FACULTY OF SCIENCE, ENGINEERING AND COMPUTING

School of *Computer Science & Mathematics*

BSc DEGREE
IN
Computer Science

PROJECT REPORT

Name: Harry Smith

ID Number: K1738426

Project Title: Audio Event Detection for Emergency Service Vehicles

Project Type: Build

Date: 10/05/2020

Supervisor: Dr Mark Barnard

Plagiarism Declaration

The following declaration should be signed and dated and inserted directly after the title page of your report:

Declaration
I have read and understood the University regulations on plagiarism and I understand the meaning of the word *plagiarism*. I declare that this report is entirely my own work. Any other sources are duly acknowledged and referenced according to the requirements of the School of Computer Science and Mathematics. All verbatim citations are indicated by double quotation marks ("…"). Neither in part nor in its entirety have I made use of another student's work and pretended that it is my own. I have not asked anybody to contribute to this project in the form of code, text or drawings. I did not allow and will not allow anyone to copy my work with the intention of presenting it as their own work.

Date

10/05/2021

Signature

[illegible]

1. Introduction

Emergency service response time is a regular political discussion. Targets and measures often change, such as the introduction of the tiered response time for ambulances instead of the blanket 8-minute target (Baynes, 2018), for example. Vast amount of consideration is put into the emergency response vehicles themselves: the colour of the lights, the wave frequency of the siren, what ramifications the doppler effect has on moving through traffic. However, throughout this discussion, little consideration is often given to the vehicles and drivers charged with responding to a siren. How soon do they hear the siren? How early can they notice the flashing lights? What are the effects of roundabouts and junctions on driver response times? This low hanging fruit seems to offer a simple way to improve driver responses to emergency service vehicles, by making users more aware of encroaching sirens at the earliest possible point, thus increasing the reaction window.

There are other reasons why it is useful to aid drivers in their notification of sirens. Firstly, for accessibility reasons. Drivers with a hearing impairment are reliant on seeing the flashing lights of an emergency service vehicle, which can often not be visible until the vehicle is much closer, thus reducing the available reaction time.

Secondly, the prevalence and development of autonomous vehicles show how another method is required for detecting emergency service vehicles without human interaction. Computer vision systems will aid in this, but obstructions and sensor technology can reduce effectiveness in this area. A noise detection system would be more effective for early warning of oncoming emergency vehicles.

The aim of this project is to create a siren detection system which can detect an incoming siren in a noisy environment. To realise this goal, I will build a deep learning system to correctly classify sirens. While considerable research is available in deep learning for computer vision, audio classification information is more sparse. To compensate for this, I will research how to cross pollinate the two techniques; using established deep learning programs for computer vision to solve an audio detection problem. Examples for developing spectrograms and other audio visualisation techniques are available, and these are what will be used to train the deep learning model.

The original scope of the project was to expand on this idea and develop features such as distance and direction detection to aid a driver’s decision making. Due to the COVID-19 pandemic this ambition has been waylaid. Developing these features would require a vast amount of binaural data, which is not available online. With the pandemic disrupting the back-up plan of recording the data, the scope of the project has adapted.

Instead of developing these features, the project has focused on deep learning research to understand and create the best methodology for siren detection. This involved examining different types of neural networks and methodologies and assessing their respective performance on training and test data. For each network, consideration was put into overfitting and underfitting as well as their performance in practice and their relative complexity. The four networks referenced in this report are: Multi-Layer Perceptron (MLP), 1-Dimensional Convolutional Neural Network (Conv1d), 2-Dimensional Convolutional Neural Network (Conv2d) and Long Short Term Memory Recurrent Neural Network (LSTM).

This report begins by considering the literature available on siren and audio detection more generally, as well as the methods and workflows already published. Here the different types of networks are explained and analysed with predictions made for their effectiveness to the task before development and testing

The report continues by looking at the analysis techniques used in the project. This includes not only the use of requirements elicitation and analysis tools and procedures, but also the project methodology and workflow.

Following this, the design stage of the report examines the data structure approaches taken, as well as some of the design techniques for the system. Here consideration is given to the user by displaying the narratives and wireframes of the software deliverable. This section also focuses on the architecture of the systems, and how the four different types of neural networks fit within this architecture.

The next section of the report looks at the technical implementation of the neural networks, with a discussion of the deep learning libraries used as well as the coding principles and practises. Here the focus is on demonstrating the workings of the software and an exploration of the technical issues encountered during development.

Penultimately, the report looks at testing and validation strategies. With an initial focus on technical testing, the report outlines the approach and results of the unit tests performed on the code. This is followed by a human approach to testing, and the results of various black box testing and feedback.

The report concludes with a critical evaluation of the project and the development process. At this stage there is particular scrutiny on what could be improved both in the technical implementation and the project management approach in future. Consideration is also given to potential future features and expansion of the product, before a conclusion reviews the original aims and objectives of the project.

2. Literature Review

Neural networks have proven to be extremely effective for image classification and they are beginning to “show promise for audio” (Hershey et al, 2017). The same paper also raises the point that audio data is more widely available now than ever before, thus using large amounts of data to train audio based Convolutional Neural Networks (CNNs) is becoming a more realistic task in many fields. While other approaches have also been presented, such as the use of support vector machines (Guodong, 2003), the vast amounts of data available suggest that the most promising area of research in this area is that of deep learning.

The most-common method for audio classification in deep learning is through the use of spectrograms. As discussed, the area of image classification is well established and researched, and using spectrograms essentially turns the audio classification problem into an image one. The use of spectrogram analysis for audio recognition has been used in earnest for over 70 years, with the representation of speech examined by Greenberg and Kingsbury in their 1947 paper (Greenberg and Kingsbury, 1997). In this paper they discuss use of the “modulation spectrogram”, a visual representation of both time and frequency for speech analysis. They begin to apply this to what they refer to as “automatic recognition” using Markov models and the multi-layer perceptron, both areas of early research into machine learning.

More recent publications expand on these ideas to demonstrate the effectiveness of using deep learning methodologies for audio classification of sirens. This was observed when looking at siren detection for safety improvements in traffic environments (Meucci et al, 2008). This paper stated multiple features sirens have in relation to their environment, including the ‘long duration of a siren signal’ in addition to their frequency. Where Greenberg and Kingsbury looked at syllables as key distinguishing features in speech recognition, the frequency and length of sirens can be used to the same effect in siren recognition to distinguish them from their urban environment (Greenberg and Kingsbury, 1997). Meucci et al conclude that the siren detection ‘reduces to a pattern recognition task’ – an area which compliments modern deep learning research (Meucci et al, 2008).

Meucci et al also examine the use of “low priced condenser microphones” (Meucci et al, 2008) for direction detection. This is in conjunction with Salekin et al, where it is stated that a similar microphone “already exists in the cabin of most modern cars” (Salekin et al, 2019). Thus, not only is using such equipment suggested in an academic setting, but it is also readily available in a practical real-world environment. While distance and direction detection out of scope for this report, it is important to consider the feasibility of the project and its stretch goals in a commercial environment.

The conclusion from this research is not only well founded but obvious: deep learning for audio classification is a growing field with initial academic research performed into its feasibility when applied to siren detection. The remaining question is what deep learning methods are promoted as being effective for such a task. Choi, Fazekas and Sandler examine the use of convolutional neural networks in the field of music classification (Choi et al, 2016). In their paper, they examine 3 types of convolutional neural networks (CNNs) before introducing a convolutional recurrent neural network (CRNN). After experimenting with all four types of network they assess their accuracy in multi-class classification of music genres. In their research, they discovered a “trade-off between speed and memory” as the number of parameters varied. Their work in this area is what will be developed on for this project; testing different deep learning neural networks on the binary classification problem of siren detection to determine accuracy for use in an urban setting while giving consideration for the speed and memory requirements aforementioned.

At this point, it is worth considering other commercial interests in this setting. Autonomous vehicles companies have published information hinting at use of audio detection for emergency service vehicles, such as Waymo (Google, 2021). Furthermore, the German company ZF have published some initial reports into the results of their audio recognition system (Askari, 2018). These results are promising and show how much traction and noise the area of research is creating in the commercial environment. However, as of time of writing, there is no published research into the effectiveness of different types of neural networks, thus the work discussed in this paper provides context and value to the field.

One final consideration must be given to open-source work performed in this field. Firstly, Mike Smales’s repository (Smales, 2020) and report looking at urban sound classification using a multi-layer perceptron model provides a useful starting point for looking at building deep learning models for urban classification as well as audio file data visualisation. Here some logic is provided for the metrics required for multiclass classification as well as a guide for data transformation of uncompressed audio files. This is expanded further by Seth Adams (Adams, 2021), who’s repository provides an example of using the Kapre library for transforming audio files on the fly, and working with multiple model types with efficient code, a crucial requirement in this project.

Both of these sources, along with the aforementioned published literature, will serve as inspiration and guides for development to answer the unique question of which deep learning model is most effective for siren classification in urban environments.

3. Analysis

3.1 - Problem Definition

The literature review presented shows the availability and growing development of audio classification systems. An assortment of such programs are available to perform different tasks. Some implementations are for academic or educational purposes whereas other audio classifier systems serve much more practical and real-world causes, such as audio genre classifiers. While audio classifiers are commonly multi-class, binary classification is a well-established area and can easily be applied to audio. The problem being defined with this project is to combine these variables to examine what the best deep learning solution is to a binary classification problem in a noisy environment. To examine this further a range of established analysis techniques have been used, starting with SWOT analysis.

3.2 - SWOT ANALYSIS

The first step in the analysis process is to look at the strengths, weaknesses, opportunities and threats to the project. This was done early in the analysis process to create a high-level view of the task in hand while examining whether it was a worthwhile exercise to build the system. The strengths and weaknesses of the SWOT table examine internal areas which exist that can assist or derail the mission. The opportunities and threats then perform the same examination for external forces.

3.3 - Functional and Non-Functional Requirements

The next step in the analysis process is to draw up a list of functional and non-functional requirements. Here the functional requirements will be a list of requirements derived from the users to be implemented. The functional requirements aim to show what is required of the developed platform in terms of features and functions (Agile Business, 2019). The technical planning for these implementations will follow in due course.

The non-functional requirements then look at the performance of the system against certain criteria. For the deep learning model being created here, this largely focuses on acceptance criteria for success rate, as well as the variance in training and testing accuracy and loss to ensure the model is not overfitting. These can also relate to security but as this project is being developed and run locally this does not apply at this stage but is worth keeping in mind during general analysis.

Functional Requirements

- Users can provide an audio file for classification.
- A method for taking audio files and converting them into data structures accepted by a deep learning model.
- A Multi-Layer Perceptron deep learning model which can take formatted audio data as an input.
- Graphing of training and test accuracy of the deep learning model.
- Multiple trained deep learning models for assessment and classification.
- Graphing of each model results to assess accuracy of training and test against one another.

- A method for converting audio files to an understandable data structure for the neural networks on the fly.
- A method for splitting audio files into equal length sound bites for processing by more complex neural networks.
- Application should be able to take a range of different audio file types and convert them on the fly.
- Method for users to change which trained neural network they use to perform classification to assess different models without changing the application.
- Real time audio splitting and monitoring for siren detection.
- Microphone interaction to process uncompressed audio from user's device's microphone.
- Distance detection to assess how far the siren is from the user's microphone.
- Direction detection to assess where the siren is in relation to the user's device.

Non-Functional Requirements

- Program should run on a wide range of devices.
- User interface compatible with non-technical users.
- Results should be classified in all models in under 3 seconds to replicate requirements of a real-world system.
- The test and training data accuracy must be greater than 85% when the model has been trained.
- Application must be tolerant to user error when inputting incorrect file types.
- Application should clearly display errors for quicker debugging purposes.
- Platform should log the results of each classification for review of accuracy over time.

3.4 - MoSCoW Analysis

The next step in the analysis process is to examine, at a high level, what requirements will be needed of the system to satisfy the appropriate stakeholders. An effective way to rank these requirements is to use MoSCoW analysis. This will specify what is needed at each point of development to meet the differing levels of production. The must have requirements specify what is needed to create the minimum viable product (MVP) – "a version of a new product which allows a team to collect the maximum … learning … with the least effort" (Agile Alliance, 2017). In this case the MVP is for a binary classifier to take an audio file and determine if it is a siren or not.

Should have requirements are important features but not strictly necessary in this iteration of development of the product. These features will form the bulk of the project and will examine the different methodologies available for this classification.

Could have requirements are desirable features but are not necessary. These features focus more on user experience and less on the technical implementation of the core project. Thus, these are considered as stretch goals which will only be included in the project if development time allows.

Finally, won’t have requirements are those which will not be included in this iteration of development and for this project, this includes the features which have been relegated due to COVID-19. Although these features would be beneficial for the development of the product, they are not compulsory for the core functionality and are postponed for development to a time of more suitable circumstances.

3.5 - Risk and Response Matrix

With the essence of the project being disrupted early due to COVID-19, risk and contingency planning was crucial to minimise any further disruption to the project. This was especially important due to my lack of development experience in Deep Learning, so contingency plans were required in case the learning rate was not as expected. Furthermore, it was important to plan for the risks in the project to assess the likelihood of them occurring. While some risks were low possibilities which could be largely side-lined, others were extremely high possibility, so the time required to rectify these needed to be taken into account when sprint planning. The risk and response matrix formalised this, allowing for certain deadlines, targets and triggers to be observed to determine whether the pre-agreed contingency plan was required and is available in Appendix 3.

The construction of the risk and response matrix also provided evidence for choosing Agile as the project management methodology. With many of the triggers likely to occur over the course of the project it was important to choose on a methodology where the requirements could change as the risk events were triggered. This weighed heavily in the favour of Agile over Waterfall.

3.6 - Analysis Findings

Analysing the project at a high level, the level of opportunity for initial development in this area and scope for future improvements on the project is clear. The analysis process started in a more abstract manner, looking at the feasibility of the project within the allotted time scale with the aforementioned skills. For this purpose, the SWOT analysis was useful to consider what was required to attempt in the project and what may derail development during the project’s lifecycle.

The next step in the analysis process was to look in detail at the functional and non-functional requirements of the deep learning siren detection system. This process encouraged considerable depth, to look at the scope of complexity in the project and derive deliverable outcomes. In particular, this was useful for educating me on what level of work was required for a minimum viable product, and what the initial steps of development would be. In this instance these requirements coalesced around shaping the audio data, building a neural network and allowing users to input a chosen audio file.

Continuing this process, MoSCoW was used to provide a hierarchy of requirements. This tool gave structure to the development process by cross-referencing the skills mentioned in the SWOT analysis with the requirements to consider what could be created in the development period. While the must haves derived the minimum viable product, what was clear was the importance of the should haves in creating value and coming up with a unique product independent of what has been previously created as explored in the literature review of this report.

As the analysis process concluded by considering the risks and contingencies when developing the system, the focus turned towards the low-level impact which internal and external events may have on project completion within the time period. With COVID-19 looming over the project, this final stage of analysis was particularly crucial. This exercise gave time to think of solutions to issues which were likely to occur in development; in the process, this was educational on the impact the pandemic may have on development of a project of such magnitude. I learnt how important it was to plan around this accordingly. As COVID-19 had already disrupted some opening thoughts on the project, it was useful to consider any future issues which may arise as a result.

The analysis process did include other steps, but these were discounted for this report as they did not provide much value in the instance of this project. For example, a use case diagram was developed but with the deliverable product a simple web application with limited use cases, this diagram did not provide much value in terms of analysis of the project so has not been included in this report.

The analysis section concluded by preparing suitably for the next two stages of the project. Once the areas of opportunity and challenges had been identified, the viability confirmed, and the requirements elicited and ordered, it was important to consider the practical responsibilities of design and implementation. This included extensive research for data sources and resources to aid in the development process, such as textbooks on deep learning frameworks. These are examined more extensively in the following two sections.

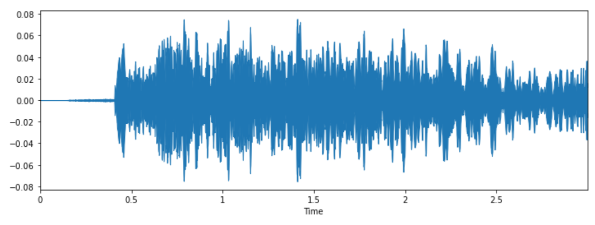
4. Design

The deliverable in terms of what the user will experience for this project is a simple web app where users will be able to select a local audio file, and then receive in text form a classification on whether it is a siren or not. Users will be provided with a dropdown menu to choose which network they would like to use to compute this result. A low fidelity wireframe for this is presented in Appendix 4. Most of this design was predetermined due to the use of streamlit, a tool for building simple python-based web apps for machine learning and data science applications. This is discussed in more detail in the implementation section of this report.

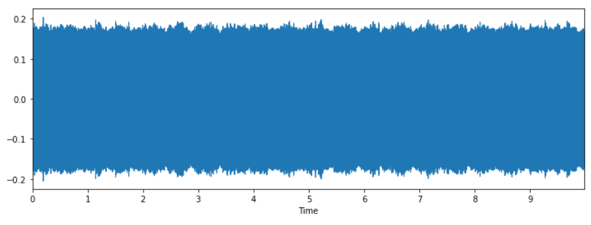
As the user interface design is fairly elementary, this section of the report is dedicated to looking at the most complex areas of design in the project: data design and model design. These two areas are often the most crucial when creating a deep learning application. The data design examines the most effective, reliable and efficient way of translating audio files to a structured data source the model can understand and process. The model design then focused on what each model would consist of and how they would be assembled.

4.1 - Data Design

When the process of building the first model began, a lot of consideration was given to how the data should be arranged when being provided to the model. The first step in this was to perform visual inspections of the waveforms for both emergency and nonEmergency files to look for the differences and similarities in their patterns. The results of these findings are shown below for both a siren and another urban sound file used to train the non-emergency class.



Emergency



Non-Emergency

The layman's observations of these two files posit a radical difference in both the frequency and frequency variations between the two classes. This level of exploration also highlighted the differing sample rates in the data. In audio processing, the process of sampling is the "reduction of a continuous-time signal to a discrete-time signal" (Wikipedia, 2021). Thus, the sample rate is the number of samples taken of the sound each second. When fewer samples are taken, less detail is recorded about the peaks and troughs of the wave. In this case, the sample rate for the non-emergency class was 48kHz whereas the emergency class was 44.1kHz. This knowledge is worth recording for later in development, as the differing sample may cause unfair comparisons in the waves in the neural networks, incorrectly biasing the classifier. To correct this, down sampling is required. Fortunately, there is a python library which, when loading audio files automatically down samples the audio to 22050Hz called 'librosa'. Thus, as this library is used to load all audio files, the sample rate of all files will be set at this level. Librosa also performs some useful background tasks such as normalising the bit-depth rate, the number of bits available in each sample, to a standard range and altering the number of channels of the signal. The detail of these modifications is beyond the scope of this report, but it is worth noting that the consistency provided will be useful when training the model.

Once this initial reconnaissance of the data had been performed, the next stage was to expand on the methodology strategized in the literature review: turning the audio classification problem, into an image classification problem. One way of doing this is to use spectrograms. A spectrogram is a “visual representation of the spectrum of frequencies of a signal as it varies over time” (Wikipedia, 2019). Essentially this means looking at the relationship between frequency and time in audio files and graphing it in a visual format. This will be particularly useful for this project due to the discussion in the literature review showing how duration and frequency are the defining characteristics of an emergency service siren. These visualisations can then be transformed into numerical format to pass to the models for training and evaluation.

While spectrograms were considered initially, this concept was expanded on in practise. Librosa has a module dedicated to taking audio files and creating mel-frequency cepstral coefficients (MFCCs). Again, the mathematics for using MFCCs over spectrograms is beyond the scope of this report, but the reasoning for this decision is not. MFCCs are compact representations of an audio signals spectrum. In MFCCs, the “frequency bands are equally spaced on the mel scale” (Wikipedia, 2019). This lines up much more closely with the human auditory system and MFCCs are commonly used for audio recognition problems. Furthermore, in the paper written by Pawar and Kokate, a sample of research is provided for using different techniques for speech emotion recognition (Pawar and Kokate, 2021). In this research, they discovered MFCCs were “the most widely used technique” as they helped in “reducing harmful contact for speakers, different sentences, speaking styles” and noises. This same logic can be applied when deciding to use MFCCs for this project, where the noisy urban environment must be accounted for. Further detail for how MFCCs were created for each file is provided in the implementation section of the report.

The resulting MFCCs then had to be transformed into a data format which could be passed to and comprehended by the model. In this case, a numpy array is used, to store the values of the MFCC features extracted from the audio file. In turn, each file has this process performed on it, and then is added to a pandas data frame containing the features and appropriate class label.

The data design was developed and adjusted for the convolutional and recurrent models. In these implementations the reduction of the sample rate and threshold detection has been enhanced, where instead of using the default down sampling option provided by librosa, a custom function has been implemented. Here, the librosa resample function is used, taking the sample rate provided by the user and the path to an audio file as the parameters. This implementation is an enhancement for two reasons. Firstly, it is much more compliant to the computational requirements of the system. The audio can be down sampled to a rate where it is not too demanding on the system the model is being trained on while examining as much detail as possible. Secondly, if computational boundaries are not a consideration, the sample rate for the model can be adjusted to the sample rate of the data. For example, if the sample rate across all the data is relatively high, down sampling to such an extent may not be necessary. Ergo, allowing the developer to customise this attribute is useful in multiple use cases.

The next required improvement in the data preparation for the RNN and CNNS is an envelope function. This function tracks how the signal changes over time, and is then used to smooth the curve, outlining the extremes of the oscillating signal. The method for this works in a similar way to a bubble sort, using a rolling window to look at a section of the data, and calculating the mean, appending it to an array and moving the window along. This works to minimise the extremes in the signal, which may cause bias in the classifier. This function is also used for threshold detection, which is a method for removing dead space in the audio. The function ensures that the calculated mean value is above a certain level. If it is below the threshold level, it is not included. This ensures the classifier does not relate the silence to a certain class, altering the resulting classification.

In the more advanced implementation, the duration of the audio files is important. The Recurrent Neural Network has a unique architecture making it well suited to time specific problems. When data is provided to an RNN, each sample must be of the same size. For this, a function is used to split each audio file into a specific delta time provided by the user. In the case of the trained and evaluated models of this project, a delta time of 1 second is suggested. As the data is being split it is saved in a new folder called 'clean'.

The final area of data design differing in the CNN and RNN compared to the MLP occurs in the model itself. Instead of extracting all features before the data is passed to the model, here the library Kapre allows for this to be performed on the fly. The first layer of the model is called 'get_melspectrogram_layer' which converts the provided audio file into a melspectrogram on the fly, for each audio file being passed to the model. This reduces the pressures on the CPU and abstracts some of the complexity of preparing the data away from the user. Further design decisions were taken when creating the four types of models. These are covered in more detail in the implementation section of this report.

4.2 - System Overview

When designing the system, it was important to consider how the different versions would contribute and combine to the overall mission. The purpose of the MLP model was largely to show the feasibility of the project and guarantee a minimal viable product reasonably early in development. This was developed on Jupyter Notebooks and was suitably isolated from the rest of the system.

The other three models were much more at the core of the systems architecture. Firstly, a file independent of the models was used to clean the data, down sample the audio files, split them into 1 second delta times and reduce any dead space in the audio. The models themselves were stored as functions in a seperate python file, with each function returning the compiled model when called. After the data had been cleaned and split, the train function could be run, which batched the data and sent it through a user determined model for training. This file also contains a file for plotting accuracy and loss as well as saving the model and its trained weights. The saved 'h5' file, can then be used independently of the rest of the code in any required application. In the case of this project, it is loaded in a python file that specifies the layout and functionality of the streamlit web app. Further information on the system layout is available in the implementation section of the report.

5. Implementation

5.1 - Tools, Libraries and Technologies

When the project was initially envisioned, the dependence on libraries was a clear prerequisite. A vast amount of research and expenditure has been plunged into the deep learning sphere recently, and with the complexity of development and level of mathematics required to develop a model from scratch, using a deep learning library was an obvious solution.

The first step in deciding on tools and technologies was to decide on a programming language. Some initial consideration was given to JavaScript, with the application eventually being deployed in the form of a web app, as well as Swift and Kotlin for their mobile first nature. However, these options had several drawbacks such as the specialisation they had in their respective development areas: web and mobile development. However, after some elementary research it was clear python was the best option. The vast number of libraries supporting the language as well as the procedural nature of its implementation meant it would be quick to develop useful code that could run on a range of devices.

Once this was determined the next step was to conclude on a deep learning framework. There are a number of frameworks available with a lot of them compatible with python. As this was to be my primary foray into deep learning, extensive documentation and widespread adoption was an obvious requirement. While considering MXNet, Caffe2 and CNTK, the clear candidates were TensorFlow and PyTorch. Eventually, the granularity available with TensorFlow as well as industry leading publications such as 'Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow' by Aurélien Géron available to aid development, TensorFlow was a clear winner.

Further libraries were required for data preparation and analysing the results. Numpy and Pandas were used for transforming and formatting the data into a structure which can be passed to the model. Numpy arrays have a distinct advantage over python lists as they can be n-dimensional; a feature suited to the challenge of deep learning. MatPlotLib has been used for graphing the results, plotting the test and training data for accuracy and loss across each epoch.

To manage all these packages, anaconda is being used to create a virtual environment. Appendix 5 shows a list of all of the packages installed in the environment. The development environment for the Conv1d, Conv2D and LSTM models is Visual Studio Code, but for the initial exploration of the data, and to build the first model, the multi-layer perceptron, Jupyter Notebooks was used.

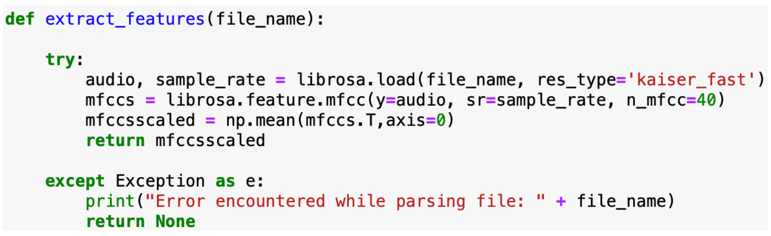
5.2 - Implementation of the Deep Learning Application

As aforementioned, the workflow for developing the different models was format the data correctly, build the model, pass batches of the data through the model to train the weights, record the training and test results and then deploy the model into the web application. With this in mind, the following section goes through each of these steps and explains the coding principles, sophistication and structure in relation to each type of model.

Furthermore, any significant challenges that were experienced in development are discussed here.

5.2.1 - Prepare the Data

For the MLP model, the preparation of the data was largely done in two steps. Firstly, a function was written to extract the features and create MFCCs as numpy arrays for a provided file path to an audio file. This function is shown in figure 5.1 below. Here, a try except statement catches errors in incorrect file types. This function is then looped through for all audio files used as shown in figure 5.2.

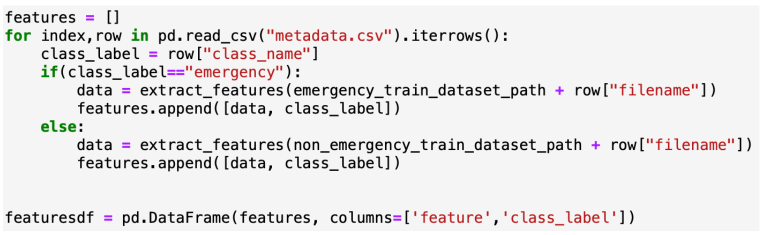


```python
def extract_features(file_name):

    try:
        audio, sample_rate = librosa.load(file_name, res_type='kaiser_fast')
        mfccs = librosa.feature.mfcc(y=audio, sr=sample_rate, n_mfcc=40)
        mfccsscaled = np.mean(mfccs.T,axis=0)
        return mfccsscaled

    except Exception as e:
        print("Error encountered while parsing file: " + file_name)
        return None
```

Figure 5.1

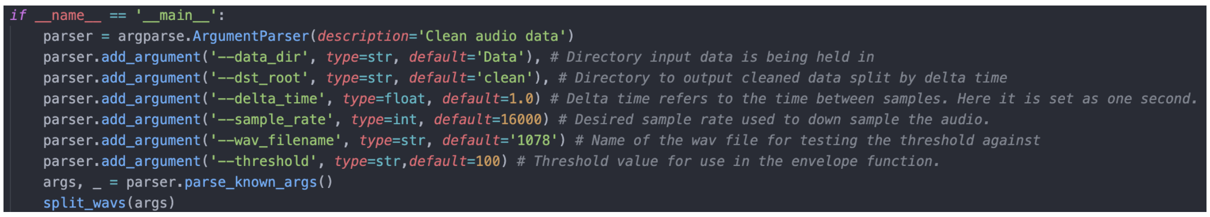


```python
features = []
for index,row in pd.read_csv("metadata.csv").iterrows():
    class_label = row["class_name"]
    if(class_label=="emergency"):
        data = extract_features(emergency_train_dataset_path + row["filename"])
        features.append([data, class_label])
    else:
        data = extract_features(non_emergency_train_dataset_path + row["filename"])
        features.append([data, class_label])


featuresdf = pd.DataFrame(features, columns=['feature','class_label'])
```

Figure 5.2

This contrasts significantly with the other three models, where much more granularity was performed in data preparation. For each of these models, the main business logic occurs in the 'split_wavs' function, where each wav file is looped through and formatted correctly. This involves calling separate functions to down sample, reduce the noise and remove dead space. Each sample is split into the delta time parameter of one second and saved in a new file under the appropriate class name once the transformations have taken place. The greatest benefits of this version of the system are the heightened level of customisation to user requirements. System owners can determine a threshold value, delta time or sample rate most suited to the input data. This is done by using an argument parser to provide the 'split_wavs' function with all of the user's requests. This allows for greater code reusability and modularity. Figure 5.3 shows the argument parser set and passed to the function when the file is being run.



```python
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Clean audio data')
    parser.add_argument('--data_dir', type=str, default='Data'), # Directory input data is being held in
    parser.add_argument('--dst_root', type=str, default='clean'), # Directory to output cleaned data split by delta time
    parser.add_argument('--delta_time', type=float, default=1.0) # Delta time refers to the time between samples. Here it is set as one second.
    parser.add_argument('--sample_rate', type=int, default=16000) # Desired sample rate used to down sample the audio.
    parser.add_argument('--wav_filename', type=str, default='1078') # Name of the wav file for testing the threshold against
    parser.add_argument('--threshold', type=str,default=100) # Threshold value for use in the envelope function.
    args, _ = parser.parse_known_args()
    split_wavs(args)
```

Figure 5.3

One technical challenge regularly haunted the project when preparing the data for use in the model and resulted in changes in the development process. It was first observed when attempting to train the MLP model, where an error message would suggest the input shape of the data was different to the one required by the model. However, when outputting the shape of an index of the data, it appeared to align. After significant debugging, I performed an examination of the shape of each audio file which had been converted by looping through the pandas data frame. The code used for this debugging is available in Appendix 6. The result showed that one file had caused an exception, and 'None' was added to the array when extracting features instead of the numerical MFCC. Upon further investigation, the file causing the issue was a hidden '.DS_Store' file. This is a file unique to macOS which 'stores custom attributes of its containing folder' (Wikipedia, 2019). As it was a hidden file it was not showing up in the debugging or testing process.

There were two implemented solutions for this. Firstly, remove the file from the folder containing the data using the command 'rm -f .DS_Store'. This resolved the issue temporarily, but if adding new data to the file, changing the file system in future or attempting to run the program on a different machine the issue would reoccur. For this reason, multiple checks were put in place throughout the system where the hidden file may cause issues. Examples of these checks are available in Appendix 6.

5.2.2 - Build the Models

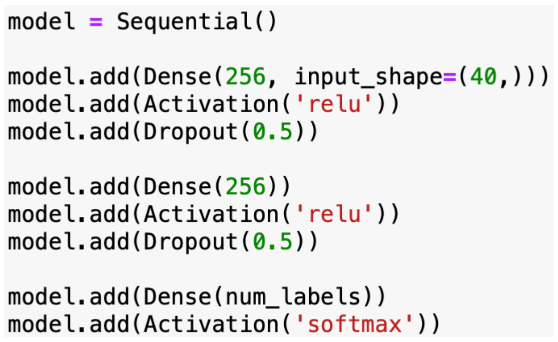
MLP

In the MLP model, the structure is relatively simple with a sequential model containing an input layer, a hidden layer and an output layer. A sequential model is commonly used in deep learning where there is one input and one output for each layer. In this model, the output of one layer is directly fed into the next.

The first layer is the input layer. As 40 MFCCs are created for each sample, the input shape for the layer is 40. There are 3 composite parts to this layer. Firstly, the dense part specifies the size of the layer to be 256 neurons. The second part of this layer is the activation function, which is specified here as the Rectified Linear Unit function – a common activation function for deep learning classification problems. The final part is a dropout, which randomly excludes nodes from each epoch in order to create a better generalisation and thus result in less overfitting. This ensures a particular feature of the MFCCs is not fixated on for classifications.

The second layer is a hidden layer and repeats the same structure as the first. As these are dense layers each neuron in the previous layer is mapped to a neuron in this layer.

The final layer is the output layer with two nodes: one for emergency and another for non-emergency. The activation function in this layer is softmax, meaning the sum of all the nodes will add up to one. This is a useful activation to use for probability problems and in practise will provide a percentage chance for each class given an audio file. A summary of this model is available in figure 5.4.



```
model = Sequential()

model.add(Dense(256, input_shape=(40,)))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))

model.add(Dense(num_labels))
model.add(Activation('softmax'))
```

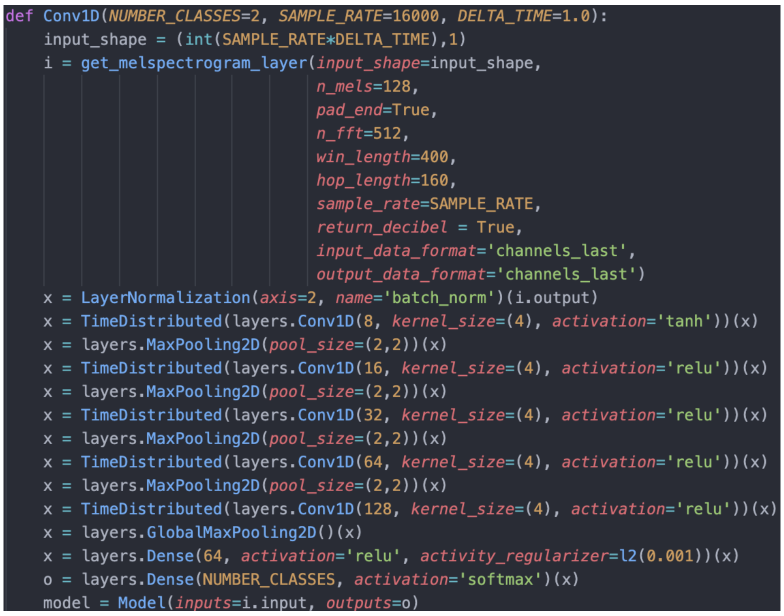
Figure 5.4

Conv1D

The next model created was the 1-dimensional convolutional neural network (conv1d). The first layer in the model creates the mel-spectrograms using the Kapre library. A full description of the arguments provided and required for this layer is available in the Kapre documentation (Choi, 2021), but it is important to note that here the sample rate, input shape and number of mel-spectrograms per sample are provided. As the data is being prepared here, a normalization layer is required to provide a uniform scale for numerical values. This takes the output of the 'get_melspectrogram_layer' as an input and normalizes it for use in the model.

There are then 5 hidden layers with similar structures but slightly differing characteristics. They all firstly contain a 'TimeDistributed' wrapper around a 1-dimensional convolution. In Keras, the API beneath TensorFlow, this wrapper is used to apply the 1d convolutional layer to each 'temporal slice of an input' (Keras, 2021). Put simply, convolutional neural networks are useful for finding patterns in data, thus they can be extremely effective for use on mel-spectrograms. The 'TimeDistributed' wrapper ensures that the time dimension is also used when examining and looking for trends in the patterns. This is an oversimplification of this function of the network, but it serves the purpose of presenting how these layers of the network work in unison. The wrapped 1d-convolution doubles the number of neurons in each layer, starting at 8 and ending with 128 on the final layer. This ensures the model starts by learning general patterns and becomes more specific as the model progresses, thus reducing the possibility of overfitting. In the first layer, the hyperbolic tangent activation function is used as alternative activations would be too strict at this stage where the neuron count is so low. In the other 4 layers the Rectified Linear Unit activation function is used. Each convolutional layer is then followed by a max pooling layer. Max pooling is used to 'reduce the dimensionality of images by reducing the number of pixels in the output from the previous convolutional layer' (Anon, 2021). This is useful primarily for reducing the possibility of overfitting and encouraging generality in pattern discovery, as well as reducing the computational requirements.

The final two layers of the model are used for the output. Firstly, a 64-neuron dense layer is used as a regularizer. Regularizers are methods to reduce complexity in a deep learning model, and thus is often used as a method in convolutional networks to avoid overfitting. Finally, a dense output layer uses the softmax activation function to output the probability for each class matching the input file. The layout of the model and its construction is available in figure 5.5.



```python
def Conv1D(NUMBER_CLASSES=2, SAMPLE_RATE=16000, DELTA_TIME=1.0):
    input_shape = (int(SAMPLE_RATE*DELTA_TIME),1)
    i = get_melspectrogram_layer(input_shape=input_shape,
                                 n_mels=128,
                                 pad_end=True,
                                 n_fft=512,
                                 win_length=400,
                                 hop_length=160,
                                 sample_rate=SAMPLE_RATE,
                                 return_decibel = True,
                                 input_data_format='channels_last',
                                 output_data_format='channels_last')
    x = LayerNormalization(axis=2, name='batch_norm')(i.output)
    x = TimeDistributed(layers.Conv1D(8, kernel_size=(4), activation='tanh'))(x)
    x = layers.MaxPooling2D(pool_size=(2,2))(x)
    x = TimeDistributed(layers.Conv1D(16, kernel_size=(4), activation='relu'))(x)
    x = layers.MaxPooling2D(pool_size=(2,2))(x)
    x = TimeDistributed(layers.Conv1D(32, kernel_size=(4), activation='relu'))(x)
    x = layers.MaxPooling2D(pool_size=(2,2))(x)
    x = TimeDistributed(layers.Conv1D(64, kernel_size=(4), activation='relu'))(x)
    x = layers.MaxPooling2D(pool_size=(2,2))(x)
    x = TimeDistributed(layers.Conv1D(128, kernel_size=(4), activation='relu'))(x)
    x = layers.GlobalMaxPooling2D()(x)
    x = layers.Dense(64, activation='relu', activity_regularizer=l2(0.001))(x)
    o = layers.Dense(NUMBER_CLASSES, activation='softmax')(x)
    model = Model(inputs=i.input, outputs=o)
```

Figure 5.5

Conv2D

The two-dimensional convolutional model does not differ radically from the one-dimensional version, but it is important to consider the variations in the two models. In a 2-dimensional Convolutional Neural Network (CNN), the kernel slides along 2 dimensions of the data, thus considering features in a more complex manner. A graphical representation of this is shown in figure 5.6. As this kernel moves in two directions, 2D CNNs are particularly suited to image data.

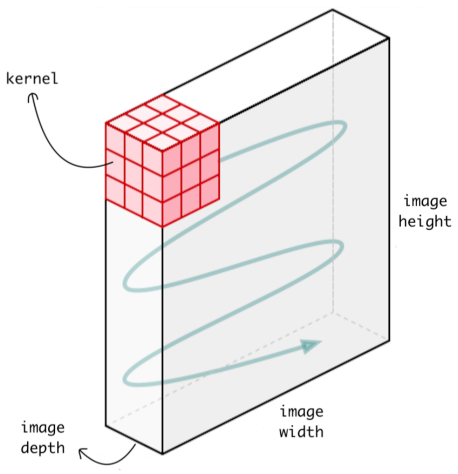
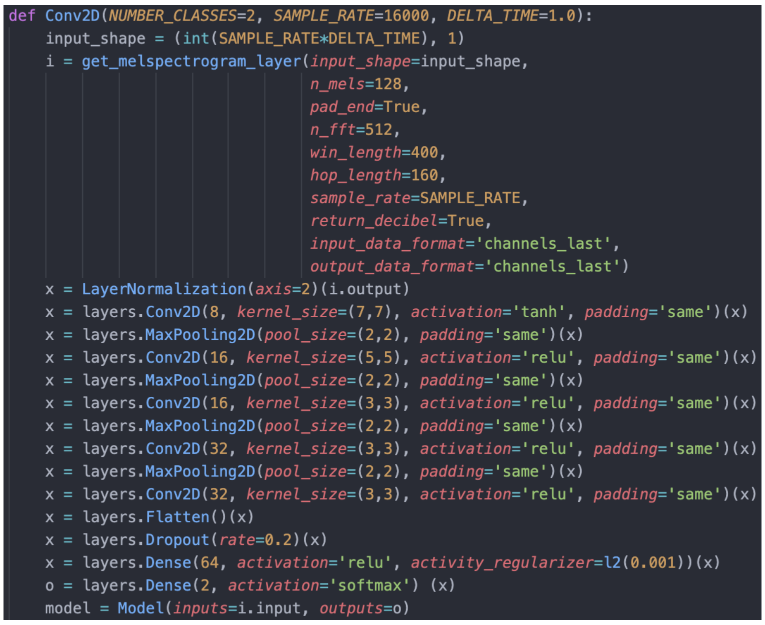


Figure 5.6 (Little, 2020)

The 1-dimensional CNN kernel slides along one dimension and moves in one direction and thus is particularly suited to time series data but is not as strong at finding patterns in image. The audio classification problem here sits within the specialties of these two models. The data is time series dependent but the method of converting the wav files to mel-spectrograms creates an image classification problem. It will be important to analyse the results of these two models in relation to one another.

The model’s construction is similar to the one-dimensional version, starting with the Kapre ‘get_melspectrogram_layer’. Following this, a conv2d layer is succeeded by a max pooling one to reduce the possibility of overfitting. As the model works in two-dimension, a flattening layer is required after the final convolutional layer for it to fit with the remaining output layers. This essentially works by translating the shape of the data in the model into a one-dimensional array. After a dropout layer to improve generalization, the same two dense layers as in the one-dimensional version are used for output. The compilation of the model is shown in figure 5.7.



```python
def Conv2D(NUMBER_CLASSES=2, SAMPLE_RATE=16000, DELTA_TIME=1.0):
    input_shape = (int(SAMPLE_RATE*DELTA_TIME), 1)
    i = get_melspectrogram_layer(input_shape=input_shape,
                                 n_mels=128,
                                 pad_end=True,
                                 n_fft=512,
                                 win_length=400,
                                 hop_length=160,
                                 sample_rate=SAMPLE_RATE,
                                 return_decibel=True,
                                 input_data_format='channels_last',
                                 output_data_format='channels_last')
    x = LayerNormalization(axis=2)(i.output)
    x = layers.Conv2D(8, kernel_size=(7,7), activation='tanh', padding='same')(x)
    x = layers.MaxPooling2D(pool_size=(2,2), padding='same')(x)
    x = layers.Conv2D(16, kernel_size=(5,5), activation='relu', padding='same')(x)
    x = layers.MaxPooling2D(pool_size=(2,2), padding='same')(x)
    x = layers.Conv2D(16, kernel_size=(3,3), activation='relu', padding='same')(x)
    x = layers.MaxPooling2D(pool_size=(2,2), padding='same')(x)
    x = layers.Conv2D(32, kernel_size=(3,3), activation='relu', padding='same')(x)
    x = layers.MaxPooling2D(pool_size=(2,2), padding='same')(x)
    x = layers.Conv2D(32, kernel_size=(3,3), activation='relu', padding='same')(x)
    x = layers.Flatten()(x)
    x = layers.Dropout(rate=0.2)(x)
    x = layers.Dense(64, activation='relu', activity_regularizer=l2(0.001))(x)
    o = layers.Dense(2, activation='softmax') (x)
    model = Model(inputs=i.input, outputs=o)
```

Figure 5.7

LSTM

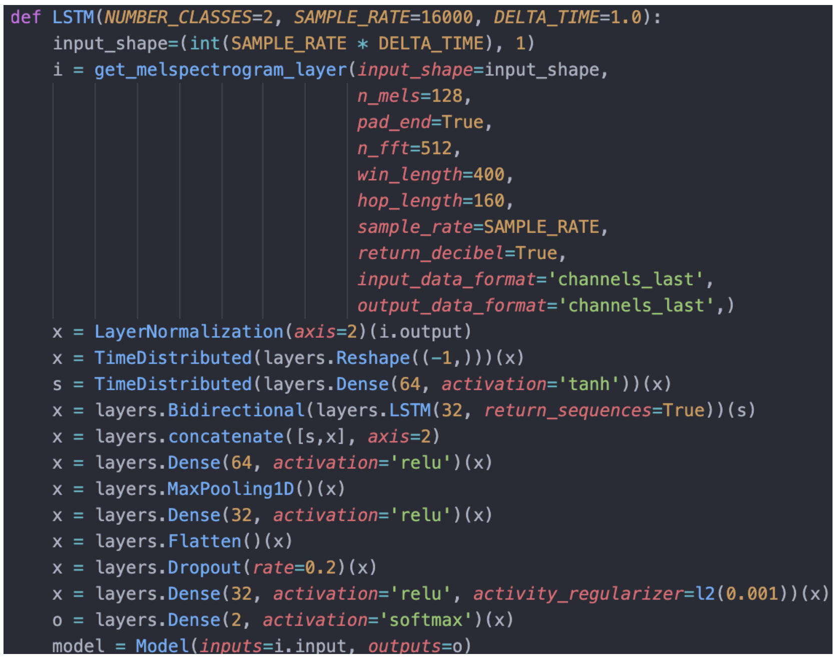
LSTM (Long Short-term Memory) model is a type of Recurrent Neural Network (RNN). RNNs are specifically designed to work with temporal and sequence data. The construction and compositions of the gates and cells of a LSTM model is beyond the scope of this report, but it is important to be aware of the strength of LSTMs in classifying time series data. Crucially, they are able to consider short term patterns in scale to long term signals and can consider how features change over time instead of looking at them in isolation. LSTM’s have become a gold standard for a lot of deep learning problems recently, with both OpenAI and DeepMind openly advocating for their use in progressing towards Artificial General Intelligence.

The LSTM used in this project follows a relatively simple structure, starting with the familiar Kapre function and layer normalization. The data structure required for the LSTM is smaller than for the convolutional networks, so a reshape layer is used to remove the channel dimension created by Kapre. Initially in development this layer was absent, and it caused multiple issues with dimensions not fitting correctly with the LSTM network. Logging the dimensions of the data at each stage was fruitful in identifying and resolving the issue.

Commonly in LSTM networks for audio problems, a dense layer wrapped in a 'TimeDistributed' wrapper is combined with the LSTM to prevent overfitting, so the next layer in the network is a dense layer with a hyperbolic tangent activation.

The LSTM itself is wrapped in a Bidirectional wrapper. This allows the LSTM to look forward and backwards in time and has been proven to have industry leading results in audio recognition problems (Graves et al, 2013). A concatenate layer then binds this with the Dense layer.

The remaining layers build towards the output layer, by using a dense layer with a maxpooling layer to prevent overfitting, followed by a dense layer and a flatten layer. The number of neurons in the two dense layers are 64 and 32 respectively. Finally, a dropout layer is followed by a dense layer with a regularizer and the output layer with two neurons. The compilation of the layer is shown in figure 5.8.



```
def LSTM(NUMBER_CLASSES=2, SAMPLE_RATE=16000, DELTA_TIME=1.0):
    input_shape=(int(SAMPLE_RATE * DELTA_TIME), 1)
    i = get_melspectrogram_layer(input_shape=input_shape,
                                 n_mels=128,
                                 pad_end=True,
                                 n_fft=512,
                                 win_length=400,
                                 hop_length=160,
                                 sample_rate=SAMPLE_RATE,
                                 return_decibel=True,
                                 input_data_format='channels_last',
                                 output_data_format='channels_last',)
    x = LayerNormalization(axis=2)(i.output)
    x = TimeDistributed(layers.Reshape((-1,)))(x)
    s = TimeDistributed(layers.Dense(64, activation='tanh'))(x)
    x = layers.Bidirectional(layers.LSTM(32, return_sequences=True))(s)
    x = layers.concatenate([s,x], axis=2)
    x = layers.Dense(64, activation='relu')(x)
    x = layers.MaxPooling1D()(x)
    x = layers.Dense(32, activation='relu')(x)
    x = layers.Flatten()(x)
    x = layers.Dropout(rate=0.2)(x)
    x = layers.Dense(32, activation='relu', activity_regularizer=l2(0.001))(x)
    o = layers.Dense(2, activation='softmax')(x)
    model = Model(inputs=i.input, outputs=o)
```

Figure 5.8

5.2.3 - Train the Models

The next step in the development process was to train the models. For the MLP model, this involved determining the optimum batch size and number of epochs. The batch size is how many samples are passed through the network before it is trained. Choosing a suitable batch size is important for minimising computational requirements and thus allowing the network to train faster. The number of epochs relates to how many passes of the training data to go through when training the model. This is often decided by the user and should correlate to the number of passes required for accuracy to improve before stabilising. After some experimentation on the validation dataset, a batch size of 32 with 30 epochs was decided upon.

For the remaining three models, the focus was on code reusability, so that the same code could be used for all three models, only changing the parameter for which model to train.

The user sets the key parameters to be passed to a ‘train’ function. These parameters include the sample rate, delta time and a string to specify which model to change. This allows for the code to be reusable and easier to maintain. The train method then creates two instances of the DataGenerator class: one for training data and one for validation data. The DataGenerator class inherits from the TensorFlow sequence class and is used to batch the data and prepare it for training. This is a requirement of using Kapre so that the data can be fed to the model in real time. After training, the test and training accuracy was recorded and is shown for each model below.

MLP:
Training Accuracy – 94.1%
Testing Accuracy – 91.5%

Conv1D:
Training Accuracy – 99.2%
Testing Accuracy – 95.5%

Conv2D:
Training Accuracy – 98.2%
Testing Accuracy – 95.8%

LSTM:
Training Accuracy – 98.5%
Testing Accuracy – 96.5%

5.2.4 - Save the Models and Plot the results

The final two steps in the development of the models is to save them and their weights and plot the results to further evaluate their performance. The models and trained weights are saved after every epoch in a h5 file. After the model has been fit to the data, it is passed to a function called ‘plotting’. Here the key values for plotting loss and accuracy are retrieved and output as a line graph in Matplotlib.

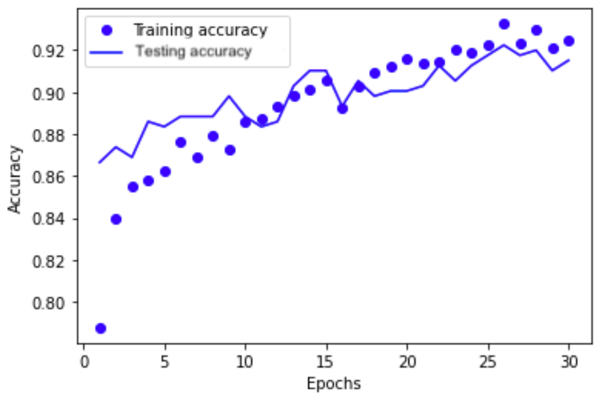
5.3 - Creating the Output

The next step in the software development process was to create the output software that a user could interact with. To fulfil the requirements outlined in the analysis phase, it was important that a user could upload a wav file, have it converted into a format suitable for the models, and output the result of the classification showing the user the determined class. For this the streamlit software was used. Streamlit is an open-source tool allowing users to create simple platforms for data science and machine learning programs. The python file had a function used for applying the down sample and envelope function to the user’s wav file and cleaning it appropriately. The trained model is then used to predict and assign the file to a class which is then displayed to the user. Furthermore, a dropdown list meets the requirement allowing users to choose which model to use for the classification.

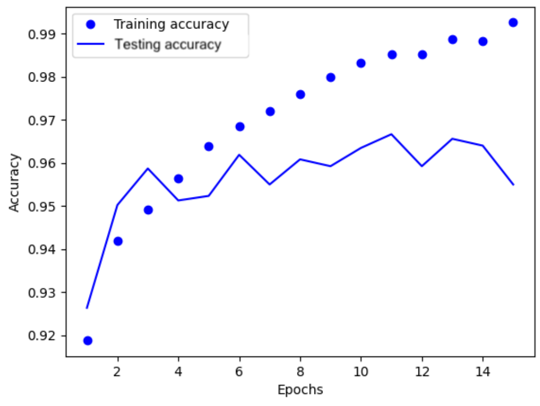
5.4 – Analysing Performance

The primary aim of this project was to build four different methodologies for deep learning to look at which is most effective as a solution, not just to create one working implementation. With this in mind, it is important to assess the performance of each model and analyse the limitations and qualities of each. It is worth considering the amount of data the models were trained with. In total there were 685 emergency files and 1004 non-emergency files. These were then split in a 60/20/20 model, where 60% of the data was used for training, 20% for testing and 20% for validation. Although this is enough data to accurately train the models, more data being available would benefit the more complex convolutional and recurrent models.

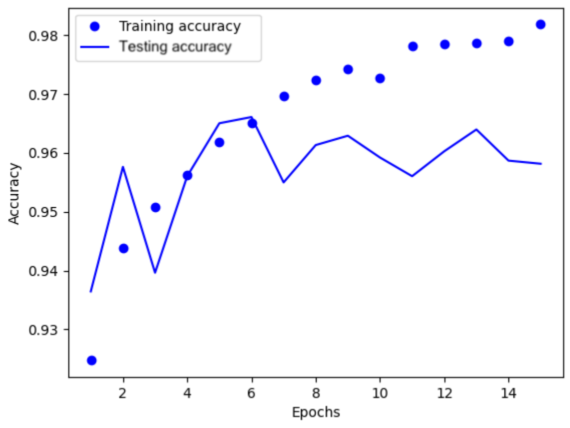
The results conform generally to what is expected in relation to the complexity of the models. The MLP model performed worst with 91.5% accuracy on the testing data and the LSTM performed best on the test data with accuracy of 96.5%. While all models performed well, as this is a safety critical project the high accuracy of the LSTM network is particularly promising. The graphs below show the improvements made for each model in terms of accuracy on the training and testing during each epoch of training.



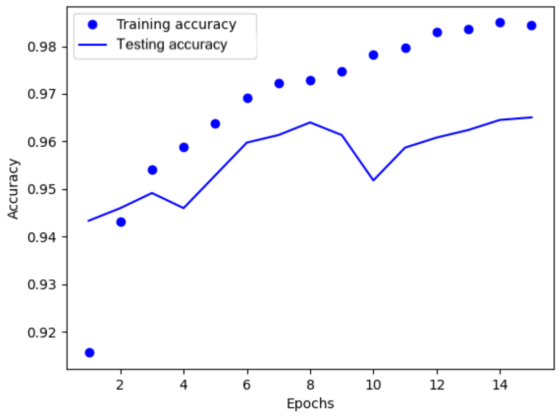
MLP



Conv1d



Conv2d



LSTM

5.5 - Implementation Extensions

As previously mentioned in this report, multiple development expansions were hypothesised earlier in the project, but their implementation was rendered mute due to the COVID-19 pandemic. However, some thought has been given to these features and how they could be implemented in the future. Realtime audio detection would be entirely possible and a useful expansion of the project. This could be done by expanding on the threshold detection concept to split the audio into delta time at points which permeate above a certain threshold. Python libraries like sounddevice and pysound could aid in the development of this to allow for real time microphone input.

Direction and distance detection could also be implemented on top of the developed models. The greatest challenge in this area would be in the collection of binaural data for use in development, but the trained models will be a starting block for detecting when a siren is in the vicinity.

6. Validation

Introduction to the Testing Strategy

Software testing is crucially important when developing a new system. Not only does it ensure quality delivery and customer satisfaction, but it also saves time and money later in the development process when issues have been caught and prevented early on. In this project in particular, testing is of unique importance as the system is safety critical. If the system were to fail it could cause harm or injury to the user relying upon it, thus violating the safety aspect of dependable system properties.

In this project, two types of testing have been performed, the first focusing on software quality, reliability and adaptability, and the second focusing on user requirements and acceptance.

Unit Testing

For the unit testing, the unittest module in Python was used. The MLP model developed in Jupyter notebooks was incredibly modular, with each code snippet running independently making it difficult to test. Furthermore, the Jupyter notebooks code focused heavily on initial data exploration and development so although it is possible to unit test in Jupyter notebooks, this code was not included in the testing here, with the focus being on the other 3 modules.

In particular, the unit testing centred around adaptability in the code to changing requirements and user needs. For example, the tests performed looked at the versatility of changes in the number of classes or the desired down sampling rate. The testing was also used to verify the success of the code in meeting the specified user requirements, such as the down sampling of audio files and the accuracy of predictions. A full breakdown of all tests performed, and their results, are available in Appendix 7.

User Testing

As expected, user testing was limited due to COVID-19. As the application is hosted locally and is not accessible via the internet, user testing could only be performed by people on the same network, so thus far user testing has only involved one individual with no knowledge of the project's development. This does not cause major issues for the development of the project as previously specified the web application version of the system is not the targeted deliverable when in production. That being said, it is important to gauge the usability of the platform as this will be used as a proof of concept for further development. The results of this testing are available in Appendix 7.

7. Critical Review and Conclusion

Critical Review

The value of the project has been discussed at length throughout this report. Not only will it be useful for technological development in the field of autonomous vehicles but also for current drivers with hearing impairments. While the original scope was adjusted due to the presence of COVID-19, the adapted aim of comparing and contrasting four types of neural networks to look at which deep learning solution is best for the project at hand has been satisfied. The conclusion of this report will consider the project and its results in relation to the wider field, but there is value in examining the methodology and development approach to look at the successes and shortcomings in the implementation of the objectives.

The first discussion point worth examining is the use of agile as a project management technique. This had clear advantages throughout the project, especially when requirements changed due to COVID-19. As discussed earlier in this report, development experience in deep learning frameworks was limited, and thus having more leeway and liberty in the time permitted to complete each stage of development was crucial. The agile methodology focused on allowing me to develop working software, instead of unnecessary documentation. This suited the deliverable deadlines for the project well and allowed for quicker turnaround times in software development. The ability to craft sprints around pockets of development was particularly helpful. For example, the first sprint looked at data exploration and visualisation, with subsequent sprints focussing on each model's development and then the development of the platform. This suited the timescale of the project well and allowed for more time to be allocated to more challenging tasks.

The use of agile did have some drawbacks in the development process. Firstly, a lot of the benefits from the agile processes come from collaboration when working in a team. As this was an individual project, not having a team barred some of the scrum ceremonies from operating, such as the use of stand ups. These ceremonies would have been extremely helpful were the project a group task and would have sped up development of required functionality allowing more time to focus on some of the stretch goals. The problem of not collaborating in a team was often overcome with extra research performed on areas of complexity but being able to discuss challenges in a stand up would have resolved them more effectively and faster.

The common problem of progress measuring in agile wasn't much of a concern, as it was simple to tick off each stage of the requirements when they were completed, but it was challenging to determine an end point for the project. Needless to say, there are many other types of deep learning models which could be implemented and tested for their effectiveness in solving the problem. The four implemented here cover good ground for an examination of the key models and offer diversity in their capabilities but for a more comprehensive analysis, more methodologies would need examining, including machine learning methods. On the other hand, agile did permit scope expansion such that more 'could have' requirements could be added to the development stage as progress went on. For example, the development of the four models ahead of the allocated time scale permitted the development of the streamlit web application.

There were other changes to the development process which would be considered were the project to be run again or continued in future. Most prominently, test driven development would be implemented. In this edition, the mistake was made to leave testing until later on in the development process. This made it more challenging to write tests for all of the code as well as make changes to the code that the testing prompted. To resolve this issue in future, the functional test for a feature would be written prior to the writing of the code to perform the function. This would fit with the agile methodology of incremental development and would also help to ensure the code is modular and can accommodate the changing user requirements. While this may increase the amount of time required in development, it would be a more comprehensive and secure approach, as well as saving time in maintenance over the products lifecycles, as several studies have shown (Elliott, 2019).

Finally, it is valuable to review the future of the project. There are shortcomings in the project, some previously discussed relating to COVID-19, while others arising from the available time scale and complexity of development. Development could certainly be enhanced by the creation of other types of models and comparing these results in relation to other machine learning methods, and an increase in the available training data would likely result in increased performance. This data could be acquired through recording or further online research of datasets.

Many expansionary features are already examined in this report, but it is important to also look at the deployment of the service in its planned environment: passenger vehicles. Information researched in the literature review pointed at the availability of microphones already built into vehicles, but for future development of the project it would be important to examine how the model's software could be implemented here and what the most appropriate method would be to display the results to a user. This would need to occur next in the development process as it would have an impact on the methodology taken for the real time audio monitoring.

These changes to the project management approach and possible future features in the undertaking are exciting and of worthy consideration. However, at this stage of the report, it is worth assessing the discoveries and results found in the implementation.

Conclusion

In conclusion, after data discovery, analysis and visualisation, four deep learning models with differing architectures were crafted. Each of these models has been assessed and evaluated to conform and complete requirements specified in the analysis process to examine the best model for the task of emergency service siren detection in urban environments. Furthermore, a minimum viable product platform has been created, allowing users to test recorded audio files against each of the binary classifiers.

As discussed in the literature review, this project is a necessary cross section of many different fields, and thus published results for this exact specification are not openly available. Companies such as waymo and zf, both of whom have expressed interests in this area, have not released accuracy results and academic papers, such as that written by

Meucci et al have also not published comparable results (Meucci et al, 2008). It is possible to consider accuracy against other classifiers in differing environments, and Mike Smales (Smales, 2020) approach which provided inspiration for the MLP classifier is a good example of this. For his paper, a 2-dimensional convolutional neural network was developed to classify the UrbanSound8k dataset. The training and test results of this are shown below, demonstrating that all four models developed for this project sit in a similar range, with the more advanced models providing a higher level of accuracy on unseen data.

Training Accuracy: 98%
Testing Accuracy: 91%

As, due to COVID-19, requirements for the project had to be adjusted fairly early in the development process, some of the original goals as mentioned in the project proposal have not been met, including real time audio monitoring and direction and distance detection. However, almost all of the revised requirements for the project have been met. Throughout the development process, lots has been learnt technically in the deep learning sphere and in software project management to help in future development. There is significant scope for future development of the project and it makes a useful and important contribution to the field.

Appendices

Appendix 1

<u>Strengths</u>	**<u>Weaknesses</u>**
• Significant python experience. • Experience with Agile methodologies so suited to adjusting requirements. • Experience with Python libraries such as numpy. • Knowledge of managing Python projects with significant dependencies using miniconda and pip.	• Lack of deep learning project experience. • Individual project so no team to share workload and aid in development. • Short timescale for development will need to be split between learning and development. • Deliverables in other areas may hinder progress.
<u>Opportunities</u>	**<u>Threats</u>**
• Deep learning boom has created a lot of learning materials for deep learning. • Frameworks like TensorFlow and PyTorch, abstract the mathematics of deep learning from the user. • Use of other audio classifiers on GitHub can be used as templates to retune the weights and models to aid in development and learning.	• Lack of deep learning experience may derail development. • Lack of the type of data required to train, test and evaluate the model. • COVID-19 pandemic may hinder accessibility at unknown times. • While no published methodology for the project is available, research suggests competitors are working on similar systems.

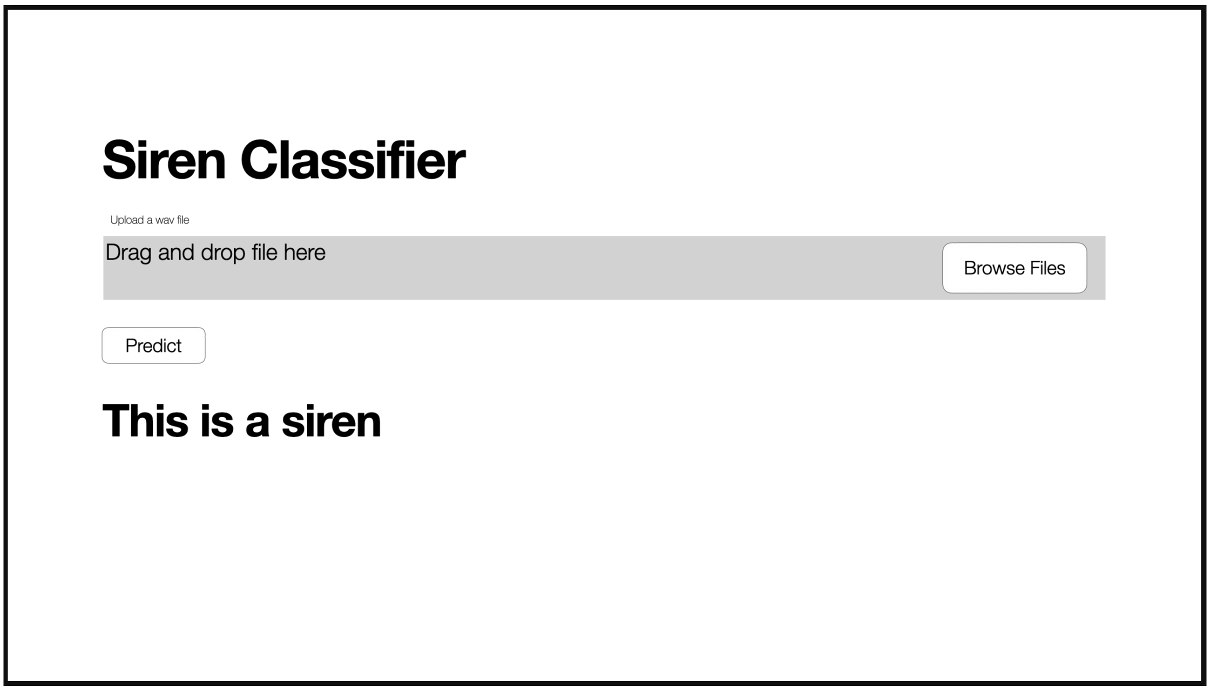
Appendix 2

Must Have	• Users can provide an audio file for classification. • A method for taking audio files and converting them into data structures accepted by a deep learning model. • A Multi-Layer Perceptron deep learning model which can take formatted audio data as an input. • Graphing of training and testing accuracy of the deep learning model.
Should Have	• User interface compatible with non-technical users. • Multiple trained deep learning models for assessment and classification. • Graphing of each model results to assess accuracy of training and testing against one another. • A method for converting audio files to an understandable data structure for the neural networks on the fly. • A method for splitting audio files into equal length sound bites for processing by more complex neural networks.
Could Have	• Program should run on a wide range of devices. • Application should be able to take a range of different audio file types and convert them on the fly. • Method for users to change which trained neural network they use to perform classification to assess different models without changing the application.
Won't Have	• Real time audio splitting and monitoring for siren detection. • Microphone interaction to process uncompressed audio from user's device's microphone. • Distance detection to assess how far the siren is from the user's microphone. • Direction detection to assess where the siren is in relation to the user's device.

Appendix 3

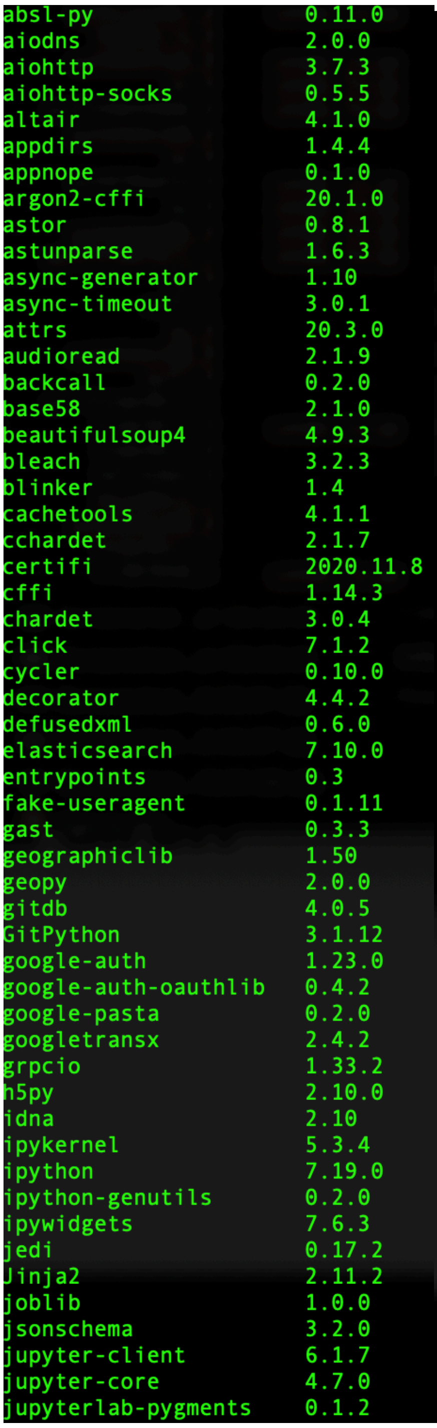
Risk Event	Response	Contingency Plan	Trigger	Responsibility	Likelihood
COVID-19 causes closure of campus.	Mitigate: Do not use university computers where possible for development work.	Use AWS deep learning as a replacement for university processing power if required.	National lockdown due to COVID-19.	HS	High
Speed of knowledge acquisition in deep learning not fast enough.	Mitigate: Study through summer and accumulate resources on deep learning.	Use machine learning techniques instead of deep learning. Alternatively, use pre trained classifiers available from Apple and Google.	Christmas first model deadline missed.	HS	Medium
Dependencies lose support.	Mitigate: Ensure dependencies have received long term support from their proprietor.	Switch to another package for the required task.	Notification on dependency newsletter, GitHub, or social media.	Dependency Vendor	Low
Hardware malfunction.	Mitigate: Ensure all work is backed up in multiple places including version control on remote GitHub.	Use machine provided by the university or another household member.	Null access to hardware over a 24-hour period.	HS	Low

Appendix 4

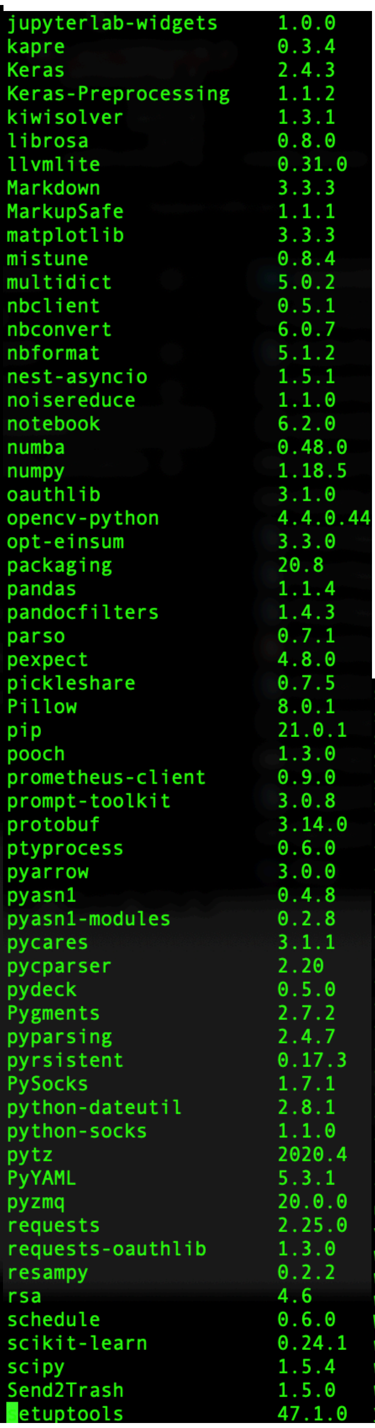

Siren Classifier
Upload a wav file
Drag and drop file here
Browse Files
Predict
This is a siren

Appendix 5

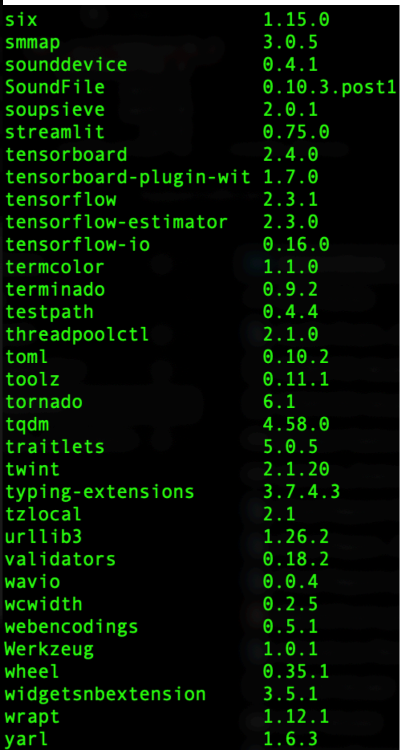
Note: Some packages are part of conda and are not used in the project explicitly.



```
absl-py                 0.11.0
aiodns                  2.0.0
aiohttp                 3.7.3
aiohttp-socks           0.5.5
altair                  4.1.0
appdirs                 1.4.4
appnope                 0.1.0
argon2-cffi             20.1.0
astor                   0.8.1
astunparse              1.6.3
async-generator         1.10
async-timeout           3.0.1
attrs                   20.3.0
audioread               2.1.9
backcall                0.2.0
base58                  2.1.0
beautifulsoup4          4.9.3
bleach                  3.2.3
blinker                 1.4
cachetools              4.1.1
cchardet                2.1.7
certifi                 2020.11.8
cffi                    1.14.3
chardet                 3.0.4
click                   7.1.2
cycler                  0.10.0
decorator               4.4.2
defusedxml              0.6.0
elasticsearch           7.10.0
entrypoints             0.3
fake-useragent          0.1.11
gast                    0.3.3
geographiclib           1.50
geopy                   2.0.0
gitdb                   4.0.5
GitPython               3.1.12
google-auth             1.23.0
google-auth-oauthlib    0.4.2
google-pasta            0.2.0
googletransx            2.4.2
grpcio                  1.33.2
h5py                    2.10.0
idna                    2.10
ipykernel               5.3.4
ipython                 7.19.0
ipython-genutils        0.2.0
ipywidgets              7.6.3
jedi                    0.17.2
Jinja2                  2.11.2
joblib                  1.0.0
jsonschema              3.2.0
jupyter-client          6.1.7
jupyter-core            4.7.0
jupyterlab-pygments     0.1.2
```



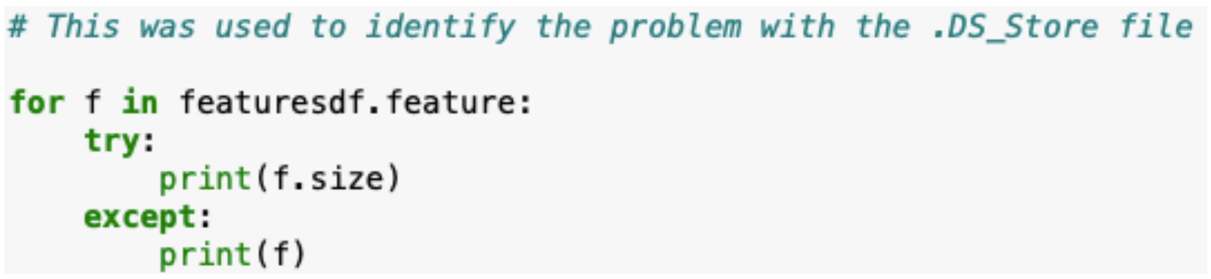
```
jupyterlab-widgets      1.0.0
kapre                   0.3.4
Keras                   2.4.3
Keras-Preprocessing     1.1.2
kiwisolver              1.3.1
librosa                 0.8.0
llvmlite                0.31.0
Markdown                3.3.3
MarkupSafe              1.1.1
matplotlib              3.3.3
mistune                 0.8.4
multidict               5.0.2
nbclient                0.5.1
nbconvert               6.0.7
nbformat                5.1.2
nest-asyncio            1.5.1
noisereduce             1.1.0
notebook                6.2.0
numba                   0.48.0
numpy                   1.18.5
oauthlib                3.1.0
opencv-python           4.4.0.44
opt-einsum              3.3.0
packaging               20.8
pandas                  1.1.4
pandocfilters           1.4.3
parso                   0.7.1
pexpect                 4.8.0
pickleshare             0.7.5
Pillow                  8.0.1
pip                     21.0.1
pooch                   1.3.0
prometheus-client       0.9.0
prompt-toolkit          3.0.8
protobuf                3.14.0
ptyprocess              0.6.0
pyarrow                 3.0.0
pyasn1                  0.4.8
pyasn1-modules          0.2.8
pycares                 3.1.1
pycparser               2.20
pydeck                  0.5.0
Pygments                2.7.2
pyparsing               2.4.7
pyrsistent              0.17.3
PySocks                 1.7.1
python-dateutil         2.8.1
python-socks            1.1.0
pytz                    2020.4
PyYAML                  5.3.1
pyzmq                   20.0.0
requests                2.25.0
requests-oauthlib       1.3.0
resampy                 0.2.2
rsa                     4.6
schedule                0.6.0
scikit-learn            0.24.1
scipy                   1.5.4
Send2Trash              1.5.0
setuptools              47.1.0
```



```
six                     1.15.0
smmap                   3.0.5
sounddevice             0.4.1
SoundFile               0.10.3.post1
soupsieve               2.0.1
streamlit               0.75.0
tensorboard             2.4.0
tensorboard-plugin-wit  1.7.0
tensorflow              2.3.1
tensorflow-estimator    2.3.0
tensorflow-io           0.16.0
termcolor               1.1.0
terminado               0.9.2
testpath                0.4.4
threadpoolctl           2.1.0
toml                    0.10.2
toolz                   0.11.1
tornado                 6.1
tqdm                    4.58.0
traitlets               5.0.5
twint                   2.1.20
typing-extensions       3.7.4.3
tzlocal                 2.1
urllib3                 1.26.2
validators              0.18.2
wavio                   0.0.4
wcwidth                 0.2.5
webencodings            0.5.1
Werkzeug                1.0.1
wheel                   0.35.1
widgetsnbextension      3.5.1
wrapt                   1.12.1
yarl                    1.6.3
```

Appendix 6

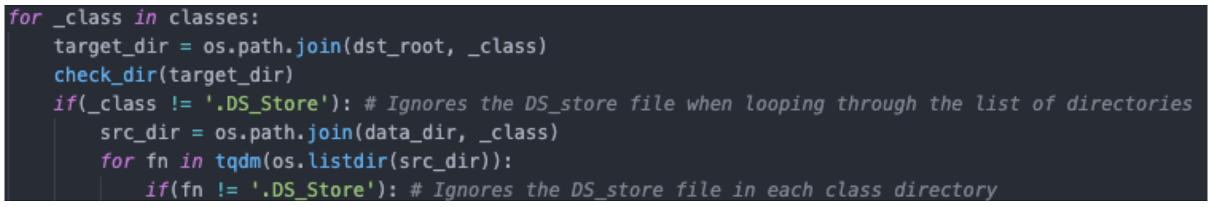
Code used to identify issue with DS_store file:



```
# This was used to identify the problem with the .DS_Store file

for f in featuresdf.feature:
    try:
        print(f.size)
    except:
        print(f)
```

Code used in Conv1D, Conv2d and LSTM to prevent DS_store issues reoccurring:



```
for _class in classes:
    target_dir = os.path.join(dst_root, _class)
    check_dir(target_dir)
    if(_class != '.DS_Store'): # Ignores the DS_store file when looping through the list of directories
        src_dir = os.path.join(data_dir, _class)
        for fn in tqdm(os.listdir(src_dir)):
            if(fn != '.DS_Store'): # Ignores the DS_store file in each class directory
```

Appendix 7

Unit Tests

TEST ID	UNIITTEST NAME	PROCEDURE	EXPECTED RESULTS	ACTUAL RESULTS	STATUS	COMMENTS
1.1	test_classifier	Assess functionality for creating models with different number of classes. This test Is for Conv1D.	The output layer should be the same shape as the number of classes specified.	Expected shape and compiled model shape are the same.	PASS	This test passes but doesn’t guarantee that model architecture is best for a multiclass classifier. For example, sigmoid and softmax functionality.
1.2	test_classifier	Assess functionality for creating models with different number of classes. This test Is for Conv2D.	The output layer should be the same shape as the number of classes specified.	Expected shape and compiled model shape are the same.	FAIL	For this model, the number of classes in the output layer had been hardcoded to 2. This has been changed.
1.3	test_classifier	Assess functionality for creating models with different number of classes. This test Is for LSTM.	The output layer should be the same shape as the number of classes specified.	Expected shape and compiled model shape are the same.	FAIL	As above.
2.1	test_input_shape	Assess the input shape calculation and examine for hard coded values. This test is for Conv1D.	The input shape should be the same as the one specified by the values.	Both values asserted equal.	PASS	No hard coded values, model can adapt to differing delta time and input shape at this level.

2.2	test_input_shape	Assess the input shape calculation and examine for hard coded values. This test is for Conv2D.	The input shape should be the same as the one specified by the values.	Both values asserted equal.	PASS	As above.
2.3	test_input_shape	Assess the input shape calculation and examine for hard coded values. This test is for LSTM.	The input shape should be the same as the one specified by the values.	Both values asserted equal.	PASS	As above.
3	test_number_of_classes	Assess whether the number of classes created from the clean and split function is different to the amount specified by the user.	The number of classes specified by the user should be the same as the number calculated when counting folders.	Assert fails with two input classes and three output classes.	FAIL	This is due to the .DS_Store issue. As a new folder is created for clean data, a .DS_Store file is made too.
4	test_downsample_function	Assess whether the down sample function is correctly changing all files to the specified sample rate.	For each file, the sample rate should be the same as the one specified for the clean data file. If one file is wrong, the assertion will fail.	Assertion passes for all files.	PASS	

User Tests

Person A

Test ID	Test Case Description	User Results	User Comments	Developer Comments
1	Access web application on local network	PASS	Able to access site.	

2	Record audio file	PASS	Able to record siren using 3rd party application.	
3	Upload audio file	FAIL	Unable to upload recorded audio file.	This is due to audio recordings taking place as an MP3 file. Wav files required by system and no build in convertor.
4	Choose network	PASS	Easy and Intuitive.	
5	Attempt prediction	PASS	Classifier correctly identified siren.	
s	Change network and re-predict	PASS	This was a pass but no evidence of the classification being rerun with new network.	

Bibliography

Baynes, C. (2018). *Ambulance urgent response time targets missed every month since shake-up*. [online] Available at: https://www.independent.co.uk/news/ambulance-response-time-999-call-targets-emergency-nhs-england-hospital-labour-a8700611.html [Accessed 10 March 2021].

Hershey, S., Chaudhuri, S., Ellis, D.P.W., Gemmeke, J.F., Jansen, A., Moore, R.C., Plakal, M., Platt, D., Saurous, R.A., Seybold, B., Slaney, M., Weiss, R.J. and Wilson, K. (2017). *CNN architectures for large-scale audio classification*. [online] IEEE Xplore. Available at: https://ieeexplore.ieee.org/abstract/document/7952132 [Accessed 20 Sep. 2020].

Guodong Guo and Li, S.Z. (2003). Content-based audio classification and retrieval by support vector machines. *IEEE Transactions on Neural Networks*, 14(1), pp.209–215. [Accessed 10 March 2021].

Greenberg, S. and Kingsbury, B.E.D. (1997). *The modulation spectrogram: in pursuit of an invariant representation of speech*. [online] IEEE Xplore. Available at: https://ieeexplore.ieee.org/abstract/document/598826 [Accessed 14 Jan. 2021].

Meucci, F., Pierucci, L., Del Re, E., Lastrucci, L. and Desii, P. (2008). *A real-time siren detector to improve safety of guide in traffic environment*. [online] IEEE Xplore. Available at: https://ieeexplore.ieee.org/abstract/document/7080691 [Accessed 10 March 2021].

Salekin, A., Ghaffarzadegan, S., Feng, Z. and Stankovic, J. (2019). *A Real-Time Audio Monitoring Framework with Limited Data for Constrained Devices*. [online] IEEE Xplore. Available at: https://ieeexplore.ieee.org/abstract/document/8804744 [Accessed 10 March 2021].

Choi, K., Fazekas, G., Sandler, M. and Cho, K. (2016). Convolutional Recurrent Neural Networks for Music Classification. *arXiv:1609.04243 [cs]*. [online] Available at: https://arxiv.org/abs/1609.04243 [Accessed 20 September 2020].

Google. *Learn how Waymo drives - Waymo Help*. [online] Available at: https://support.google.com/waymo/answer/9190838?hl=en. [Accessed 10 March 2021].

Askari, M. (2018). *Cars That Can Hear Sirens Are Coming*. [online] Car and Driver. Available at: https://www.caranddriver.com/news/a23397054/zf-emergency-vehicle-detection-ai/ [Accessed 20 September 2020].

Smales, M. (2020). *mikesmales/Udacity-ML-Capstone*. [online] GitHub. Available at: https://github.com/mikesmales/Udacity-ML-Capstone. [Accessed 26 March 2021].

Adams, S. (2021). *seth814/Audio-Classification*. [online] GitHub. Available at: https://github.com/seth814/Audio-Classification [Accessed 29 March 2021].

Agilebusiness (2019). *Chapter 15: Requirements and User Stories*. [online] Available at: https://www.agilebusiness.org/page/ProjectFramework_15_RequirementsandUserStories. [Accessed 10 March 2021].

Agile Alliance. (2017). *What is a Minimum Viable Product (MVP)?* [online] Available at: https://www.agilealliance.org/glossary/mvp/.[Accessed 10 March 2021].

Wikipedia. *Sampling (Signal Processing)*. [online] Available at: https://en.wikipedia.org/wiki/Sampling_(signal_processing) [Accessed 24 March 2021].

Wikipedia (2019). *Spectrogram*. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Spectrogram. [Accessed 24 March 2021].

Wikipedia. (2019). *Mel-frequency cepstrum*. [online] Available at: https://en.wikipedia.org/wiki/Mel-frequency_cepstrum. [Accessed 24 March 2021].

Pawar, M.D. and Kokate, R.D. (2021). Convolution neural network based automatic speech emotion recognition using Mel-frequency Cepstrum coefficients. *Multimedia Tools and Applications*. [Accessed 10 March 2021].

Wikipedia. (2021). *.DS_Store*. [online] Available at: https://en.wikipedia.org/wiki/.DS_Store [Accessed 24 March 2021].

Choi, K. (2021). *keunwoochoi/kapre*. [online] GitHub. Available at: https://github.com/keunwoochoi/kapre [Accessed 10 March 2021].

Keras. *Keras documentation: TimeDistributed layer*. [online] keras.io. Available at: https://keras.io/api/layers/recurrent_layers/time_distributed/. [Accessed 10 March 2021].

deeplizard.com. *Max Pooling in Convolutional Neural Networks explained*. [online] Available at: https://deeplizard.com/learn/video/ZjM_XQa5s6s [Accessed 10 March 2021].

Little, Z. (2020). *Conv1D, Conv2D and Conv3D*. [online] Medium. Available at: https://xzz201920.medium.com/conv1d-conv2d-and-conv3d-8a59182c4d6 [Accessed 24 March 2021].

Graves, A., Jaitly, N. and Mohamed, A. (2013). *Hybrid speech recognition with Deep Bidirectional LSTM*. [online] IEEE Xplore. Available at: https://ieeexplore.ieee.org/abstract/document/6707742 [Accessed 13 February 2021].

Elliott, E. (2019). *The Outrageous Cost of Skipping TDD & Code Reviews*. [online] Medium. Available at: https://medium.com/javascript-scene/the-outrageous-cost-of-skipping-tdd-code-reviews-57887064c412#.qmzgmfd2u [Accessed 13 February 2021].