Architecture Document

Client: Zeehondencentrum Pieterburen

ANDREI MIHALACHI *s3491862*
ANNIKA MÖLLER *s3585832*
BRIAN DE JAGER *s4104579*
MIKE LUCAS *s3519120*
ROAN ROSEMA *s4109449*

Lecturer: Dr. Andrea Capiluppi
Teaching Assistant: C.M. Rosiu
June 14, 2021

Contents

1 Introduction

The vets who work with the rescued seals at the Zeehondencentrum Pieterburen use several methods to assess the health of the seals once they have been brought to the sanctuary. The goal of this application is to create a tool that will make use of artificial intelligence technology to aid them with this process. Wavealyze is a Windows application that uses two convolutional neural networks (CNN) to analyze the health level of a seal based on auscultation recordings taken by the vets. Auscultation recordings are typically recordings of the heart or lungs of an animal as it breathes - in our case we use recordings of the lungs. The user can upload an auscultation recording, which is analysed by two CNNs. The first model outputs a binary yes/no value indicating whether or not there is a whistle in the seal's lungs, and the second model indicates the severity level of the rhonchus in the seal's lungs, if it is present.

In order to keep the design cohesive and straightforward, we use a table format for the main screen of our application. Here the user can upload sound files via a button in a menu bar. The table has further functionalities that allow the vets to enter additional information about the seals, as well as edit the table. These will be discussed further on in this document.

A requirement from our client was that the data generated by the CNN should be portable. As such, the application will allow the user to download the generated table as a .csv file, as well as upload previously generated .csv files to the application window and continue editing it.

Throughout this document we will refer to features. The feature matrix used can be found at the end of the document in the Appendix.

PIETERBUREN
ZEEHONDENCENTRUM

2 Architectural Overview

2.1 General Overview of Components

The application contains the following components:

- A graphical user interface implemented using PyQt5, which functions as the front end of the application.
- The CNN that analyzes the sound files and outputs a health score. This was developed by our TA who worked on it in parallel. Our team did not personally have any influence on the model's design and implementation.
- An InputHandler module, which handles all input from the user into the program. This input is either the sound files or a .csv file.
- An OutputHandler module, which handles the exporting of a table to a .csv file.
- A BackEnd module, which stores and handles the editing of data in the current session. We also use it for implementing our auto-save functionality, as well as saving relevant details from the current session to be used upon reopening the application.

2.2 Conceptual view

To gain an understanding of how these different components interact with each other, we have designed a conceptual model as seen in figure 1. The user can interact with the system using the User Interface. They can either upload a .csv file to the InputHandler, which directly updates the back end, or they can upload X number of .wav audio files to InputHandler. In case of a .wav file, the InputHandler parses each uploaded audio file to the correct format. For each audio file, the CNN model generates a result that is sent back to the InputHandler. Together with the file name of the uploaded audio file, the InputHandler updates the back end accordingly.

The user will also have the option to to download the data as a .csv file. The Backend will send its data to the OutputHandler, where it is then converted to the .csv format. The OutputHandler then sends the .csv file back to the user, with all of the data that is currently in the Backend.

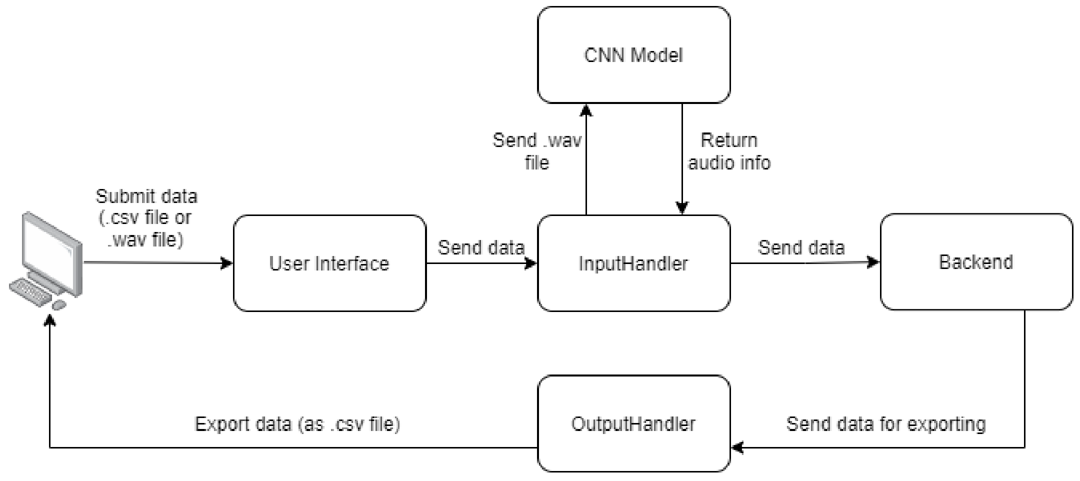


Figure 1: Conceptual Model

2.3 Model-View-Controller Pattern

In order to organize all these components, we chose to use the model-view-controller pattern. We made this choice because there was a need for separation between the data itself, how it would be presented to the user, and how the user would interact with it. Overall, we felt that it was the best fit. To easily explain this concept, we can use the following figure. The full details of the application architecture will also be explained in the context of the MVC pattern.

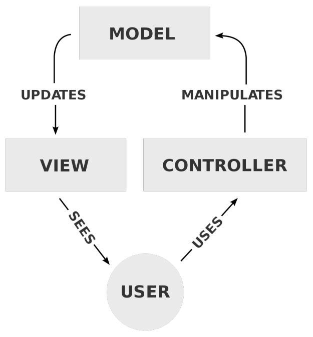


Figure 2: Model-View-Controller Pattern

As we can see in figure 2 the user uses the controller and sees the view. Using buttons from the controller, the user can alter the internal state of the application which is represented by the model. Once the data within the model is manipulated by the user using one of the buttons, the model sends the view a message to update. Now the user can see the result of their action.

2.4 Model-View-Controller Implementation

A bird's eye perspective can be found in figure 4 that shows a detailed graph that represents the connections between the various parts of our MVC pattern. Every square that is not colored represents a package within the program. Hence, the main three packages are the Model, the View and the Controller, which is in line with the MVC pattern. Within the main 3 packages, various other packages are made to further split up important parts of the code. Within this layer of packages are colored squares, these represent different kinds of models. In the legend in figure 3 can be seen that there are four different squares and 3 different arrows. These squares all represent different modules of different file types. The arrows represent different types of relationships between the modules or packages.

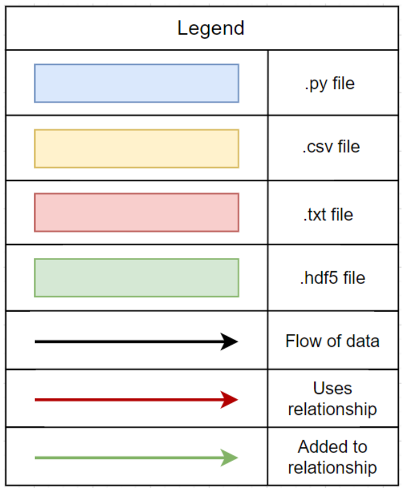


Figure 3: Legend

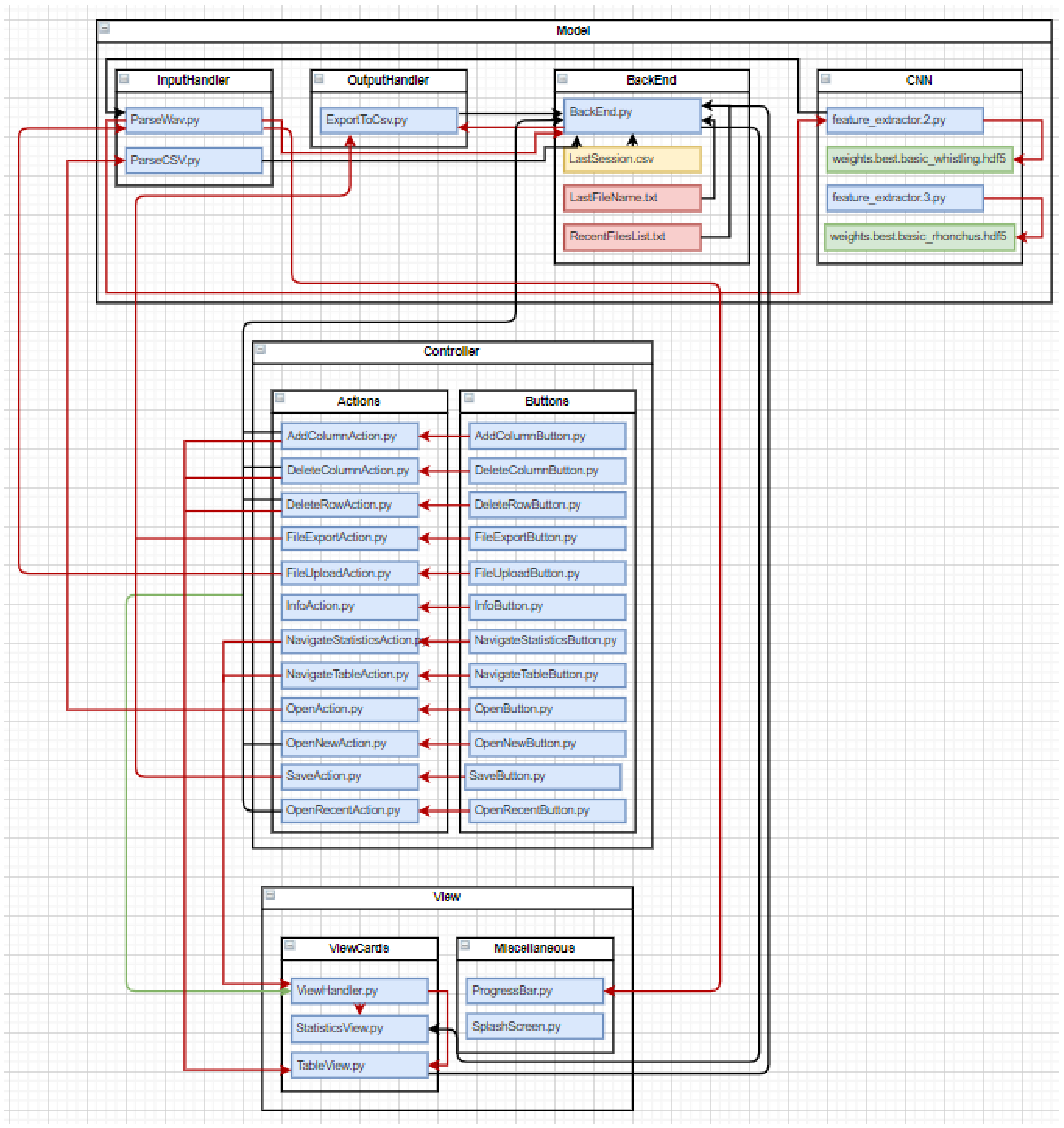


Figure 4: MVC implementation overview

2.4.1 Model

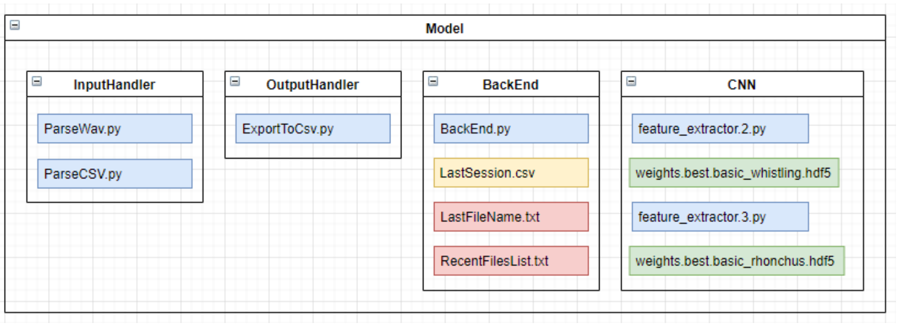


Figure 5: Overview of the 'Model' package

1. CNN

When an audio file has been uploaded, the `InputHandler` will send this file path to a method where the pre-processing of the audio is done. (Feature 1)

The CNNs require the input sound files to be in a specific format - i.e an array with the first 40 Mel Cepstral coefficients. These are a set of features that concisely describe the overall shape of a spectral envelope. All of this is done with the `librosa` package in Python.

The Mel Cepstral Coefficients array is then reformatted such that it follows the CNN input format. This array is then put into both models, where it will return the corresponding health labels for the seal. These labels are sent to the `InputHandler`, where they are further handled in conjunction with the respective audio file.

2. InputHandler

Inside of the `InputHandler` package, we have two files: `ParseCSV` and `ParseWav`. These handle the input of .csv (previously downloaded tables) and .wav (auscultation recordings) files respectively. (Feature 1, 6)

The Input Handler is used to ensure that the audio files are processed correctly and that the correct files are sent with the corresponding CNN-output to the data storage. Inside the program, the user can decide whether they want to upload a sound file, multiple sound files, or a .csv file. When selecting to upload a sound file, the user can select a file or multiple files, after which the files (file paths) are sent to `ParseWav`. Here, the audio files are sent to the CNN models, which return a value for each file separately. The files and the corresponding values are then sent to `BackEnd`.

The following sequence diagram in figure 6 shows what is done after the uploading of .wav audio files in our program (per UC1 in the requirements document):

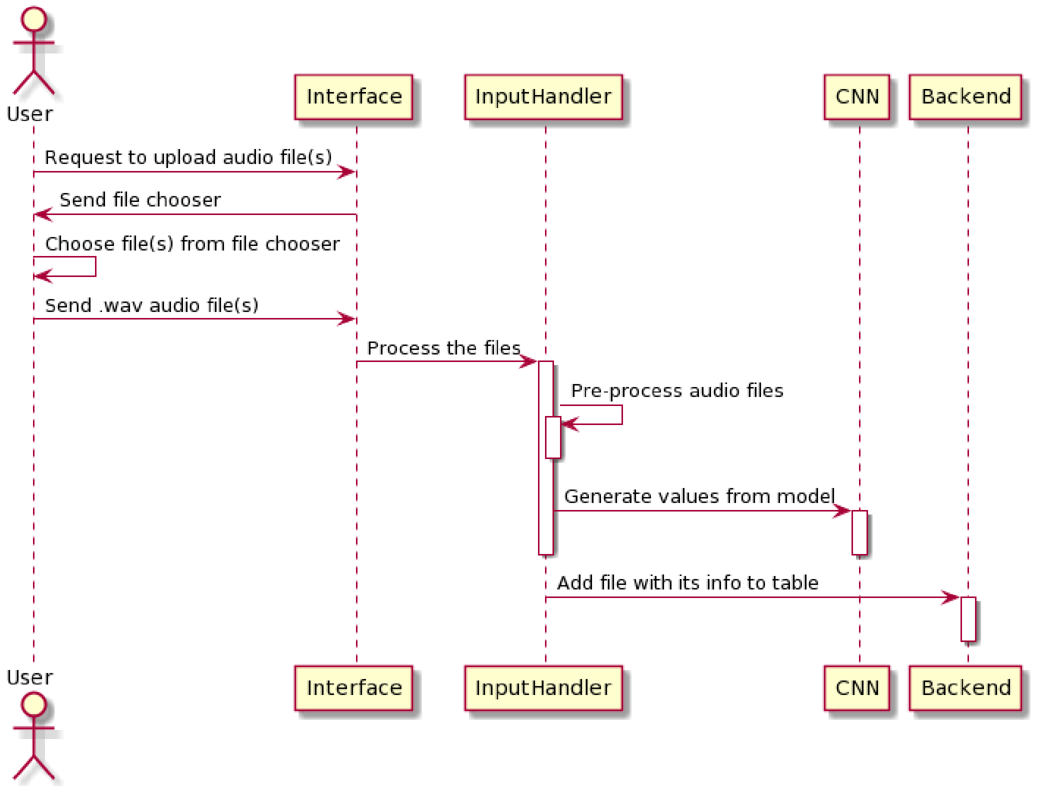


Figure 6: Use Case 1: Upload a .wav audio file

When selecting to upload a .csv file, the user can select a single .csv file. This file is then sent to `ParseCSV`, which parses the data to a pandas data frame that is used to update `BackEnd`. As seen in figure 7, this is done as follows (per UC3 in the requirements document): (Feature 6)

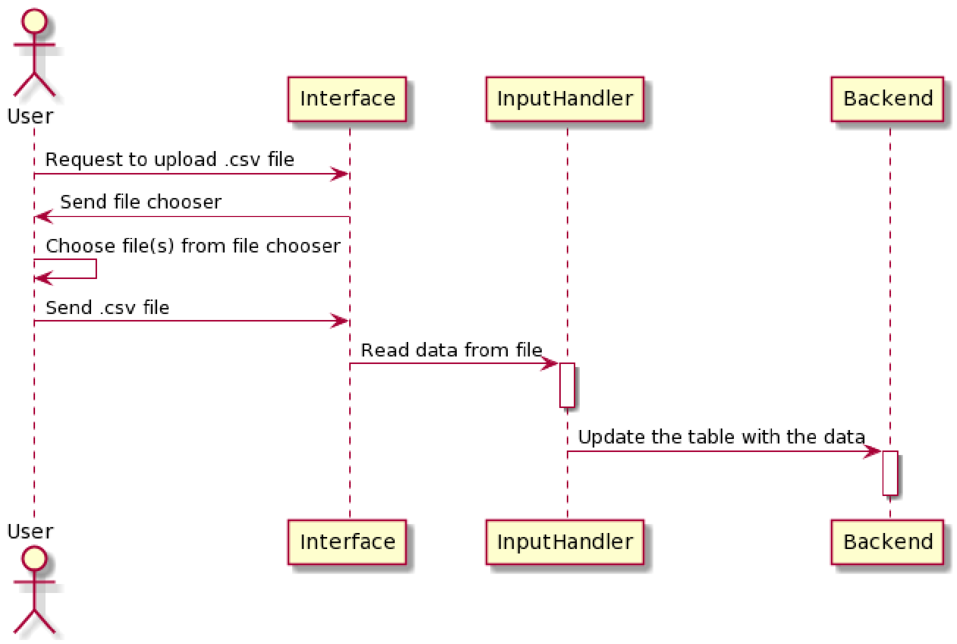


Figure 7: Use Case 3: Upload a .csv file

3. OutputHandler

The `OutputHandler` package has one file: `ExportToCSV`. Here the data stored in `BackEnd` is converted from a pandas data frame to a .csv file. This is done with a single built-in pandas function call. Shown in figure 8, as per UC2 in the requirements document. (Feature 5)

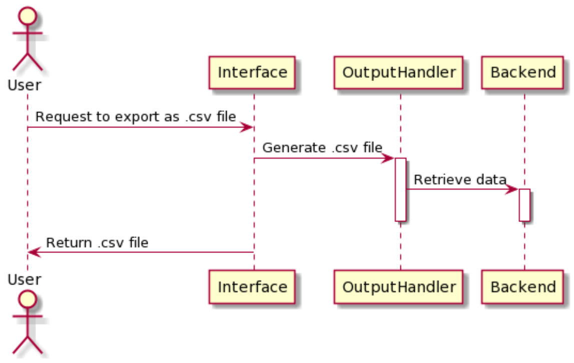


Figure 8: Use Case 2: Export a .csv file

4. BackEnd

Storing the data in the table The back end of our application is a single class which functions as a mini database. This class holds a pandas data frame, which stores all the data available in the application. The idea behind this is that data which is in the back end is shown to the user via the front end (graphical user interface). We chose a data frame to store all the data because its structure corresponded with the layout of the table neatly in the sense that pandas also uses columns and rows.

Auto-save Functionality When the application is loaded, the back end attempts to recreate the latest session by a user. This last session is stored in a separate .csv file as a part of our program, and it is written to every 20 seconds by an auto-save function. When the program is first loaded, it checks in `LastSession.csv` whether there is data in the file. If there is data in it, it is loaded automatically. If not, an empty table screen with the default columns is loaded. (Feature 14)

Uploading a sound file - update() function When the user uploads a sound file (this is done in the `InputHandler` module), the data from this upload is sent to the BackEnd class via the `update` function. This data includes the file name and the model's result, for example:
`PV18218_031218_L.wav` or `PV18218_031218_R.wav`
The `L` or `R` indicates whether the sound comes from the left or the right lung of the seal, and since we output the score for each lung, we split the file name into the seal ID (`PV18218_031218`) and the side (`L` or `R`). Alongside this, the result of the CNN (`1` or `0`) is passed to the `update()` function. The seal ID is added to the table and the score for the lung is written to the corresponding 'Left' or 'Right' column. If the seal ID already exists in the table, we retrieve the index and write the score to the corresponding column of that index. After this, the tableView's `populateTable()` function is called so that the view is updated with this new information. (Feature 1, 6)

Writing to and reading from storage files In our program, we decided to implement an `Open Recent` menu, which shows a drop down menu of all the files a user has recently edited. The paths of these files have to be saved for the next time the program is opened, so they are written to a text file called `RecentFilesList.txt` by the function `writeRecentFiles()`. When the program is started, this file is read from by `readRecentFiles()`. The same concept is implemented for the path of the last file that the user had open in the application, since this is displayed at the top of the application window. For this we use `LastFileName.txt`, `writeLastFileName()`, and `readLastFileName()`. (Feature 5, 6)

Updating Recent Files list The list of recent files mentioned in the previous section is stored in the `BackEnd` class. Whenever a user opens or creates a new file (this is handled elsewhere in the program), the `updateRecentFiles()` function is called. This adds the new file path to the list, and drops the previous one such that only the 5 most recent files are kept in the list. (Feature 7)

Editing the table - refresh() function The user has the ability to add a new column, delete columns, and delete rows. Whenever this is done (elsewhere in the program), the `refresh()` function is called. This updates the `BackEnd` with the new data. The following diagrams show how columns are added (Figure 9 as per UC4 in the requirements document), and how columns/rows are deleted (Figure 10 as per UC5 in the requirements document). (Feature 12)

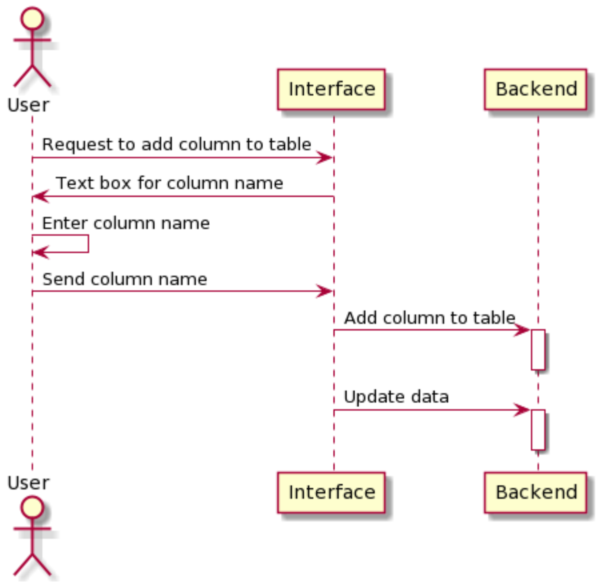


Figure 9: Use Case 4: Add a column to table/data

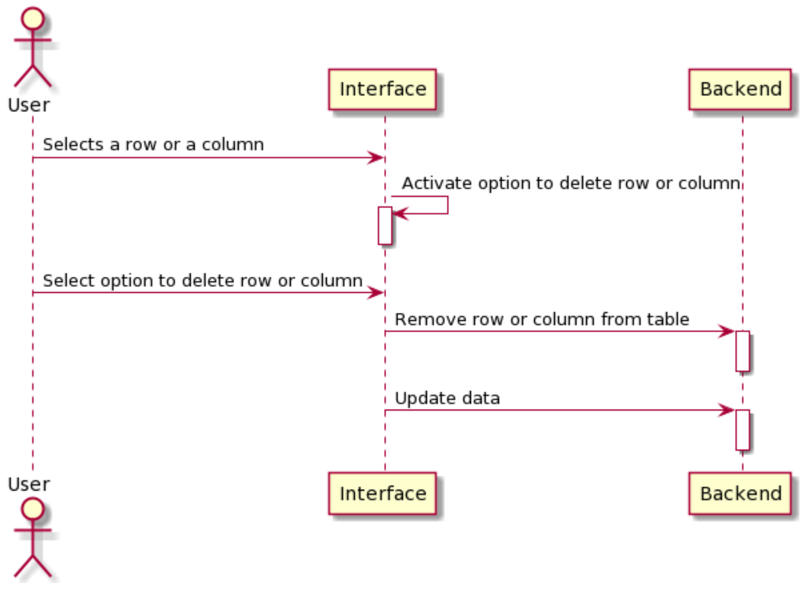

User
Interface
Backend
Selects a row or a column
Activate option to delete row or column
Select option to delete row or column
Remove row or column from table
Update data
User
Interface
Backend

Figure 10: Use Case 5: Delete a column or row from table/data

2.4.2 View

Inside the View module is where all the components of the program that are to do with the look of the application are handled.

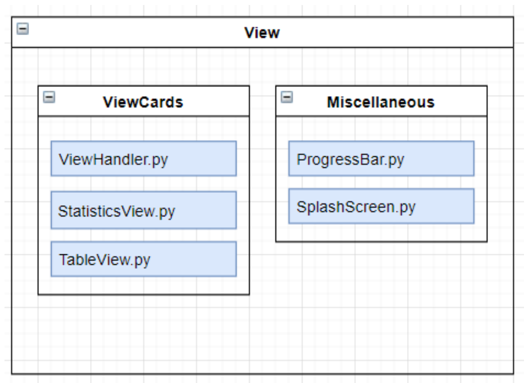


Figure 11: Overview of the 'View' module

1. ViewHandler

The `ViewHandler` class represents the main application window and acts as a canvas on which the different views are painted. It inherits from the PyQt5 QMainWindow class. It contains a `TableView` object, a `StatisticsView` object, and a menu bar.

Menu Bar The function `initializeMenuBar` creates the menu and adds the buttons (stored in the `Controller` section) to this menu. The function `initializeActions()` connects each button's action (also stored in the `Controller` section) to the buttons. The following diagrams show how the *Menu Bar* is split up. Figure 12 shows the layout of the menubar and its components.

- In the *Info* section, the general idea and usage of the application is explained.

The other sections each have a drop-down menu of various buttons. The following three diagrams explain each section separately:

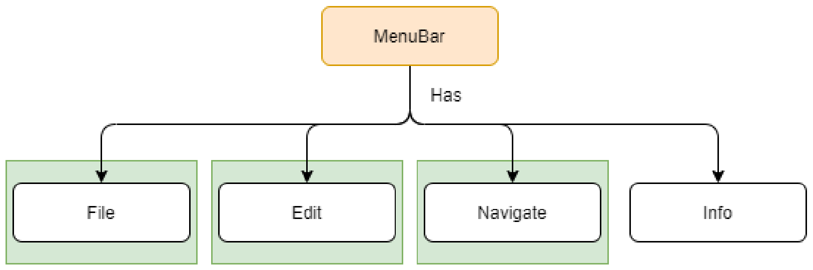


Figure 12: Menu bar layout

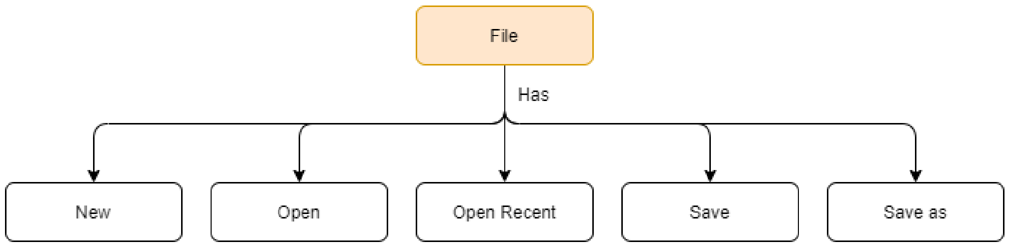


Figure 13: File menu layout

The *File* section, as shown in figure 13, contains all necessary buttons that work with the .csv files of the information of the table

- *New* creates a new empty table/workspace. (Feature 12)
- *Open* is used to open .csv files. (Feature 6)
- *Open Recent* is a menu that is used to quickly open .csv files that have recently been edited. (Feature 7)
- *Save* saves the current workspace. (Feature 5)
- *Save as* saves the current workspace with a desired .csv file name. (Feature 5)

The *Edit* section, figure 14, contains buttons that are mainly used to add/adjust/edit information in the program.

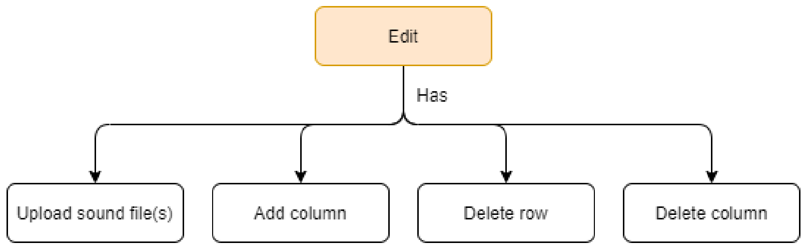


Figure 14: Edit menu layout

- The *Upload sound file(s)* button is used to upload sound files that are then analyzed in the CNN model. All of the valuable information is then added to the table. (Feature 1)
- *Add column* adds a column to the table with a user inputted column name. (Feature 2)
- *Delete row* deletes the currently selected row. (Feature 4)
- *Delete column* deletes the currently selected column. (Feature 3)

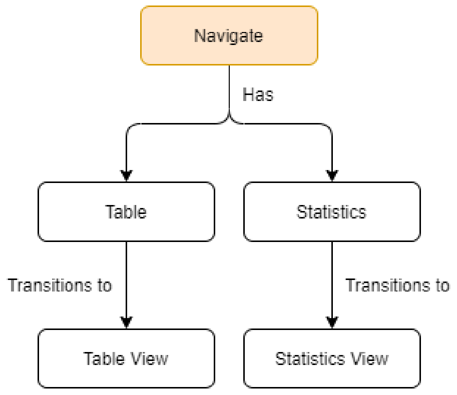


Figure 15: Navigate menu layout

The *Navigate* section, figure 15, is used to navigate between the different views of the app

- *Table* makes the user transition to the Table View
- *Statistics* makes the user transition to the Statistics view
- The *Statistics View* shows statistics of the sound files in the table that are helpful for the veterinarians to analyze

The Table View and Statistics View are explained in more detail in sections 2.6.2 and 2.6.3 of this document.

Enabling buttons The `Delete Row` and `Delete Column` buttons can only be clicked when a respective row or column is actually selected by the user, so we have functions to handle this: `activateDeleteRowButton()` and `activateDeleteColumnButton()`. Both these functions call the `setEnabled()` function from the QButton class, enabling it to either true or false depending on what is passed from the TableView, where the actual check for selected rows/columns is performed.
Likewise, the Statistics page can only be navigated to when the user is currently on the Table page, and vice versa. For this we have `activateNavigateStatisticsButton()` and `activateNavigateTableButton()`. These functions also use QButton's `setEnabled()` function, but they receive their input from within the ViewHandler class, where the `switchViews()` function indicates which screen the user is currently on and which navigation button should be enabled.

Close Event When the application is closed, we use the function `closeEvent()` from the QMainWindow class to perform a few actions that need to be executed upon closing the program. These include saving the recent files for the `OpenRecent` button in the file menu, and saving the file path at the top of the window. The functions for writing these to their respective text files in `BackEnd` are used for this. (Feature 7)

2. TableView
The table view of the program inherits from the PyQt5 QFrame class, and we use a QTableWidget for the table widget that is added to this page.

Writing to and reading from the table This class implements the function `populateTable()`, which takes a data frame from the `BackEnd` and fills the table with it. It is used when it is necessary to fill the table with data that has been input to the `BackEnd` first (loading the program, opening an uploaded .csv file, or clearing the table). It also implements `getDataInTable()`, which takes all the data currently in the table and returns it as a data frame. This is used when it is necessary to retrieve the data that is in the table (e.g in the `refresh()` function of the `BackEnd` - used for updating its data frame after the table has been edited by the user).

Enabling buttons As mentioned in section 2.6.1, this class is responsible for passing a boolean value to the `ViewHandler` class to indicate whether the 'Delete Column' and 'Delete Row' buttons should be enabled. It does this through
`activateDeleteColumnButton()`, which checkes whether the amount of selected columns is more than 0, in which case the boolean value passed is true. If not, it passes false. The same is done for the rows with
`activateDeleteRowButton()`.

3. StatisticsView
The statistics view inherits from the PyQt5 QFrame class. It is used to provide a overview of the most relevant data that is currently in a session.

The statistics view can be split up in an upper (Fig 16) and lower half (Fig 19). The upper half draws bar graphs of categorical data that is stored in the 'standard columns' of the data (Feature 8). These standard columns are the columns that contain data produced by the application's model. The user can also select additional columns to be drawn as graphs in a list of dynamically updating checkboxes (Fig 17) (Feature 9). (Un)checking a checkbox triggers `uponCheckBoxInteraction`, iterates over every checkbox to see which ones are checked. If a box is checked and it is not already drawn, it is drawn on the view. The statistic view can retrieve data from the backend and thus dynamically update the graphs every time the view is visited by a user (Fig 20). These bar graphs are drawn using the `matplotlib` library with the data present in the application.

The lower half has a scrollable list of currently uploaded sound files and

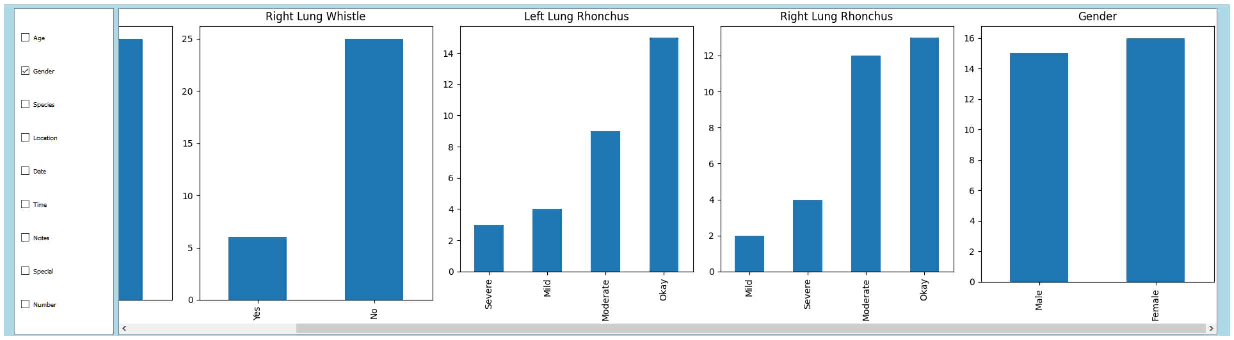


Figure 16: Bar graphs statistics view

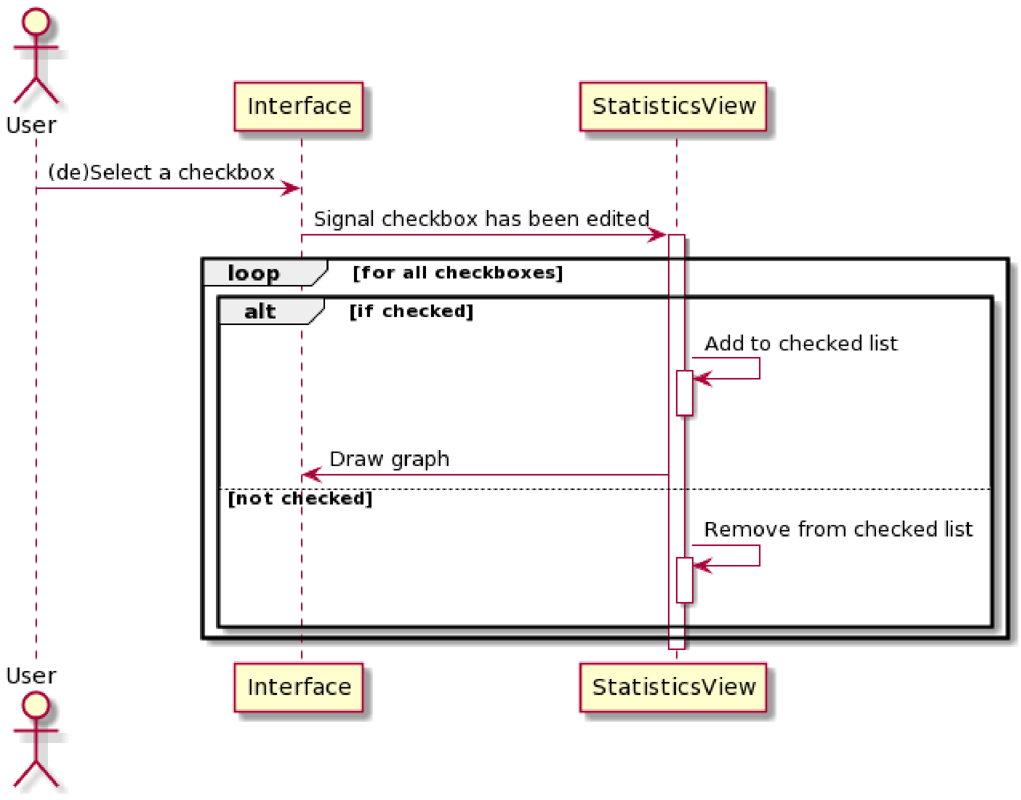


Figure 17: System diagram checkboxes

a graph of the corresponding spectogram (Feature 10). The list of currently uploaded sound files is dynamically updated every time the view is selected by the user (Fig 20). The list consists of radio buttons, allowing only a single button to be selected at any time. Selecting a button will trigger the application to draw the corresponding spectogram of the sound file associated with the radio button (Fig 18). The spectogram is generated by the `librosa` library and the graph is drawn by the `matplotlib` library. When a spectogram is selected, a user can press the button at the bottom left (see figure 19) to download the spectogram. (Feature 11)

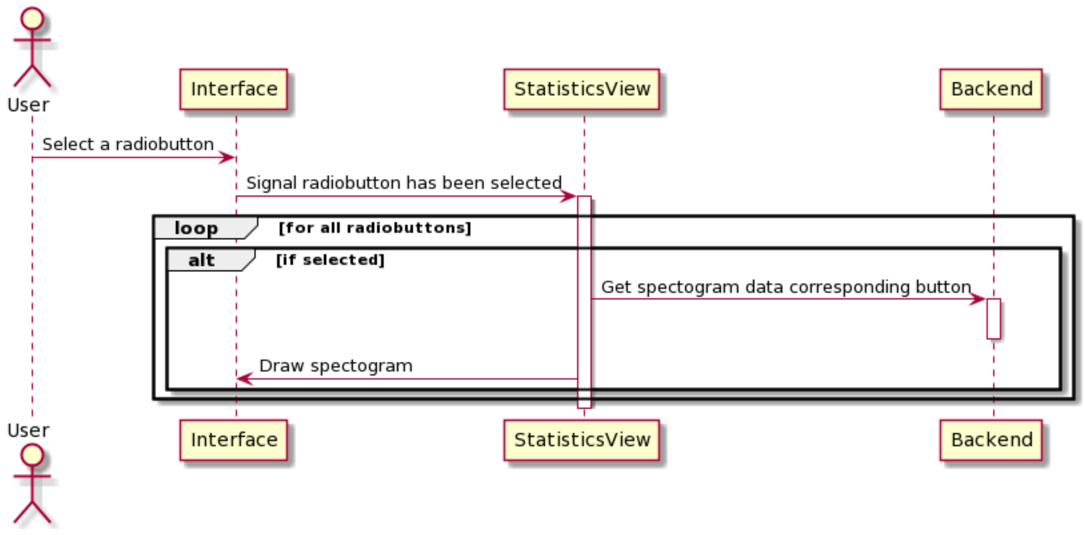


Figure 18: System diagram radiobuttons

Every time the user selects the statistics page, the following methods are called to set up the entire view (Fig 20). A `refreshStatisticspage()` method is called. This method calls `refreshRadioButtons()`, `refreshCheckboxes()` and `plot()`. The first removes all radiobuttons currently presented on the page, and then adds a radiobutton for each soundfile currently uploaded in during the current session. Every button is connected to a `plotSpectogram()` function. The second removes all checkboxes currently on page. Then adds a checkbox for each additionally user-added column. When a checkbox is checked, it gets stored in a list that keeps track of checked boxes. The third

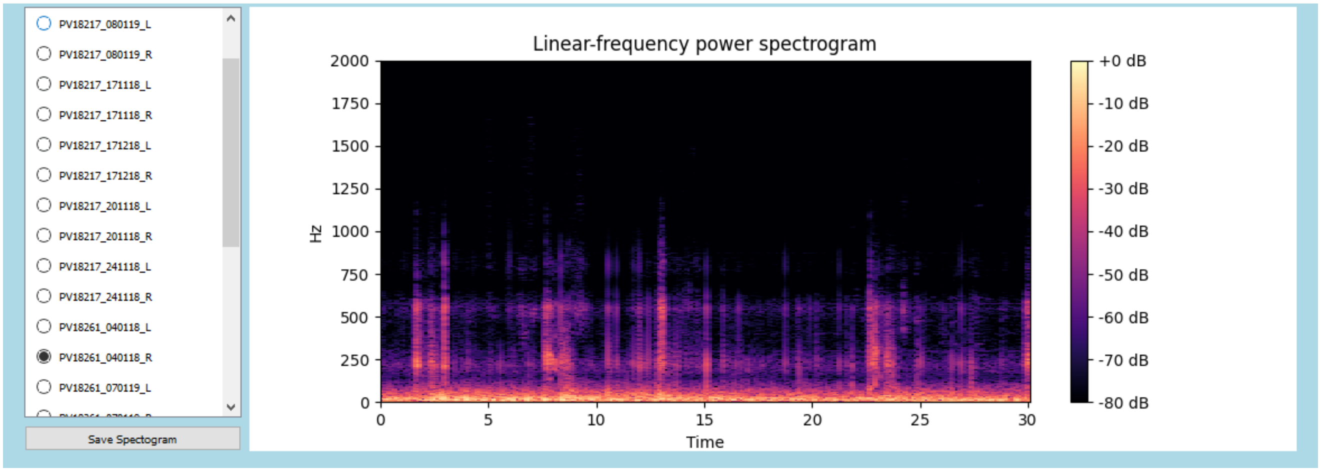


Figure 19: Spectogram statistics view

plots all the graphs. This is done by repeating the same set of steps for all graphs. It clears the graph currently drawn, and checks if data relevant to that graph is available. If it is, a graph is drawn. If it is not, an empty graph with corresponding axes is drawn. (Feature 8, 10)

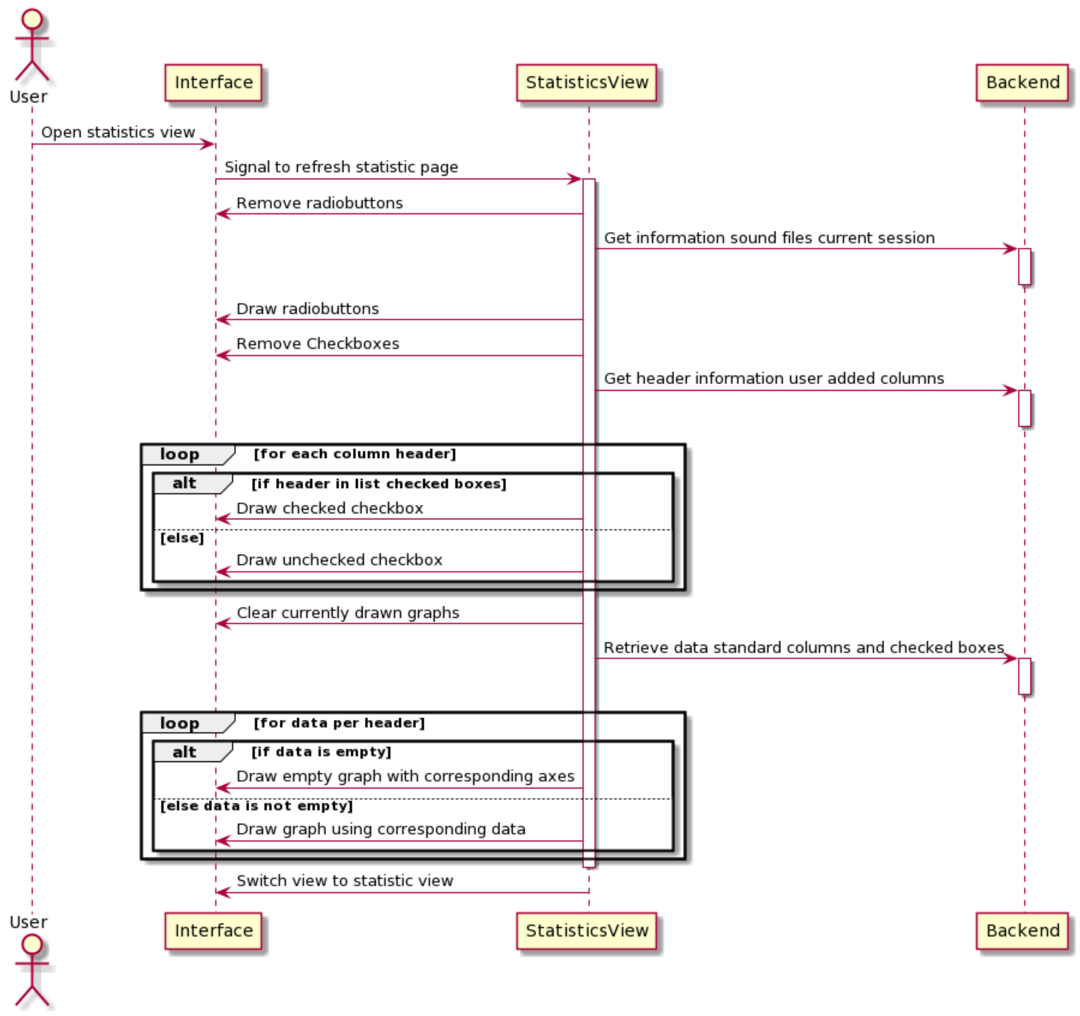

User
Interface
StatisticsView
Backend
Open statistics view
Signal to refresh statistic page
Remove radiobuttons
Get information sound files current session
Draw radiobuttons
Remove Checkboxes
Get header information user added columns
loop [for each column header]
alt [if header in list checked boxes]
Draw checked checkbox
[else]
Draw unchecked checkbox
Clear currently drawn graphs
Retrieve data standard columns and checked boxes
loop [for data per header]
alt [if data is empty]
Draw empty graph with corresponding axes
[else data is not empty]
Draw graph using corresponding data
Switch view to statistic view

Figure 20: System diagram statistic view

2.4.3 Miscellaneous: Splash screen and progress bar

The `Miscellaneous` package stores the `ProgressBar` and `SplashScreen` classes. The progress bar is shown when the user uploads a sound file, for the duration of the time that the CNN needs to analyze the file. The splash screen is shown for 3 seconds upon opening the program. Credit for the design of the splash screen is given to: Robin Vredenborg.

2.4.4 Controller

Overview of the controller:

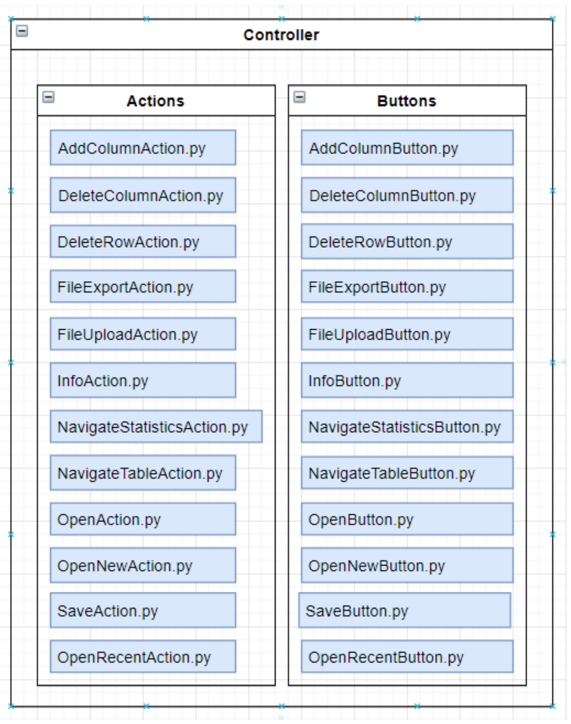


Figure 21: Controller

1. Buttons

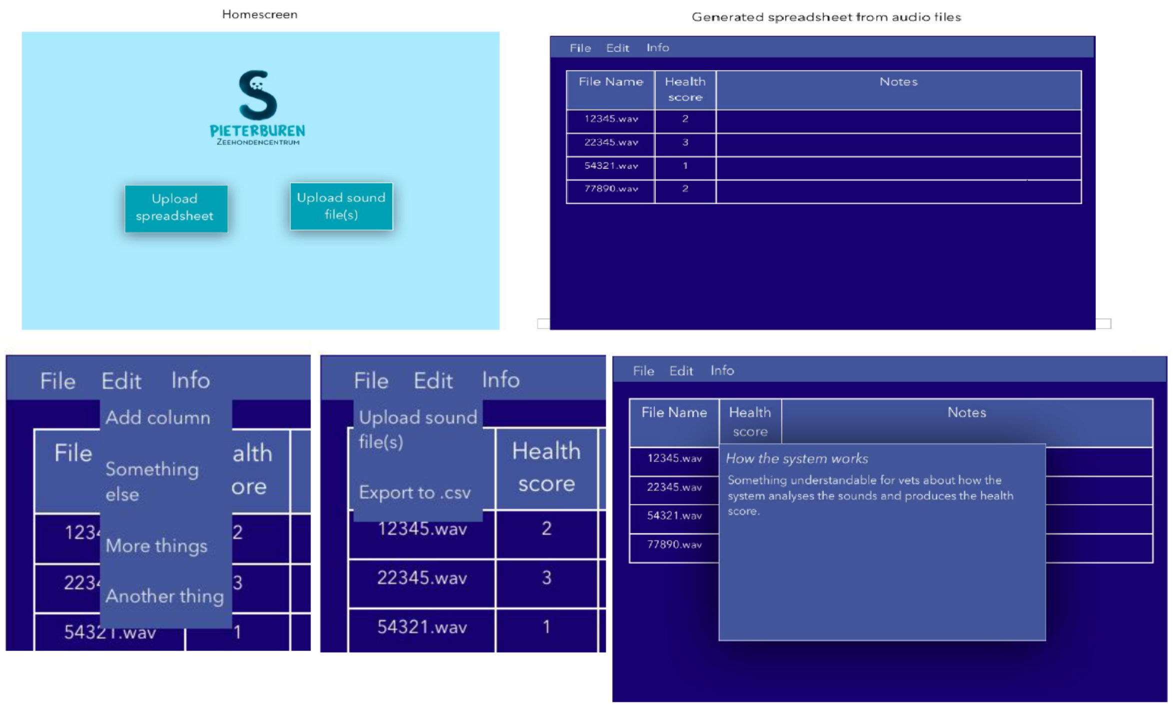
As discussed in the **Menu Bar** part of section 2.6.1, Wavealyze has various buttons located in a menu bar at the top of the screen. The user clicks these to perform various actions within the program. Since the functionality of each button has already been elaborated on in the section previously mentioned, this section of the document will focus solely on the architecture that we used. The `Buttons` folder contains a python file for each button, in which the button itself is initialized as a PyQt5 QAction object. Here we also set the keyboard shortcut for each button, and also connect the to the method inside of the `Actions` folder that is triggered when the button is clicked.

Some buttons also need a specific condition to be met before they can be enabled, so their enabled status is initially set to false. Changing the enabled status to true is handled elsewhere in the program, which is discussed in sections 2.6.1 in **Enabling buttons** and 2.6.2 in **Enabling buttons**.

2. Actions
As mentioned in the above section, each button has a corresponding action file (for example, `FileExportButton` and `FileExportAction`). Inside of each action file, we have a function `uponActionPerformed()`, which is the function that the relevant button is connected to. Inside of this function is another function call, and this function contains that actual code for executing the process for the respective button. Going with our previous example, this function is the name of the action, e.g `fileExportAction()`. The reason for this is to create a separation between the action event that takes place when the button is clicked, and the process itself. Sometimes we use the function for the process itself elsewhere in the program. For example `fileExportAction()` is not used only inside the `FileExportAction` file, but also inside of `SaveAction`. In this case, it is cleaner to use `fileExportAction()` instead of `uponActionPerformed()`.

2.5 GUI Design Process and Prototype

For the design of the graphical user interface, we first created a simple prototype using a graphical design software (Fig 22). The prototype design contains the following aspects: a main screen for the app, acting as a homepage, containing two buttons. The buttons are for either uploading a sound file, or uploading a spreadsheet. The other screen included in the prototype, is the screen displaying all of the information and data. This screen is called the table view. It contains a table which takes up most of the screen, and has some columns which should always be included (in the early stages of development, these are just the file name and the health score given by the cnn model). The other columns are optional and can be added. At all times, there is a menu bar present above the view. This menu bar provides menu's named: file, edit and info. The file menu would produce a submenu with options like exporting and saving. The edit menu covers options like adding columns. The info menu is supposed to offer an overview for the veterinarians on how to use the application, sort of like a digital manual. Below, you can find the prototype with images of the home screen, table view, file menu, edit menu and info menu respectively.



File Name	Health score	Notes
12345.wav	2	
22345.wav	3	
54321.wav	1	
77890.wav	2	

Something understandable for vets about how the system analyses the sounds and produces the health score.

Figure 22: Prototype design

When contemplating the design of the user interface for this particular project, multiple frameworks have come to mind. It is unnecessary to opt for a language/framework that offers complex and detailed interface design, due to the simplistic nature of the application. The entire application is written in the python programming language, and python offers frameworks with complexity sufficient for the project. The initially considered option is a package that is standard included with python. This framework is called Tkinter. The framework is meant for quickly realising simple interfaces with little lines of code required. Initially, this simplicity seemed compatible with the desired design. However, the framework lacks components that are fundamental to the project. Tkinter offers very little choice when it comes to complex 'widgets'. While setting up the interface with this framework, this lack of widgets prosed an issue. The framework does not support widgets resembling a spreadsheet or a table, which is crucial to the project. The two viable options were: use an unofficial package or design/ create this component with the means provided by framework. After having our productivity affected by the inadequacy of the initially chosen framework and its lack of crucial components, we opted for a different framework.

An informal inventory of crucial interface components was: table/spreadsheet, graphs. The framework which offers everything needed for the project is called PyQt5, and has a sufficient documentation available. It also comes with a table widget and spreadsheet widget, which is exactly what is needed. Since PyQt5 offers plenty documentation and a lot of representation online, it is a suitable option for this project.

Using the PyQt5 framework, we have created a graphical user interface the following way. One class is designed to act as the 'main' window of the application, and has frames stacked on top of it. These frames represent the various views which the app should provide. Users are able to traverse the interface by clicking buttons, which in turn call functions that put the desired frame/view on top of the stack. The views are designed using a grid layout manager. This allows us to divide the views into rows and columns and place items at desired locations. An example of this is given below in figure 23 (Note: this is the home screen that we used initially in Block A, but we decided to remove it from the program and only use the table view in Block B).

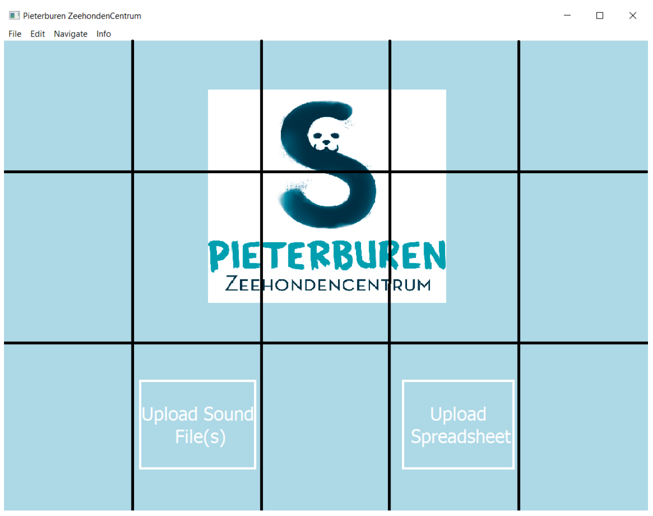


Figure 23: Grid layout example

Using this technique, components can be placed with sufficient precision. In this example, by splitting the frame up into three rows and five columns, we can allow the logo (a placeholder image in this example) to span three columns and two rows, and place it in the middle. We then allow the buttons to span a single row and column, and place these to the left and right of the logo without touching the borders of the frame.
The front end relates to the view- and the controller part of the MVC model. The following to paragraphs talk about the position of these parts in the overall application.

2.5.1 Block B:

After presenting the initial version of our application to the client, some more insight into a desired design was gained. Originally, it was thought that a home screen would be good for aesthetic purposes. It would provide the application with some identity and serve as a introduction to users. However, functionally the home screen is very redundant. The functionality provided by the buttons on the home screen are also accessible via the menu bar. Therefore, it was decided to scrap the home screen, and instead direct the user to the table view when opening the application.

To maintain some of the application's identity that is now lost due to the absence of the home-screen, a brief splash-screen was introduced, which is included in figure 24.

Figure 24: Splash-screen

2.6 Requirements Program Usage

File Naming Conventions
For the application to work as expected, it expects certain conventions to be in place. For the user experience, it is desirable that recordings of both the left and right lung of the same recording id are inserted into the same row. The effect of this is that a row is dedicated to a certain recording ID, and thus a recording ID is uniquely present in the table.

- The last character of the file name should indicate whether the sound file is from the left lung or the right lung (indicated by a L or R respectively).
- This character has to be either an uppercase L or uppercase R.
- If the last character of a file name is not either an L or an R, the program will choose to place the result in that of the right lung's column.

CSV File Upload Convention
When the user uploads a csv file to work on and extend in the program, the program expects this file to have certain conventions regarding its columns.

- The uploaded csv file is expected to have the following columns for indicating the audio file name analyzed, the health score for the left lung and the health score of the right lung.
- The exact names of these columns are ”File Name”, ”Left Lung Whistle”, ”Right Lung Whistle”, ”Left Lung Rhonchus” and ”Right Lung Rhonchus”.
- If these Column names are present, the program can proceed to automatically put the results of the analysis under these columns.
- If these columns are not present, the program will add these columns and put results of analysis under these columns. This could result in a an undesirable layout of document.

3 Technology Stack

The software was asked to be made for Windows, so is currently only available as a Windows application. The software is based on a single user profile, so no login is currently required.

Languages

Python

Python is among the most used programming languages currently available. With its easily readable syntax, it is used for many purposes, such as web development, software development, etc. Our entire program, so front-end and back-end, is written in Python, with help from multiple Python packages.

Python libraries

PyQt5

PyQt is a Python binding for Qt, which is a cross-platform Graphical User Interface toolkit. It connects the Qt C++ framework with the Python language, so that it can be used as a GUI module. We have used the fifth version, PyQt5, as our main package when designing the GUI of the application.

librosa

`librosa` is a Python package made for music and audio analysis. In our program, mainly used to extract features from audio files to be used in the CNN models and to display the audio spectograms.

NumPy

NumPy is an extension on the Python language with the purpose of adding the support of multi-dimensional arrays and matrices, along with a large library of mathematical functions that can be used with these arrays. The package has helped us doing exactly that; we've used mathematical functions to work with larger arrays that we're needed with the processing of the audio files.

math

The `math` Python module provides access to many mathematical functions defined by the C standard. We've used this package to do mathematical tasks that would have taken more time to program by ourselves.

versioned-hdf5

Hierarchical Data Format (HDF) is a set of file formats designed to store and organize large amounts of data. The weights/models used in Wavealyze are of type `hdf5`, that are then used to analyze the audio files.

Keras

Keras is a software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library. We've used the Keras library only to load in the models used to analyze the audio files.

TensorFlow

TensorFlow is a software library for machine learning. It focuses particularly on the training and inference of deep neural networks. Keras supports only Tensorflow as its backend, which is thus also used to load in the models used to analyze the audio files.

pandas

pandas is a fast, powerful, flexible and easy to use data analysis and manipulation tool, built on top of the Python programming language. Mainly used in our program to store all of the data that is uploaded and calculated, which can then easily be converted to a .csv file and vice versa.

matplotlib

The `matplotlib` library is a library used to create static, animated, and interactive visualizations in Python. The statistics of the data in our program can be visualized clearly in our `statisticsView` by plotting the data using this library.

4 Testing

Testing was essential for this project to confirm that the methods used in our code work well. As we have gotten and thought of important requirements by discussing the project with our client, they expect that our program fulfils these requirements and that the features based on these requirements work as expected. Therefore, all of the testing focuses on the implementation of these requirements.
In this project we've done three types of testing: Unit testing, Integration testing and User testing. The upcoming sections will describe how the form of testing is implemented.

4.1 Unit/Integration Testing

Unit Testing is a method of testing the smallest piece of code, typically a class or method. The main goal is to validate that each unit of the software performs as expected.

Integration testing is defined as a type of testing where software modules are tested as a group. These modules are also tested to check whether these modules are integrated logically.

We have used the Python unit testing framework for testing the important methods of our program, which mostly correspond to the various requirements. Some of these tests are unit tests, however most of our tests test multiple components at once, so are seen as integration tests.

pytest

4.2 User Testing

The most essential part of testing was the user testing phase. We have to make sure that every requirement is properly implemented, so the veterinarians can actually work with our program. This phase was split up in two parts: For the functional requirements the user could enter 'Succesfully implemented' or 'Other', where they could specify what the problem was they were experiencing. For the non functional requirements the user could enter a number between 1 and 10, stating how well they agreed with the requirement.

Unfortunately, the data samples for this testing phase were not gathered from our client. We were experiencing technical difficulties while making the executable, and so the user testing phase was pushed too close to the deadline to gather information from them directly. Because we know this type of information is very important for our project, each group member asked a family member or friend to test out our application to get some feedback.

The results of our User Test phase was very positive. No user found an error while testing out the program. Following are the results of the non functional requirements, showing in a barplot how well the user scored on intuitivity and practicality, and professionality.

As can be seen in figure 25 and figure 26, the users tell us that our ap-

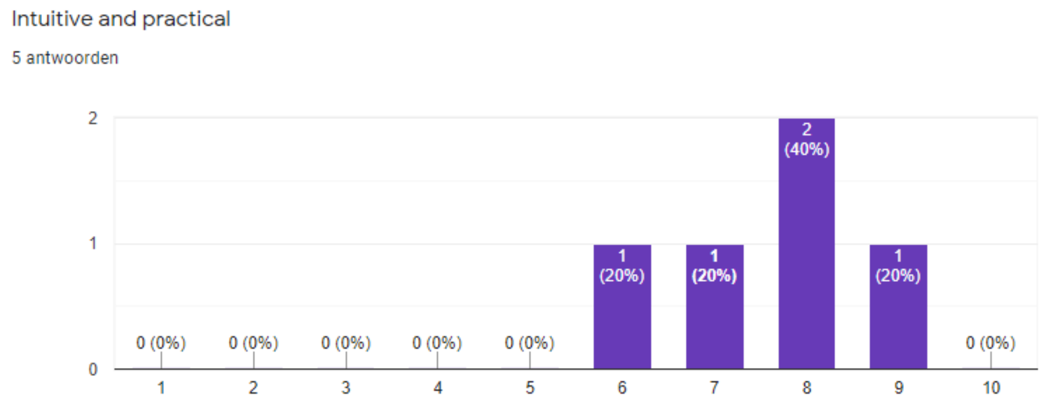


Figure 25: Intuitivity and practicality

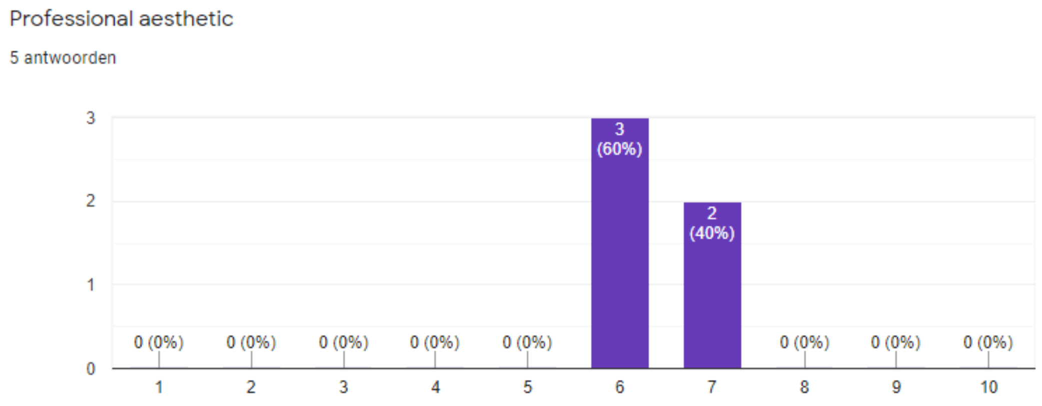


Figure 26: Intuitivity and practicality

plication is very intuitive and practical, but feel like the professional look of the application could be better.

5 Team Organization

5.1 Planning

We started our project by organizing a meeting with our client, who acted as an intermediary between our Software Engineering group and the vets at Zeehondencentrum Pieterburen. Together with her, we went over the initial proposal document and discussed what the requirements for this project would be. From here we came up with an initial requirements list, including those that were essential to the project, those that would be nice to have but were not absolutely essential, and those that we would not do. The initial meeting was not fully sufficient to clarify all the details of what the client wanted, so we had a conversation via email to address this.
Next, we met as a team to create a prototype, which we then sent to Estefania to ensure that everybody was on the same page about how the application should look and function. The client felt that the prototype represented her vision for the application well, so no further changes were needed.
We had another meeting amongst ourselves to plan the architecture of the program. The following figure shows the initial structure that we came up with. The application itself is relatively simple, and the requirements that make up the backbone of the application were clear from the beginning, so this meeting was successful, and the final architecture is similar to what we planned. We emphasized modularity in our architecture planning, so that adding additional elements or changing things along the way would not be a difficult task.

5.2 Communication within team

Our team had meetings every week in order to discuss the progress on the application amongst ourselves and with our TA. We used these meetings to clarify anything that was unclear, bring up new ideas, and organize the tasks for the week. After these meetings, we created a list of tasks and everyone chose what they would work on.
For communication with our TA, we used Slack. Here we planned meetings, asked short questions, and exchanged files and documents.
Within our team, we used WhatsApp to communicate. Significant questions and ideas were discussed in meetings, but smaller topics and questions that

didn't require a meeting were discussed in the WhatsApp group. We also used this to divide up tasks and organize deadlines.
We used GitHub to host our code and for version control, so that we could all contribute to the project.

5.3 Communication with the client

In the first weeks of the course, our client was present during two of our Google Meet meetings. One meeting was to extract the requirements for the project, and the other was to show her the prototype that we had created. After we started working on the project, our communication was mainly via email. She was also present for our intermediate and final presentations, in which she gave feedback and asked questions. Our email correspondence generally took the form of us asking questions about aspects of the project requirements that we needed clarification on, as well as updating her when we thought changes to the initial requirements were necessary. A log of all client communication is given in section 6 of the requirements document.

6 Appendix

6.1 Feature tracking matrix

Following from the requirements document, we can specify features relating to those requirements. This table can be used to track progress on the features throughout this project. Requirement ID can be found in the requirements document.

Feature ID	Feature name	Requirement ID	Included
1	Upload an audio file to get the health scores	4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.3.1, 4.4.4	✓
2	Add columns to the table	4.2.1, 4.2.5,	✓
3	Delete column from the table	4.3.1	✓
4	Delete row from the table	4.2.6	✓
5	Save session data as .csv file	4.2.2	✓
6	Open selected .csv file	4.2.3	✓
7	Store recently used .csv files	4.3.6	✓
8	View dynamically updating bar graphs of CNN generated data	4.3.2	✓
9	Select additional columns to be graphed	4.3.3	✓
10	View selected spectogram of uploaded .wav file	4.3.4	✓
11	Download spectogram of uploaded .wav file	4.3.5	✓
12	Edit table contents	4.2.1	✓
13	Autosave	4.2.7	✓

7 Change Log

Who	**When**	**Which section**	**What**
Annika	05-03-21	The document	Created the document
Brian & Roan	09-03-21	Start of models	First draft
Brian	09-03-21	Conceptual model	Addition of InputHandler and additional description
Brian & Roan	16-03-21	Conceptual model	Change of flow
Roan & Brian	16-03-21	Module View	Updated the module view
Roan & Brian	16-03-21	MVC Diagram	Added MVC Diagram with explanations
Brian & Roan	29-03-21	The document	Restructured document and added additional information to View and Input Handler
Mike	29-03-21	Architectural Overview	Added Frontend and Backend
Roan	30-03-21	Architectural Overview	Added Audio pre-processing and CNN
Brian	30-03-21	Architectural Overview	Updated Frontend, Backend and InputHandler
Brian & Roan	30-03-21	Technology Stack & Team Organization	Added info to the sections
Mike	30-03-21	The document	Making adjustments and adding info
Mike	11-05-21	Requirements Program Usage	Created section, wrote name and csv conventions

Who	**When**	**Which section**	**What**
Annika	26-05-21	All	Re-organized the structure of of the document & rephrased some sections
Annika	26-05-21	Introduction	Edited introduction to be up-to-date with current version of program
Andrei	26-05-21	2.3 Model-View-Controller Pattern	Added this section
Roan	26-05-21	Conceptual model	Added 'General structure of Frontend' with explanations and diagrams
Roan	28-05-21	Model	Added sequence diagrams from requirements UCs to corresponding text
Brian	28-05-21	Module view	First updated draft
Annika	29-05-21	2.1 General Overview of Components	Wrote this section
Andrei & Annika	29-05-21	2.5 Model	Wrote the sections for CNN, InputHandler, OutputHandler, & BackEnd

Who	**When**	**Which section**	**What**
Andrei & Annika	29-05-21	2.6 View	Wrote the sections for ViewHandler, TableView, & Miscellaneous (note: diagrams were made by someone else)
Roan	31-05-21	Technology Stack	Rewritten entirely and added logos
Mike	31-05-21	StatisticsView	Created section
Mike	31-05-21	GUI Design Process and Prototype	Updated/rewritten text & added block B
Brian	01-06-21	MVC	Additional explanation of MVC and the overview of the model, view and controller
Brian	03-06-21	MVC Implementation	Start of this subsection
Andrei	07-06-21	2.7 Controller	2.7.1 Buttons and 2.7.2 Actions
Mike	07-06-21	2.6.3 StatisticsView	Updated text, added figures and references, added system diagrams
Brian	08-06-21	MVC Implementation	Restructure + birdseye perspective
Mike	08-06-21	Document	Added figure captions/numbers Added references
Brian & Mike	08-06-21	Feature tracking + matrix	Feature tracking
Annika	09-06-21	Team Organization	Wrote this section
Andrei	11-06-21	Document	Visual changes
Roan	11-06-21	Testing	Added text to Testing

Who	**When**	**Which section**	**What**
Andrei	13-06-21	Whole Document	Fixed formatting
Annika	13-06-21	Whole document	Edited some sentences for clarity according to feedback from TA
Roan	14-06-21	3.2 Conceptual View	Updated diagram and text
Andrei	14-06-21	Appendix	Created this section
Roan	14-06-21	Testing	Added changes to Unit/Integration Testing
Brian	14-06-21	Testing	User test
Brian & Mike	14-06-21	Document	Restructure and proof reading