Final Report

(to be read with the attached code next to it)

Marten Jakob Stubenrauch – 149896

Thor Møldrup – 127318

Elias Aslaksen – 149880

Marin Mes – 149899

19/12/2021

Foundations of Data Science: Programming and Linear Algebra

Copenhagen Business School

Number of pages: 15 pages

Number of characters: 33.271

Table of Contents

Introduction *3*
Orthogonality – The Fire Hose Problem *3*
Class 1: FireHoseProblem 4
Class 2: StartPage 5
Class 3: PageOne 6
Class 4: PageTwo 7
Class 5: PageThree 8
Calculations 8
Calculations in Python 9
Naïve Bayes Classifiers *10*
Training / Testing 11
Gaussian Naïve Bayes 12
Multinomial Naïve Bayes 13
Bernoulli Naïve Bayes 14
Conclusion *16*
References *17*

Introduction

In this project, we aim to show a combination of our knowledge of linear algebra and Python through two main topics. The first topic is Orthogonality, for which we created a user interface to solve a real-life problem. The second topic is Naïve Bayes Classifiers, where we utilize the statistical and probabilistic foundation from the course to create three different machine learning models by hand: a Bernoulli classifier, a Multinomial classifier, and a Gaussian Classifier. We programmed these without the use of external machine learning libraries such as SciKitLearn. The purpose of this report is to offer, in addition to the comments to be found in the code, explanations about our programs.

Orthogonality – The Fire Hose Problem

We decided on the topic of Orthogonality as we wanted to apply linear algebra in a real-life scenario with a practical application. In class we were introduced to the Fire Hose Problem which describes a situation where a house is on fire at a certain coordinate, and the solver must find out whether the user is able to fight the fire from a certain street with a firehose of a certain length (Figure 1). To do this, the solver is provided with two coordinates, one for the house, and one for the street, and with a firehose of a certain length. Using calculations related to the concept of orthogonality the solver must find out whether the house can be saved (Mukkamala). For our project, we decided to build an interactive interface where Fire Hose Problem game can be played. We expanded the basic problem by adding a third vector representing a second street, leading to the game results being which street is qualified best to fight the fire and whether the fire hose can reach the flames from that street. In the paragraphs below we will describe how we built the user interface and how the code behind it works together.

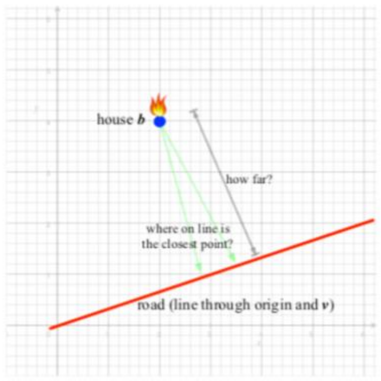
house b
how far?
where on line is the closest point?
road (line through origin and v)

Figure 1: The Fire Hose Problem

To create a user interface, we decided to use the *Tkinter* library. *Tkinter* is the standard GUI library for Python and, in combination with Python, it provides a powerful object-oriented interface. Since we were taught object-oriented programming and since it is relatively simple to create widgets and features using the *Tkinter* library, we felt this was the way to go. So, the first step was to import the *Tkinter* module. Additionally, we needed to import the *messagebox* extension separately to be able to display error messages. Furthermore, in order to plot the graphical representation of the problem the player of the game is facing, we needed to import the *matplotlib.pyplot* library and the *FigureCanvasTkAgg* extension. Since we discussed these modules in the course, we felt like it was suitable to use these in our final project as well. Lastly, we needed to import the *math* and the *numpy* module to do the orthogonal calculations.

Before we created our interface, we defined our *callback* function which validates whether the information the user puts in in the interface is an integer that can be used for the calculations, and not a float, string, or blank spaces. We decided not to allow floats to simplify the code. The reason this function is defined at the beginning, is because it needs to exist outside of the interface for us to be able to call it within the interface. We will come back to this function later.

Class 1: FireHoseProblem

As mentioned earlier, *Tkinter* provides a powerful object-oriented interface. The first class we defined is the class in which we create our interface. To clarify that this is the basis for our interface and that our program is built around this class, we called it 'FireHoseProblem'. Within this class we inherit all properties from *Tkinter* which is why we include it as an argument in the class. The first method we define is the initialization function which will be called every time we create a new instance of the class, i.e., every time we run our code. The initialization naturally passes the 'self' argument, and we decided to include '*args' and '**kwargs' as well. Including these arguments will allow us to pass through any variables (**args*) and any dictionaries (***kwargs*) we want. Within our initialization we made sure to initialize *Tkinter* again and pass '*args' and '**kwargs' as arguments, to guarantee that every feature is loaded and ready to use when running the program.

The next step in our initialization is to define the container where our game will be displayed, and which will contain all the following pages/features/widgets that we code. We created this container by using *tk.Frame* and we named it the 'fire_hose_problem'. After we created it, we *packed* the frame since this is necessary for it to be visible. This holds for every feature of the *Tkinter* library and thus throughout the code, you will see that all widgets and elements are being packed after their creation. While *packing* the features, it is possible to instruct the program where exactly in the frame you want to pack it and how you want it to show on the screen. The arguments we used for this in the *pack* functionality are *side, fill,* and *expand*.

Next, we created an empty dictionary for all the pages in the game. We want this dictionary to be created every time we play the game, so it is part of the initiation. The dictionary is empty since we do not want all pages to be called immediately when we start the game as this will cause errors. Therefore, we created a *for* loop matches the name of the page (value) with the correct key and after enables the correct page to be showed whenever we press a button. The driving force behind this is the *show_frame* function that is called in the last line of the *for* loop. This function raises the page that we call to the top of our container of pages, thus showing the desired page. It does this by setting the *controller* as the key in the dictionary *self.frames* and afterwards displays the correct value in our frame. The function *show_frame* itself does not run within the initialization since we only want to call different pages whenever the user asks for that in the game.

Lastly, we define our *reset_game* and *close*_window function. The *reset*_game function allows the user to restart the game. When it is called, the function destroys the interface, after which it immediately calls it again, thus allowing for a restart. The *close_window* function allows the user to terminate the game whenever they want to. This function is almost identical to the *reset_game* function, except it does not call the interface again after it has been destroyed. There is no need to initialize these functions since we do not need them to be called every time we start our game. We define these methods in the parent class as we want to be able to call these functions from any other class.

Class 2: StartPage

After creating our baseline, we were ready to create different pages. For every page we created a separate class, all inheriting from *Tkinter* and everything from our baseline frame (our backend named 'fire_hose_problem'). Our first page is the 'StartPage'. On this page we explain the game and explain which data the player needs to play the game. We display this text (and all other text messages in the interface) using *Tkinter's* label widget which responds to *tk.Label.* Within this widget one writes the text in the *text* argument, the font in the *font* argument, and one determines the color in the *bg (*for 'background') and *fg (*for 'foreground') argument.

Additionally, we wanted the user to be able to choose between two buttons: one for starting the game and one for quitting the game. These buttons are created with *Tkinter's* button widget, which responds to *tk.Button.* Within this widget one can display the text of the button (*text* argument), set the color of the button (*bg* and *fg* argument) and finally, one can choose the *command* the button needs to execute. These commands can be, and in our case will be, functions defined in the current or parent class. If the user decides to 'Play the Game', the *show_frame* function from the parent class will be called and the user will be directed to the page specified in the command, in this case 'PageOne'. Consequently the 'PageOne' class is called and executed. If the user decides to 'Exit the game', the *close_window* function is called, and the game will be ended. For the commands to be executed only when we clicked the button (and not immediately when running the class), we had to include 'lambda:' before writing the buttons command function. We did this for all buttons in our interface. All widgets in this page (and most in the rest of our code) are initialized since we want them to be visible and functioning whenever we call the page.

Class 3: PageOne

The next class represents a page where the user inputs the information the game needs. To be able to accept user input, we used *Tkinter's* entry box widget, which responds to *tk.Entry*. This widget shows an entry box where the user can type in information. We validated the input of the user by creating a variable called 'reg' which checks user input. Next, we configured each entry field, and tell the program to validate the input of the user using the 'reg' variable. These configurations take various arguments: the *validate* argument is used to specify when the *callback* function (defined at the beginning of the code) will be called to validate the input. The "key" value specifies that validation occurs whenever any input from keyboard changes the widget's contents. The *validatecommand* argument is used to tell the program to use the 'reg' variable while validating, and '%P' is passed to denote the value that the text will have if the change is allowed. For the entry boxes it was necessary to put the name of the class, *PageOne*, in front, so we would be able to use the value of these inputs in another class.

PageOne displays four buttons. The first button is a 'continue' button which will bring the user to the next page. However, before it brings the user to the next page, it will do another check on the input of the user by running the *validate_input* function. This function checks whether all boxes are filled, and if not, will prompt error messages. Additionally, it will check whether the streets are useable to fight the fire. For instance, when a street does not reach the same x-coordinate as the house, the game will assume that the street is not suitable to fight the fire and the user will be prompted to use another street. Lastly, whenever the house lays on one of the streets themselves, the user receives the message that that street is best to fight the fire from. Once all checks have been done, the user will be taken to the next page, PageTwo. It is not needed to initialize the *validate_input* function since it should activate whenever the user wants to go to the next page.

The second button is the 'reset' button. Whenever the user clicks this button, all input will be deleted from the entry boxes and the user can fill in new numbers without having to delete everything manually. To do this, we command the button to execute the *delete* function, which is also defined after the initialization of the class. The third and fourth button are the 'Start Over' and 'Reset Game' button. The former calls the *reset_game* function (defined in the parent class) and the latter calls the *close_window* function (also defined in the parent class).

Class 4: PageTwo

The next class represents a page where the user is offered certain options to choose from in the form of different buttons. The first button allows the user to see the graphical representation of the information it filled in the entry boxes. To show the graph, the button calls the *orthogonal_projection* function, which is defined after the initialization of the class, and calculates the coordinates of the orthogonal projection of the house on the two streets. For the function to work, we transformed the data type of the input to floats and used the *.get()* extension to use the value of the input. We then transformed the inputs into variables to avoid long lines in the calculations. After executing the calculations, we created lists of the streets' coordinates in order to plot a visual representation. It must be noted that we added the origin as a default coordinate for both streets. To visualize how the house was connected to its orthogonal projection, we decided to include a dotted line from the house to its two orthogonal projections. Additionally, we added a legend, so the user knows what they are looking at. To get our *matplotlib* figure in our canvas, and not as a separate pop-up window, we used the *FigureCanvasTkAgg* method and the *get_tk_widget.* The last part of the *orthogonal_projection* function is the removal of the 'Show Graph' button since we do not want this button to exist whenever the graph is already being shown. This is done by using the *destroy* feature of the button widget. The second button will guide the user to the next page, PageThree, where it is able to see the results of the game. The third and fourth button are the 'Start Over' and 'Reset' button, which work the same as in the previous classes.

Class 5: PageThree

This next, and final class displays the end results of the game, revealing what street is best positioned to fight the fire, and whether the fire hose can reach the flames. We initialize two labels which describe what the user will see when clicking the 'show' buttons, which we also initialize. It was necessary to put 'PageThree' in front of these widgets so we could use them in our calculations. Both 'show' buttons call a function for calculating the distance from the house to its corresponding orthogonal projections. Since we have two streets, we made two functions, *distance_street_1* and *distance_street_2*. Just like before, we transformed the data type of the input to floats, used the *.get()* extension, and created separate variables for the input to avoid long texts in the calculations. At the end of both functions, we configured the texts of our labels to now show the calculated distance, and not the previous text anymore, and, naturally, destroyed the show buttons. In *distance_street_2*, we added the calculation of which street is more strategic to use and whether the fire hose is long enough or not. Lastly, we included the 'start over' and 'exit game' button, just as a 'back' button allowing the user to go back to the graphical representation page. The last three lines of our code are that where we define our app, give it a title, and call the *mainloop*. The *mainloop* is called at the very end since it will block all code after it.

Calculations

The calculations in our code are those to find the orthogonal projections and the distance between the house and these orthogonal projections. We will explain how we did these calculations below using the example coordinates from the lecture.

- V1 is represented as the vector of the first street.
- V2 is represented as the burning house.

Let $V1 = [6, 2]$ and $V2 = [2, 4]$

$$V1 * V2 = 20$$

$$V1 * V1 = 40$$

Calculating the closest point to the house (*A-hat)*, or the orthogonal projection of V2 on V1:

$\frac{20}{40} = 1/2[6, 2] = [3, 1]$ = *A-hat.*

We have now calculated the coordinates of the orthogonal projection of the burning house on the street, and we call it *A-hat*.

Next up we calculate the distance to the burning house:

$$[2,4] - [3,1] = \left|\left|[-1,3]^2\right|\right| = \sqrt{10} = 3.162277.$$

So, 3.16 is the shortest distance from the burning house to the street. Depending on the length of our fire hose the house can be saved or not.

Calculations in Python

To execute the calculations, we had to import Python's *math* and *numpy* library. We needed the *math* library to be able to calculate the square root to get to the distance, and we needed *numpy* so we could use arrays and do multiple calculations in one go. As mentioned in the paragraphs above, we created variables from the user input in our calculations. Below we describe these variables.

A1 = x-coordinate of the house
A2 = y-coordinate of the house
B1 = x-coordinate of the first street
B2 = y-coordinate of the first street
C1 = x-coordinate of the second street
C2 = y-coordinate of the second street
C3 = length of the fire hose

y_u1 = vector street 1 * vector house
u_u1 = vector street 1 * vector street 1
z1 = multiplier
y_hat1 = a list of the coordinates of the orthogonal projection on street 1

y_u2 = vector street 2 * vector house
u_u2 = vector street 2 * vector street 2
z2 = multiplier

y_hat2 = a list of the coordinates of the orthogonal projection on street 2

d = subtracting the x coordinate of the orthogonal projection from the x-coordinate of the house and the y coordinate of the orthogonal projection from the y-coordinate of the house.
d_2 = squaring the coordinates of *d.*
d_3 = adding the two squared results to each other
distance_street_1 = the final distance from the house to the first street.

w = subtracting the x coordinate of the orthogonal projection from the x-coordinate of the house and the y coordinate of the orthogonal projection from the y-coordinate of the house.
w_2 = squaring the coordinates of *w.*
w_3 = adding the two squared results to each other
distance_street_2 = the final distance from the house to the second street.

We perform the calculations following the formula for orthogonal projection and calculating the distance as explained above. The comments in the code illustrate what is calculated where.

In the *if*-statements we determine which street is located most strategically to fight the fire, i.e., which distance is the shortest, and following that we check whether our fire hose is long enough to cover that distance and, subsequently, whether the house can be saved.

Naïve Bayes Classifiers

The Naïve Bayes Classifiers are a group of classifier algorithms, that are all based on Bayes Theorem. All algorithms share the same fundamental that all features are assumed independent of each other (Kumar, 2021). This means that when we have a value for a first feature in a dataset, this would not affect the values of the other features in any way. Of course, this will be difficult to achieve in real life because most datasets have features that are somewhat dependent on each other. The independence of each feature also assumes that each feature has an independent and equal contribution to the outcome, which also might not make sense in the real world. Take for example a fictive dataset containing data on whether people passed or failed a test. The features are if they are *Confident*, if they are *Sick* and if they have *Studied* for the test. Assuming independence would imply that the fact that they are confident would not be correlated with whether they studied or not, and that all features have an equal contribution to the outcome. This would mean that it would affect the result equally if they feel confident or if they studied. Here, it would however make sense that whether someone studies weigh most. Hence, the classifiers are named ‘Naïve’. The independence and equally traits of the Naïve Bayes work however quite well in practice, making Naïve Bayes classifiers some of the worlds’ most used and easiest to implement, in particular for predicting the occurrence of an event given another event.

Naïve Bayes is widely used in Natural Language Processing or simple prediction as it has the strength of only needing a small number of training data required to estimate the occurrence of an event given another event. The basic Naïve Bayes formula to calculate the conditional probability is as follows:

$$posterior = \frac{prior * likelihood}{evidence}$$

$$P(C_k|x) = \frac{P(C_k) * (P|C_k)}{P(x)}$$

Formula 1: Naïve Bayes

The prior probability is the general probability of an event occurring which is based on the training data. The likelihood is the conditional probability of one of the features in the training data set. These features could be words in the email that we want to predict to be either spam or normal. The evidence is then the probability of the feature to occur. In our example it would be the probability of a certain word to appear across all emails.

Training / Testing

To be able to use a model to predict, it needs to be trained. This means that we use some of our dataset for calculating all the different probabilities, allowing the model to make predictions based on this. It is important that the model sees the correct answers, so it can classify the different instances better. After the model has been trained on the training dataset, it is then ready to be tested on the testing data, making predictions for the target variable based on the value of the features. A common division is to use 70% of the data as training data and 30% for testing. To measure the accuracy of the model one compares the predictions that the model made with the actual values of our target variable. Then the number of times that the model guessed correctly is divided by the total amount of times that it guessed. This results in a percentage reflecting the accuracy of the model for the given dataset.

Gaussian Naïve Bayes

When one wants to predict an event given features that are continues variables, the Gaussian Naïve Bayes can be used. Through the assumption that the features are based on a normal distribution it is possible to calculate the likelihoods of the features. After calculating the likelihoods across all the different features one can multiply that with the prior of a certain class to receive the posterior probability of the class given the set of features. The next step is then to compare the posteriors of each class and return the class with the highest posterior. We test our algorithm based on a the widely used iris data set which collected data on three different iris flowers on four features. As the features are continuous it is as a good testing dataset for our algorithm.

To create the Gaussian Naïve Bayes algorithm in Python, we must import *numpy* and *pandas* to support the calculations and provide a way to read in external data sets to test our algorithms. We first create a class *GaussianNaiveBayes* that will be initiated when creating an instance of the algorithm and holds four other methods. Two are the main functions open to be performed on either training data, namely, *fit* and *predict*. The third and fourth functions are helper functions: *_predict* and *_likelihood*. These will perform the calculations needed to get the posteriors for the testing data. Furthermore, to avoid extremely small numbers that could arise in these calculations we simplify the calculation by first removing the denominator as it is the same for each feature. Next, we take the logarithm of the probability and instead of multiplying the probabilities we add the probabilities together.

$$y = argmax_y \, P(y|X) = argmax_y \frac{P(x_1|y)*P(x_2|y)*P(x_n|y)*P(y)}{P(X)})$$

$$y = argmax_y \, P(y|X) = argmax_y \, P(x_1|y) * P(x_2|y) * P(x_n|y) * P(y)$$

$$y = argmax_y \log(P(x_1|y)) + \log(P(x_2|y)) + \log(P(x_n|y)) + \log(P(y))$$

Formula 2: Simplification of the Naïve Bayes algorithm y = prior, x = n features

The *fit* function takes in the training data as *X* and the labels of the data as *y*. The purpose of this function is to calculate the prior probabilities of the unique label (classes) and to prepare the program for the prediction of the unknown data. Next to calculating the prior probability we also calculate the mean and the variance of the different features. We need these values when calculating the likelihood of the different features, for which we will use the Gaussian function. Using the iris data set, the fit function calculates the mean and the variance across the difference features for each of the unique flowers. In the end it creates a 3 by 4 matrix where the cells show the respective mean and variance of the class and the given feature.

The *predict* function provides the most likely class for the given features. Within this function we loop through all the samples of the test data and call our helper function *_predict* on each of the rows. First, the function creates an empty list to collect all the posteriors for that data row after which it loops through the different classes. This is to create a posterior probability for each of the classes which is needed later to compare and select. In the loop through the classes, we select the prior probability corresponding to the class and afterwards calculate the likelihood for each of the features. By calling the likelihood function, using the Gaussian function, and summing the results, we get the likelihoods from the different features given the class. The last step is to add the prior probability and the likelihood to calculate the posterior values.

$$P(x_1|y) = \frac{1}{\sqrt{2\pi\sigma^2}} * \exp\left(-\left(\frac{\left(x_i - \mu_y\right)^2}{2\sigma_y^2}\right)\right)$$

Formula 3: Class conditional probability with the Gaussian Distribution Function

Throughout the code we implement error handling and input validation to ensure that the user provided the right inputs, and that the algorithm can predict the data correctly. First, we check if the input data can be converted to a *numpy* array which is necessary for our calculations. Second, we include a check when calculating the mean and the variance of the different features given the class. At this step of the calculation, we can safely check if the input are numbers and not strings and validate that the further calculations will not be in jeopardy. Lastly, we assume that the user will know what type of data they will input. Thus ultimately, the user only needs to initiate the *GaussianNaiveBayes* class, train via the *fit* function with continues data and then via the *predict* function calculate the predictions for the input data.

Multinomial Naïve Bayes

When one wants to predict an event given features that are discrete variables, the Multinomial Naïve Bayes can be used. Through assuming that the user inputs data that can be counted, such as words in a sentence, we calculate the conditional probability of receiving a certain kind of message based on its content. After calculating the likelihoods across all the different words (features) one can multiply that with the prior of the certain class to receive the posterior probability of that class given the set of features. Like the Gaussian Naïve Bayes, after calculating the posterior probability across the different classes, we can take the maximum posterior to determine which class has the highest probability given the features. We test our algorithm based on a dataset of made-up emails with different words and a label determining whether it is spam or not.

We will implement the Multinomial Naïve Bayes algorithm in Python by first importing *numpy* and *pandas* to support the calculations and provide a way to read external datasets to test our algorithms. We first create a class *MultinomialNaiveBayes* that will be initiated when creating an instance of the algorithm and which will hold four functions. Just as in the Gaussian Naïve Bayes, we have *fit*, *predict*, *_predict* and *_likelihood*. Furthermore, to avoid extremely small numbers that could arise in these calculations, we simplify the calculation by cancelling out the denominator (Formula 2). Following this structure simplifies the algorithm to a degree where it makes sense, and it still resembles the original formula of Naïve Bayes.

The main functions of the *MulitnomialNaiveBayes* are very similar to the functionalities that we implemented in the Gaussian Naïve Bayes. However, as the algorithm is aimed towards analyzing text data and predicting based on the probability that the words appear in a class, the algorithm also includes a data cleaning mechanism and simplified word frequency tracking. In both functions *fit* and *predict,* the algorithm cleans the email content of all is non-alphanumeric letters and deletes words that are stop words. These stop words have little meaning are thus redundant to keep track of.

The second unique addition to the *MultinonminalNaiveBayes* is the creation of a frequency table. When the user is initiating the class and uses the *fit* function, the algorithm creates a frequency table with the unique classes e.g., 'spam' and 'normal', and the columns as unique words. This table will then be filled with the different counts of words in each of the classes. Later in the *predict* function, the algorithm then accesses this table to calculate the given likelihood of the word performed by the Naïve Bayes prediction. If the word in the testing data does not appear, the algorithm counts it as a new one every time instead of ignoring it. Thus, ultimately the user can initiate the functionality and use the algorithm to test if their text data matches to certain class.

Bernoulli Naïve Bayes

When one wants to predict an event given features that binary, the Bernoulli Naïve Bayes can be used. For example, we look at the dataset that describes if the student passed or failed a course. The attributes of this dataset are Confident, Sick, and Studied, and the target variable is Result. All possible values are either 0 or 1, which equals Yes or No, or True or False respectively. The general idea is to predict the results depending on the values of the different attributes.

The first function in the code is the *prep_data*. This is used to split our dataset into the attributes and the target column. This is necessary to make the further analysis easier, when having to calculate probabilities for the Bernoulli Naïve Bayes.

Next the program loads a dataset (the dataset that to be used for this program is the "SimpleBernoulliData.csv") where all the attributes as well as the target are binary (we validate whether all the columns in the dataset are binary by checking the unique number of values in each column and if it is more than two, then we raise an exception). The used dataset is randomly generated, which makes the real-life use limited, but it gives a way to use the program. The dataset is then split into training and test dataset with a test size of 30%. Both the training data and the testing data are then passed through the *prep_data* function to split them into *X* and *y*. Then the *BernoulliNaiveBayes* class is initialized, and the training data is fitted on the model.

Initially in the *BernoulliNaiveBayes* class, variables such as a list to store the attributes, a dictionary to store the likelihoods, a dictionary to store the class probabilities and a dictionary to store the prior probability of the predictors are created. To be able to store our values in the different dictionaries that we have created, it is necessary to initialize them and input 0 for the values. This is done for every attribute and target, which makes us able to store values and calculate probabilities for the attributes. Then the class probability of the dataset is calculated with the function *_calc_class_prob*. This is the total probability of the result being either 0 or 1 for the whole dataset. This is calculated by finding the total amount of 0's in the target variable and dividing it by the total amount of rows in the dataset. The same is done for the total amount of 1's. This gives us the total probability of the two possible values for the target value.

Next up is to calculate the likelihood, also known as the probability of predictor. This is calculated for every possible combination of attribute and target. For example, we calculate the possibility of the first attribute being 0, while the result is 0, the first attribute being 1, while the result is 0 and so on. Then the total probability of an attribute being 0 or 1 is calculated and stored in a dictionary.

After the dictionaries have been initialized and the model has been fitted, it is time to make predictions from the testing data. The predict function takes *X* as an input, which is then turned into an array. The *X* that we are inputting is the row from the *X_test* data, that we want to make predictions on. First off, we loop through the unique values for the target variable and then retrieve the total class probability of the given value. Then we calculate the likelihood and evidence for each attribute in the testing dataset, which is also called the posterior for each row. The posterior is calculated using

$posterior = \frac{likelihood * prior}{evidence}$. The values that we get for the different results e.g., 0 and 1, are then stored in the dictionary. Finally, the largest probability is returned which then represents the predicted value. So, if the probability of a given row having the result being 0 as the highest probability, we will then predict that the given row has a result of 0. Lastly, an accuracy score is calculated by dividing the total number of correct guesses of our model by the total number of guesses.

Conclusion

Throughout this report we laid out our process of creating two programs with high realistic relevance and areas of applications. Through the Fire House Problem, we learned how to use linear algebra to find the shortest path between vectors and package it in a python program to create an interactive game. For the second program we coded the three different Naïve Bayes algorithms from scratch to apply probability theory and linear algebra to calculate if e.g., an email is spam or normal. Throughout this process we learned about linear algebra in conjunction with the Python language and how certain algorithms work the way they do. During the process of programming, we used several programming communities such as StackOverflow and YouTube to guide us on the right path.

References

Kumar, N. (2021, December 12). *Naive Bayes classifiers*. GeeksforGeeks. Retrieved December 19, 2021, from https://www.geeksforgeeks.org/naive-bayes-classifiers/

Mukkamala, Raghava. “Eigen Vectors and Inner Product.” Foundations of Data Science: Programming and Linear Algebra, 01.12.2021, Copenhagen Business School. Lecture.