, p.93].
Our starting point was the question how a matrix or a vector should look like in Python if we cannot use the convenient Numpy Array. For simplicity reasons, we have chosen to use a list in the case of a vector, and a so-called nested list in case of matrices. In the python context, a list is a standard mutable multi-element container which can contain multiple data types, including more lists [2].

2.1 Question 1.1

The goal of this method is to create a vector containing only ones. Recall that a vector is a list by our definition. Therefore, our implementation of the method takes an integer n as input describing how many elements this vector contains. Then, we create this vector by multiplying a one contained in this list with the input integer. This new list with n elements is, then, returned by the function.

2.2 Question 1.2

The goal of this method is the same as in Question 1.1, with the difference that this method should create a vector of zeros. Therefore, our implementation is similar to the implementation in Question 1.1. Our input is an integer n representing the length of the new vector. Once again, we are multiplying a list containing a zero with n. The function, then, returns this list with n zeros.

2.3 Question 1.3

The reshape function takes a vector and intends to reshape it into a matrix with the specified dimensions. Hence, our implementation takes a vector in list form and a tuple containing the dimensions as input for the function. In the Python context, tuples are defined as ordered sequences of elements[1, p.56]. Our first steps in the function are extracting the vector length and requested number of rows and columns from the given parameters. Next, we multiplied the number of rows and columns in order to compare this to the length of our vector and confirm that splitting it into the specified parameters was possible. We have implemented some error handling in a conditional statement which notifies the user that the dimensions are not compatible. If the statement is true, we loop through the

specified number of rows and create each of the rows of the new matrix. Each row within the matrix consists of its own list, which once filled with the values from the vector up to the specified column number, we append to our main matrix until it is complete. Then, the function returns the final matrix.

2.4 Question 1.4

Question 1.4 asks for a shape function which returns the dimensions of a given matrix or vector. As a first step, we calculate the length of the input (row number). Next, we are differentiating between a matrix and a vector. A vectors column dimensions will always be 1, but for a matrix we calculate the length of the first list at position 0 to find the number of columns. From the length of both lists, the function can calculate and return dimensions of the matrix in a tuple.

2.5 Question 1.5

The goal of the append function in Numpy and our Snumpy class is to append two matrices or two vectors. First, we have implemented a check to confirm whether each array is a matrix (nested list) or a vector (list). However, according to our definition, a matrix is a nested list, and simply calculating the length of the outer list does not give any information about how many elements the inner lists contain, i.e. information about the number of columns. First, in our code, we have to establish whether the first element within the outer list is a list too which is indicative for a matrix. So, in the case of two matrices, we are calculating the column dimension. After this, our function includes some error handling for the case when the dimensions of the matrices differ. If they do not differ, we append each row in the second matrix to the first and will then return our resulting matrix. In the case of vectors, at first, we handle the event when the user tries to append a matrix to a vector with an appropriate error message. As in the Numpy append function when the axis is set to zero, our method does not allow for the case of appending a vector to a matrix or vice versa. If both arrays are vectors, we add the elements of the two vectors to a new resulting vector which is then returned by the function.

2.6 Question 1.6

Our get method's task is to return a specific value defined by coordinate points from a matrix or a vector. Its inputs are an array, which can either be a vector or a matrix, and a tuple with the coordinate points. At first, we check if our array is a matrix, in which case we calculate its dimensions. The next step is to evaluate whether the coordinate points are inside the range of the dimensions of the matrix, and make sure they are not negative values. If the input values don't fail on these conditions, our method returns the requested value from the matrix. After this, we treat the input array if it is a vector. As we are handling vectors in vertical forms, the column dimension will always be equal to 1, while the length of

the given array is treated as the row dimension. Error handling is also implemented here, which checks if the given positions match with the dimensions of the vector and filters out any negative values. Given it doesn't produce any errors, the function returns the value from the vector at the specified position.

2.7 Question 1.7

The next method in our Snumpy class is the add method, which is capable of the addition of two matrices or two vectors. For that reason, it takes two arrays as input, which can either be matrices or vectors. The first if statement tests if both input arrays are matrices, and in this case, each of their dimensions is assigned to the variables r1, c1, r2, and c2. In the next step, we create a new matrix that will hold the result of the addition. This new matrix C is created by appending a list with r1 empty lists which in the next step is filled with c2 zeros. Then, given that the dimensions of the matrices match (otherwise, we have provided error handling), the method performs the addition for every element in a nested for-loop (based on the indices corresponding) and finally returns the matrix with the result. In the second half of the code, we handle vector addition, where after confirming both vectors have the same number of rows, the corresponding elements are added, appended to the result vector, and returned as a list.

2.8 Question 1.8

The structure of the subtract function is the same as the add function's, only the completed operations differ. Hence, the inputs are two arrays again, and after checking if those are matrices or vectors, the method completes the subtraction accordingly. The same error handling implementations ensure the correct behavior. Namely, they don't allow the subtraction if the input arrays are not the same type, and if the two matrices or vectors are not in the same size.

2.9 Question 1.9

The dot product function takes two arrays, which can be vectors or matrices, and returns the dot product of those two arrays. The first element in our code is an if statement which checks if the two inputs are matrices or vectors, or if the second array is a vector. In case of two matrices, we assign the first array's column, and the second one's row dimension to variables, and compare those values. This is necessary for determining if it is possible to calculate the dot product, as the number of columns of the first matrix must be equal to the number of rows in the second matrix. If there are no errors, we transpose the second matrix, and compute the dot product with the help of repeatedly nested for loops. Similarly, we calculate the dot product for when the function has been given a matrix and a vector. When the inputs are vectors, we carry out an element-wise multiplication using a for loop and sum up the results.

2.10 Question 1.10

The goal of the linear solver function is to find the solutions of a system of linear equations utilizing the Gaussian elimination and row reduction method as described in the appendix. As input, our function takes a matrix A which represents the coefficient matrix and a vector b which represents the results of the linear equations. Our first step is to check whether the coefficient matrix is actually a matrix. If it is not, we raise a Value Error. If it is, we calculate the length of the linear system. Then, we have included a second check which compares the column number with the length of vector b. We require this because there needs to be an equation for each variable in the linear system. Next, we loop through each row in the coefficient matrix. In this loop, we create 1s on the diagonal of the coefficient's matrix and zeros outside the diagonal, we aim at creating the reduced echelon form. We create the zeros outside the diagonal by iterating through each element in that row which is outside of the diagonal and adding a multiple of another row to the current row in the matrix A as well as to the respective element in vector b. Once the matrix A is in reduced echelon form, the function returns a tuple containing the matrix A and the x vector which is essentially the modified vector b. Our reasoning for this is: the user is able to check whether the matrix A is actually in reduced echelon form and whether the system has a solution.

3 Question 2: Hamming Code

Hamming Code is a linear binary block code, “linear because it is based on linear algebra" and “block because the code involves a fixed-length sequence of bits” [3, p.211]. Error-correcting codes are used for the transmission and storage of data. The purpose of parity bits in Hamming Code is to identify the presence of an error and narrow down the location of an error. This is possible because each code represents three of the digits from the 4-bit binary value. The value of a parity bit can be found by adding the values of set digits of the 4-bit value and then taking a modulo of 2. The chart below shows which aspects of the 4-bit value, and therefore the 7-bit vector, each parity bit overlaps with [4]. This overlap is what allows for the location of the error to be narrowed down.

	p1	**p2**	**d1**	**p3**	**d2**	**d3**	**d4**
Parity bit 1 (p1)	Yes		Yes		Yes		Yes
Parity bit 2 (p2)		Yes	Yes			Yes	Yes
Parity bit 3 (p3)				Yes	Yes	Yes	Yes

Given the generator (G), parity check (H), and decoder (R) matrices provided, it is clear that the vectors throughout the code must be vertical rather than horizontal so that the number of columns of the matrices may be equal to the number of rows of the vectors.
For our code, we imported two libraries: NumPy (as “np”) and Random. We utilized the NumPy (Numerical Python) library in order to work with the 4-bit binary value and 7-bit vector both in the form of an array, and the G, H and R matrices (as multi-dimensional arrays).

First Step: Creating or Checking a 4-Bit Binary Message

Our first step is to either randomly generate a 4-bit binary code word or check the validity of a code word that has been input by the user. To generate a random code word we begin by creating an empty list under the variable msg, and then in a range of 4 using the random.choice([0,1])function to select either a 0 or 1 to be appended to the list. We then convert the resulting list into a numpy array to be used in the remaining functions. To check the validity of an entered code word, we take a list as input and iterate through the digits within the list, confirming that they are either 0 or 1 (binary) and that the list is 4 digits in length. If these conditions are met then the loop continues, finally returning the list as a NumPy array, otherwise the user will receive an error message.

Second Step: Encoding the Message

In order to encode a 4-bit binary value, we took the dot product of the G matrix and the 4-bit value with the NumPy np.dot()function, and then used the built-in modulo (%) operator to find the remainder of division by 2. Using modulo keeps the output in binary form as necessary for Hamming code. The result is a 7-bit vector. By taking the dot product of each row of the G matrix with the 4-bit codeword, the generator creates the 7-bit vector following the pattern in the figure above. The original message is joined with 3 parity bits, which are calculated using the message.

Third Step: Noisy Channel Simulation

As a message passes through the noisy channel bits can be flipped (changing a value in the vector from 0 to 1 or 1 to 0). To simulate this process, we created a function which takes the message and number of bits to be flipped as input. First, we create a copy of the message to preserve the original. Then we create a list ranging the from 0 to the length of the message (in our case 7) using the * operator to unpack the individual values, and random.sample()to select at random a digit for each of the desired number of bits to be flipped. These random numbers are used as the index to flip a part of the message. We do this with a simple for loop iterating through the list sample list, if noisy[i] == 0 then it is changed to be 1, else a 1 is changed to a 0. Once complete, the list containing errors is returned. Since Hamming code is only able to detect 1- or 2-bit errors and only able to correct 1-bit errors, these two requests are best suited for the function. However, we have chosen to only return an error when more than 3 bits are requested to be flipped, to allow for better testing of the code.

Fourth Step: Parity Check & Correction

In the case of a 1-bit error, the message recipient must figure out the error vector (e) in order to return the message to its original state. The transmitted message is equal to the original message plus the error vector. When one error is introduced into the 7-bit array, the dot product of the H and the 7-bit array after calculating the modulus of two will leave a 3-digit syndrome vector, which will confirm both the presence and location of the error. If the vector returned is null then it can be concluded that there is no error. This is because the set of possible codewords is the null space of matrix H. This leaves the H multiplied by e equal to H multiplied by the transmitted message [3, p.212].
Otherwise, the returned vector is compared to the H matrix. The column with which it matches, is the location of the error in the 7-bit array. To perform this comparison in Python, we first transposed the parity check matrix to have each column as a array within the main array for the matrix. This was done using np.transpose(). We then used a for loop to iterate through the arrays within the H matrix, and if/else statements to add a 0 to a new list (which we called error_vector) for each array

unequal to that from the paritycheck calculation (which we called par), and a number 1 when equal. The e vector, as described by Klein, is "the vector with 1's in the error positions" [3, p.212].
To correct the 1-bit error, we simply used np.add(noisy,error_vector)%2, which adds the transmitted vector (which may contain an error) to the error vector and takes a modulo of 2. This will flip the one bit which erroneously flipped back to it's original state. We considered an alternative route for this portion of the code, where we converted the arrays to lists in order to obtain the index of the par list equivalent within H matrix and use this index in the 7-bit vector to flip for correction. However, we decided against this route, believing it would be best to follow the logic of the math behind Hamming code and to avoid converting the arrays to lists unnecessarily.
In the case of a 2-bit error, the parity check will return a non-zero array which will signify the presence of an error. However, error location and correction will not yield the correct values due to the fact that Hamming code cannot generate two codewords when H is multiplied with par. Without being able to distinguish the two errors, the correlation to the arrays in the H matrix is no longer clear and there can be more than one way to reach the same error syndrome. Hamming code is still able to reliably identify a two-bit error (but not more than two bits of error, for instance) because with only two bits flipped the parity bits are still able to discern when there is an inconsistency.

Fifth Step: Decoding

After the matrix has been corrected (or if no correction was necessary), we are able to decode it back to the original 4-bit binary value by finding the dot product of the Hamming Decoder Matrix and the 7-bit vector. After taking the modulo of 2, we are left with the original binary value. Decoding is only successful when only one bit of error was introduced in the noisy channel.

Sixth Step: Testing

We took several approaches to testing our Hamming code program. Our first step was to combine the elements described above into one function to simplify the process for running the code. We created two versions of our overall Hamming code function. The 1st is called customHamming and the 2nd autoHamming. The only difference being that custom allows a custom 4-bit binary message to be entered by the user in the form of a list, while the auto program randomly generates a list on the user's behalf. We believe both options are important, depending on how the user intends to use the program. Both functions also return a message to encourage the user to check for error, dependent on which limitation of Hamming code an error may stem from.

To begin testing the code we ran the customHamming program and input messages to ensure that each of the 16 possible 4-bit words that can be transmitted were tested. This allowed us to confirm that each of the 16 messages could also be corrected when a single bit was flipped. However, because our noise function randomly introduces error, we could not test every scenario/know that it was being tested. To help with this, we created a function named MessageCorrection to run the customHamming program repeatedly on one message with one bit of error introduced each time to ensure it was always decoded correctly. We unsured decoding was correct by confirming two things, that an error was introduced (meaning the syndrome vector was not null) and the starting message and decoded message were identical. When running the Hamming program 10,000 times, the rate of correction was 100% for this small experiment. We did not allow for the number of errors to be increased, since Hamming code is unable to correct errors greater than one bit.

Our autoHamming program introduced automation to simplify testing, but made it difficult to ensure each of the 16 4-bit messages was tested. Because of this, we decided to build a test which would confirm the ability of our program to identify errors (which we named ErrorIdentify). Identifying an error was defined by the syndrome vector not being a null vector, as explained previously. We used this to test 1-bit errors, as well as performance with 2- and 3-bit errors. We confirmed that when running 10,000 trials, zero bits of error were identified 0% of the time (as expected), 1-bit errors 100% of the time, and 2-bit errors again 100% of the time. When testing the code on 3-bit errors, we found that they were identified about 80% of the time. However, this is consistent with the fact that Hamming code cannot identify 3-bit errors, since in some cases where three bits are flipped enough change has occurred that the error cannot be identified with the help of the parity bits.

4 Question 3: Text Document Similarity

We carry out the following steps to compute the text similarity between documents. When a search document is imported, the code should calculate and display the similarity to a previously imported list of documents.

First step: Import documents

First, we import the documents we want to compare into Python using the OS module. This module includes a wide range of functions to interact with file systems. We use os.listdir (path) and os.path.join(path, file) to get the working directory where the documents are stored and open the files found in the specified folder. Error handling is implemented here in order to check if the provided files are indeed have a .txt extension. If it proves to be true, the TXT documents are read in using the Python built-in function open(). We decided to use TXT files as they can be recognized and processed by most software programs. TXT files allow to store plain and unformatted text which is sufficient for our purpose.

Second step: Clean and split text into words

After the documents are imported, we extract the words from the documents. To obtain each single word, the text in the documents is separated by the Python build-in function split().

In many languages, words at the beginning of a sentence are capitalized, whereas words in the rest of the sentence are written in lower case. We assume it is irrelevant whether a word is in lower or upper case. Therefore, we set all the letters to lower case using the build-in function lower(). Special characters, as listed in unwanted_punc, should not be considered either, so these characters are replaced by a space.

Third step: Store the word frequencies in a dictionary

In Python, dictionaries are mappings, i.e. dictionaries map values to particu£lar keys. Python uses curly braces to list key-value pairs. While colons are used to join a key and a value; commas separate key-value pairs. Dictionaries are mutable elements, so the value mapped to a key can be modified after creation. The main purpose of the dictionary is to look up a value that belongs to a certain key. This is done by index notation. [5, pp.317f.].

For this step, we create the dictionary frequency where we store the unique words. We define the unique words as the key and the absolute frequencies of the words as the value of the dictionary. When the word appears in the text for the first time, the counter of the word, i.e. the value of the dictionary, was set to 1. If the word already occurred in the text, the counter is simply incremented by one.

Fourth step: Creating a word vector for each document from the input list and the search document

After creating our frequency dictionary storing the unique words found in the input documents, the next step is to turn each of those documents to word vectors. This is implemented in the create_word_vectors function where we create a vector called word_vector for each input file, examine if each word from the frequency dictionary is present in the document, and append 1 to its newly created word vector if it is, 0 if it isn't.

For storing these vectors in one place, we define a list_of_word_vectors dictionary and save the name of the documents and their corresponding word vector as key-value pairs.

A word vector for the search document is also created here the same way we have just described, after cleaning the input also provided as a text file, using the method from the second step. Error handling is included here, which checks if a file with the given name exists in the folder.

Fifth step: Compute the text similarity

The word_vectors dictionary and the search_doc_vector list are now used as parameters in the function calculate_similarity. The function computes the similarity between each word vector and the search document vector, i.e. the distance between the vectors, expressed in mathematical terms. To be able to compute the dot product and the Euclidean distance between them, these inputs are converted into one-dimensional Numpy arrays. For the mathematical foundation, please refer to Appendix A. Numpy arrays are an essential part of data manipulation in Python, as they allow to easily access and reshape data. Newer libraries such as Pandas, that we will use later, are built around Numpy arrays [5, p.42].

1. **Dot product** The concept of the dot product is explained in detail in Appendix A. In Python, we compute the dot product between arrays by numpy.dot [6, p.93]. With the help of a for-loop, we iterate through each vector contained in word_vectors and calculate their dot product with the search document vector. As both parameters are converted into one-dimensional arrays, the dot product is equivalent to the inner product of vectors. We save the results in the results_dot dictionary, where the keys are the names of the files, and the values are the corresponding dot products.

2. **Euclidean distance** We compute the Euclidean distance using numpy.linalg.norm, using the same for-loop as for calculating the dot product. In addition, a results_eucl dictionary is defined for storing its outputs.

After the for loop, we order the results_dot and the results_eucl dictionaries into a descending order based on the similarity. In the case of the calculated dot products, it means a descending order of the results, because the larger the dot product, the higher the similarity. For the Euclidean distance,

it is the other way around, and the lower numbers mean that the two files are more alike. Finally, the function returns the ordered keys of both dictionaries.
So far, we used two measures of distance between vectors. The decision which measure performs better cannot be generalized but depends on the particular purpose we need it for. To recall the definitions of the measures we used: The Euclidean distance is declared as the distance between the vectors´ ends. The dot product is the cosine of the angle between the vectors multiplied by the vectors´ lengths. That is, the dot product increases with the length of the vectors. As the dot product accounts for the vectors´ length, we would slightly prefer it over the Euclidean distance in this case. The reason is that we have vectors with large lengths in our text similarity exercise [7, p. 2].

Sixth step: Provide the computed results

Inside the text_similarity function, all the above-mentioned functions are nested into each other, so the user only has to call this function in order to calculate the similarity between the documents. In other words, it is a so-called wrapper function. Its inputs are a path leading to the list of documents stored in one folder and the individual search document.

Seventh step: Testing

For the purpose of code testing, we use some random Lorem ipsum text documents. In the following, we display and explain the output of some selected variables that are most relevant for comparing text similarity.
After the working directory is specified in the variable path, the txt files stored in this working directory are imported. In search_doc, the name of the search document is provided. In case the working directory contains some files that are not in txt format, the following error message is displayed:

```
ValueError: The files found in the specified folder are not txt files.
```

After special characters are removed, words are split and set to lower case, separate_text_documents returns the words that occur in each txt file. For file1.txt, the following is outputted. 'Vel' and 'in' occur multiple times in the list, because the words are not uniquely filtered yet.

```
{'file1.txt': ['duis', 'autem', 'vel', 'eum', 'iriure', 'dolor', 'in',
'hendrerit', 'in', 'vulputate', 'velit', 'esse', 'molestie', 'consequat',
'vel'...}
```

The dictionary frequency displays the unique words and their frequency as key-value pairs.

```
{'duis': 5, 'autem': 3, 'vel': 6, 'eum': 3, 'iriure': 3, 'dolor': 16, 'in': 6,
'hendrerit': 3, 'vulputate': 3, 'velit': 3, 'esse': 3, 'molestie': 3,
'consequat': 5, … }
```

Word vectors created for both the list of documents and the search document. For the search_doc_vector, the following is returned where 1 stands for a newly created word and 0 if not.

```
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

Finally, a list of documents is displayed in descending order according to their similiarity with the search document. As we get the same results for both similiarity measures, we cannot conclude that one performs better than the other for our sample files. In both cases, file2.txt is most similar to the search document, whereas file.txt shows the least similiarity.

```
Result for dot product: dict_keys(['file2.txt.txt', 'file3.txt.txt',
'file.txt.txt'])

Result for euclidean distance: dict_keys(['file2.txt.txt', 'file3.txt.txt',
'file.txt.txt'])
```

5 Conclusion

As we worked through the three questions, we were able to see the deep connection between programming and linear algebra, especially the prevalence of matrices in calculations. In rebuilding portions of NumPy functionality, we utilized object-oriented programming to develop our own Snumpy class. In creating instance methods under this class, we were able to define and group our functionalities together. All of the questions required an understanding of general matrix features and how the dot product is calculated, in question one for the rebuilding of the NumPy dot product function, in question two for understanding the encoding, parity check and decoding processes, and in question three for understanding how document similarity is calculated. In general, a foundational understanding of linear algebra has been proven to be essential for many computations in Python and we have only covered a few scenarios in which it is utilized.

References

[1] John V. Guttag. *Introduction to Computation and Programming Using Python, revised and expanded edition*. MIT Press, 2013. Google-Books-ID: 69IyAgAAQBAJ.

[2] Jake VanderPlas. *Python Data Science Handbook: Essential Tools for Working with Data*. "O'Reilly Media, Inc.", November 2016. Google-Books-ID: xYmNDQAAQBAJ.

[3] Philip N Klein. *Coding the matrix: Linear algebra through applications to computer science*. Newtonian Press, 2013.

[4] Hamming(7,4), November 2020. Page Version ID: 987401370.

[5] John M. Zelle. *Python Programming: An Introduction to Computer Science*. Franklin, Beedle & Associates, Inc., 2004. Google-Books-ID: aJQILlLxRmAC.

[6] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[7] Scalar Products and Euclidean Distances. In Ingwer Borg and Patrick J. F. Groenen, editors, *Modern Multidimensional Scaling: Theory and Applications*, Springer Series in Statistics, pages 389–409. Springer, New York, NY, 2005.

[8] Gilbert Strang, Gilbert Strang, Gilbert Strang, and Gilbert Strang. *Introduction to linear algebra*, volume 3. Wellesley-Cambridge Press Wellesley, MA, 1993.

[9] David C. Lay. *Linear algebra and its applications*. Addison-Wesley, Boston, 5th ed edition, 2012. OCLC: ocn693750928.

A Linear Algebra

Here, we are going to explain which concepts of Linear Algebra we used to implement the functions in Question 1 to Question 3.
Linear Algebra is built on vectors and matrices. A vector can be thought of as a list of numbers that are called entries or elements of the vector, so a vector with four entries is called a 4-vector over $\mathbb{R}$ and can be written as $\mathbb{R}^4$ [3, pp. 80-81], while "a matrix over F is a two-dimensional array whose entries are elements of F" [3, p. 185]. A matrix can be referred to by its dimensions i.e., its row and column numbers. A m x n matrix is a matrix with m rows and n columns [3, p. 185]. For instance, a vector is a m x 1 matrix.

In Linear Algebra, you can perform operations on matrices and vectors such as addition, subtraction or multiplication. Vector or matrix addition is defined in terms of the addition of their corresponding entries [3, p. 86]. Vector addition can be graphically represented as such [8, p.2]:

$$\boldsymbol{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \quad \text{and} \quad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \quad \text{add to} \quad \boldsymbol{v} + \boldsymbol{w} = \begin{bmatrix} v_1 + w_1 \\ v_2 + w_2 \end{bmatrix}$$

This requires that the vectors or the matrices which are to be added have the same dimensions. The subtraction of vector works similarly, the corresponding element in the second matrix is subtracted from the first element. Multiplying a vector by a scalar is defined as multiplying each entry of the vector [3, p.90]. Another operation that could be done on vectors and matrices is the so-called dot product. Graphically, this can be represented as follows, given two vectors u and v:

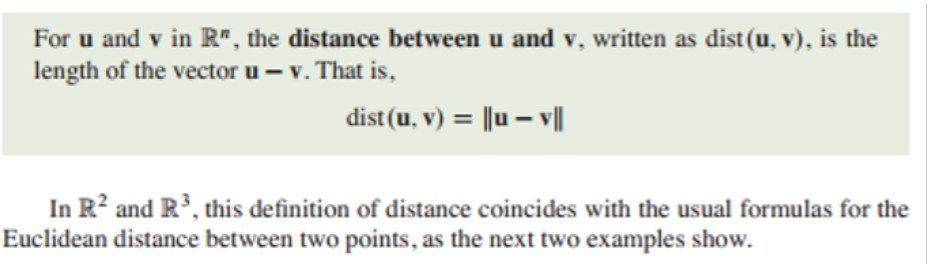
$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} \quad \text{and} \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

then the inner product of **u** and **v** is

$$\begin{bmatrix} u_1 & u_2 & \cdots & u_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} = u_1v_1 + u_2v_2 + \cdots + u_nv_n$$

The dot product is a form of multiplication in which the products from the elements in vector u and v are summed. In the case of a matrix, if M is an R x C matrix and u is a R-vector, then $u*M$ is the C-Vector v such that v[c] is the dot-product of u with column c of M [3, p.201]. Simply put, each value within each row of the first matrix is multiplied by the corresponding row of the second matrix,. Then, the sum of these is taken on a per row basis once the calculation has been done for each value within the row of the first matrix. The number of columns in the first vector/matrix must match the number of rows in the second vector/matrix, and the resulting vector or matrix will have the same number of rows as the first matrix and columns as the second matrix.
Vectors can be represented geometrically in a coordinate system. As such, one can calculate the distance between two vectors which is called the Euclidean distance, also referred to as the distance

norm [9, p.335]. The distance between two vectors u and v can be defined as the length of the difference between the two vectors $u-v$ [3, p.418]. The length of a vector v is typically called norm and written as v [3, p.418].
Mathematically, it can be expressed like this:



For **u** and **v** in $\mathbb{R}^n$, the **distance between u and v**, written as dist(**u**, **v**), is the length of the vector **u** − **v**. That is,

$$\text{dist}(\mathbf{u}, \mathbf{v}) = \|\mathbf{u} - \mathbf{v}\|$$

In $\mathbb{R}^2$ and $\mathbb{R}^3$, this definition of distance coincides with the usual formulas for the Euclidean distance between two points, as the next two examples show.

Linear Algebra and computations based on matrices play a very important role, especially when it comes to large scale data calculations. One important method to perform on matrices is the so-called Gaussian Elimination or the Row Reduction Algorithm to solve systems of linear equations.
A linear equation in the variables x_1, x_2, ..., x_n is an equation that can be written in the form

$$a_1x_1+a_2x_2+\cdots+a_nx_n=b \tag{1}$$

where b and the coefficients a_1,..., a_n are real or complex numbers [9, p.2]. A system of linear equations is a collection of one or more linear equations involving the same variables [9, p.2]. This system of linear equations can be represented in a matrix. The matrix containing the coefficients is called the coefficients matrix[9, p.4]. The matrix that is a combination of the coefficients matrix and the results vector is called an augmented matrix [9, p.4]. In the row reduction algorithm performed on the augmented matrix, one must replace one linear system with an equivalent system which is easier to solve [9, p.4]. One can either replace one equation by the sum of itself and a multiple of another equation (another row), interchange equations or multiply all terms in one equation with a non-zero constant [9, p.4]. After each operation, the new matrix must be row equivalent to the previous one, otherwise the two matrixes will not have the same solution set [9, pp.6-7]. This must be continued until the augmented matrix is in echelon or reduced echelon form.
A matrix in Echelon form is a matrix where all non-zero rows are above any rows of all zeros, each leading entry is in a column to the right of the leading entry of the row above it and all entries in a column below a leading entry are zero [9, p.13]. A matrix is in reduced echelon form when, additionally to the above-mentioned criteria, the leading entry in each nonzero row is equal to 1 and each leading entry is the only non-zero entry in its column [9, p.13]. The columns containing a leading entry are called a pivot column with the position of the leading entry being the pivot position [9,

p.14]. When the augmented matrix is in reduced echelon form, one can deduce the basic variables from the corresponding pivot columns [9, p.18]. The other variables are called free variables which means that any value can be chosen for it [9, p.18].