

Software Engineering Department

Capstone Project Phase B

**Manipulated Reality**

**Video and audio analysis using Deep Learning for deepfake detection.**

<https://github.com/MaximL98/Manipulated-Reality>

**Project ID 24-1-D-5**

**Supervisor: Dr. Reuven Cohen**

|  |  |  |
| --- | --- | --- |
| **Name** | Maxim Lebedinsky | Dmytri Kislitsyn |
| **ID** | 318171485 | 323349738 |
| **E-mail** | Maxim.Lebedinsky@e.braude.ac.il | Dmytri.Kislitsyn@e.braude.ac.il |

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **Index** | **Subject** | **Page number** |
| **1** | **Abstract** | [3](#s1) |
| **2** | **Introduction** | [3](#s2) |
| **3** | **The project** | [5](#s3) |
| 3.1 | Website | [5](#s3_1) |
| 3.1.1 | Activity | [5](#s3_1_1) |
| 3.1.2 | UI example | [5](#s3_1_2) |
| 3.1.3 | Project Package Diagram | [6](#s3_1_3) |
| 3.2 | Deepfake detection using Machine learning | [7](#s3_2) |
| 3.3 | Video Model | [7](#s3_3) |
| 3.4 | Audio model | [8](#s3_4) |
| **4** | **Development process** | [11](#s4) |
| 4.1 | Video Model | [11](#s4_1) |
| 4.2 | Audio Model | [11](#s4_2) |
| 4.2.1 | Pre-processing | [11](#s4_2_1) |
| 4.2.2 | Model training | [12](#s4_2_2) |
| 4.3 | Web UI | [12](#s4_3) |
| 4.3.1 | Backend | [12](#s4_3_1) |
| 4.3.2 | Frontend | [13](#s4_3_2) |
| 4.4 | Datasets | [13](#s4_4) |
| 4.4.1 | Video dataset | [13](#s4_4_1) |
| 4.4.2 | Audio datasets | [14](#s4_4_2) |
| **5** | **Technological infrastructure** | [15](#s5) |
| 5.1 | Video classification | [15](#s5_1) |
| 5.2 | Audio classification | [15](#s5_2) |
| 5.3 | Tests and edge cases | [16](#s5_3) |
| **6** | **Constraints and challenges** | [17](#s6) |
| 6.1 | Video classification | [17](#s6_1) |
| 6.2 | Audio classification | [18](#s6_2) |
| **7** | **Results and experiments** | [19](#s7) |
| 7.1 | Video initial experiments | [19](#s7_1) |
| 7.2 | Video model results | [20](#s7_2) |
| 7.3 | Audio initial experiments | [22](#s7_3) |
| 7.4 | Audio model results | [23](#s7_4) |
| **8** | **Conclusions** | [26](#s8) |
| 8.1 | Key Takeaways | [27](#s8_1) |
| **9** | **References** | [29](#s9) |
|  | **End of project book** |  |
| **10** | **User guide** | [30](#s10) |
| **11** | **Maintenance guide** | [34](#s11) |
| 11.1 | Installation | [34](#s11_1) |
| 11.2 | Video model | [35](#s11_2) |
| 11.2.1 | Video prediction pipeline | [35](#s11_2_1) |
| 11.2.2 | Video model training | [35](#s11_2_2) |
| 11.2.3 | Fine tuning pretrained InceptionV3 model | [38](#s11_2_3) |
| 11.2.4 | Video training | [40](#s11_2_4) |
| 11.3 | Audio model | [42](#s11_3) |
| 11.3.1 | Preprocessing datasets | [42](#s11_3_1) |
| 11.3.2 | Training the audio model | [45](#s11_3_2) |
| 11.3.3 | Preprocessing audio for detection | [47](#s11_3_3) |
| 11.4 | Frontend | [50](#s11_4) |
| 11.5 | Backend | [51](#s11_5) |

**1. Abstract**

This development and research project aims to contribute to the ongoing efforts to combat the generation of deepfakes. We present a novel framework for the detection of deepfake media that involves two distinct models: A hybrid deep learning architecture utilizing the Keras framework for the classification of deepfake content. The framework incorporates a pre-trained InceptionV3 model for video feature extraction, followed by a Long Short-Term Memory (LSTM) network to capture temporal dependencies.

Additionally, by utilizing the SoundFile and Librosa frameworks, which are used for processing audio, we feed the processed audio into a sequential CNN model for identifying fake speech. By combining these models in a unique detection application, we can effectively detect deepfakes that incorporate both visual and auditory manipulations. We can also consider cases involving solely video or audio sources.

Our experimental results demonstrate an accuracy of approximately 83% for detection of video-based deepfakes and 92% of audio-based deepfakes. To ensure widespread accessibility, we have developed a user-friendly web application that allows users to upload suspected deepfake content and receive a probability score indicating its authenticity.

To further enhance the model's performance, a comprehensive hyperparameter optimization study is recommended exploring various combinations of parameters such as loss function, number of epochs, learning rate etc. Additionally, augmenting the training dataset with a diverse range of video and audio content could be beneficial.

**2.** **Introduction**

Deepfakes are a subset of artificial intelligence that primarily leverage two neural network architectures: Generative Adversarial Networks (GANs[[1]](#footnote-1)) and Autoencoders (AEs[[2]](#footnote-2)). These networks are used to create highly realistic, fabricated content as illustrated in [Figure 1](#f1).

The term "deepfake" encompasses both the technology behind these creations and the fakes themselves, combining "deep learning" and "fake." Deepfakes often manipulate existing content by replacing one person with another or generating entirely new content featuring individuals performing actions or saying words they never did.

Within the past year, the number of reported deepfakes identify online skyrocket [[1]](#b1). News reports from various global regions highlight the growing frequency of incidents involving the malicious use or potential harm caused by deepfake content [[2]](#b2)[[3]](#b3)[[4]](#b4)[[5]](#b5). The United States federal government has enacted legislation aimed at mitigating the risks posed by deepfake technology [[6]](#b6).  
To address the growing concern about deepfake content, we developed two independent machine learning models: one for video and one for audio detection. The video model was based on a fine-tuned InceptionV3 [[7]](#b7) architecture and a long short-term memory (LSTM[[3]](#footnote-3)) network. For training, we utilized the Celeb-DF-v2 [[8]](#b8) dataset, consisting of 590 genuine and 5639 deepfake videos collected from YouTube.

The audio model was made by utilizing the Deep Voice, In the Wild, and Fluent Speech Corpus datasets. These datasets were pre-processed by partially following the method purposed in article [[13]](#b13). We preprocessed the data by extracting the MFCC[[4]](#footnote-4) features [[Figure 8](#f8)], and the first and second derivatives of the MFCC’s called delta[[5]](#footnote-5). These features are then fed into the model as training data. These features essentially represent different dependencies in the sound spectrum of a human’s voice and can be analyzed by a machine learning model to classify speech as synthetic or generic.

By providing numerical outputs from both the video and audio models, we developed an approach to detecting deepfakes that incorporate both visual and audio manipulations.

Our models are computationally efficient, requiring minimal hardware resources. The fine-tuning process has significantly reduced the computational demands of the detection task, making it feasible on a wide range of devices. Despite their compact size, these models demonstrate robust performance, achieving approximately 83% accuracy that was tested on ~3000 videos for video analysis and a remarkable 99% accuracy that was tested on ~61k audio recordings for audio analysis.

To smooth out the application of machine learning models, we've developed a user-friendly web interface. Users can register to upload suspected content and select from three detection options: video and audio, video only, or audio only. Based on their choice, the models preprocess the data and initiate the detection process. Upon completion, the results for each detection are stored in the user's profile for convenient access and review.

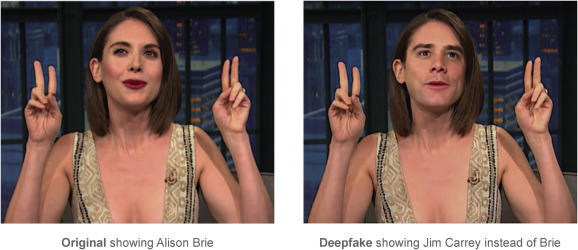


Figure 1: A deepfake of Jim Carrey instead of Alison Brie.

**3. The project**

**3.1 Website**

To allow wider and simple access to the detection models, we developed a modular website-based UI that enables easy future expansion and upgradeability. This website was built by using the React framework to create an interactive client side while utilizing the Flask framework as the backend, to allow developers to access the machine learning code directly through the website. This website allows users to create an account, upload video or audio files, choose the detection type (Video, audio, or both), and see their previous results in their personal profile page.

**3.1.1 Activity diagram**

[Figure 2](#f2) illustrates the entire process of the deep fake detection.

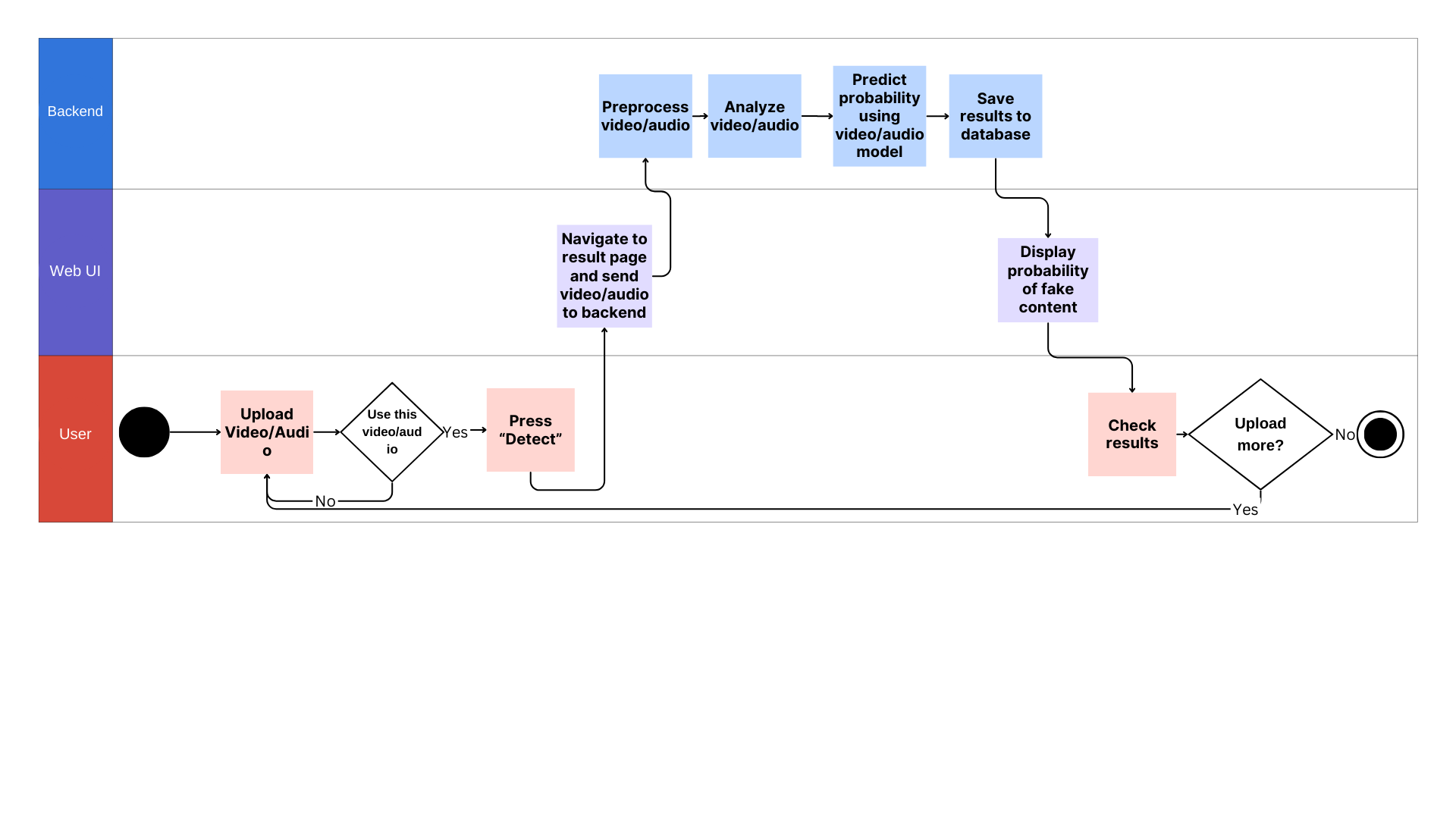
****

Figure 2: Deepfake detection process activity diagram.

**Important:** The project is currently developed locally and not deployed online, since hosting such a computation-heavy backend of machine learning and database management code is expensive to be publicly published online.

**3.1.2 UI example**

[Figure 3](#f3) illustrates the user interface that is displayed upon successful detection of a fake video. The presented video is an unseen sample from the Celeb-DF v2 dataset, that was classified as having a 23.16% probability of being real. This low probability strongly suggests the video's authenticity is highly questionable. From this interface, users can upload additional files or access their profile page to compare current results with previous detections.

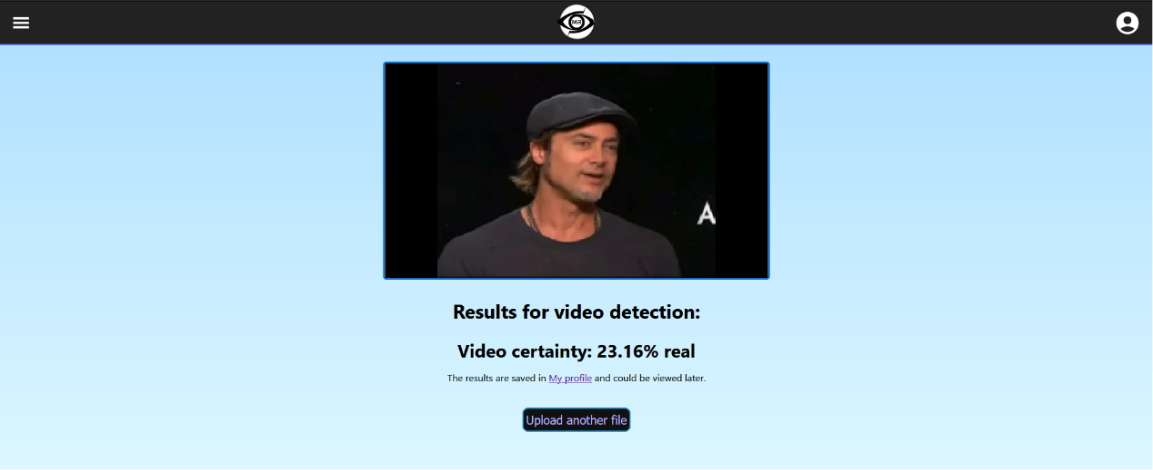


Figure 3: Detection result for a fake video.

**3.1.3 Project package diagram**

The following diagram in [Figure 4](#f4) illustrates the complete architecture of our project. It details the components, the integration of our models, and the connection between the backend and the frontend (Additional package diagrams can be found in the [Maintenance guide](#MG) section).

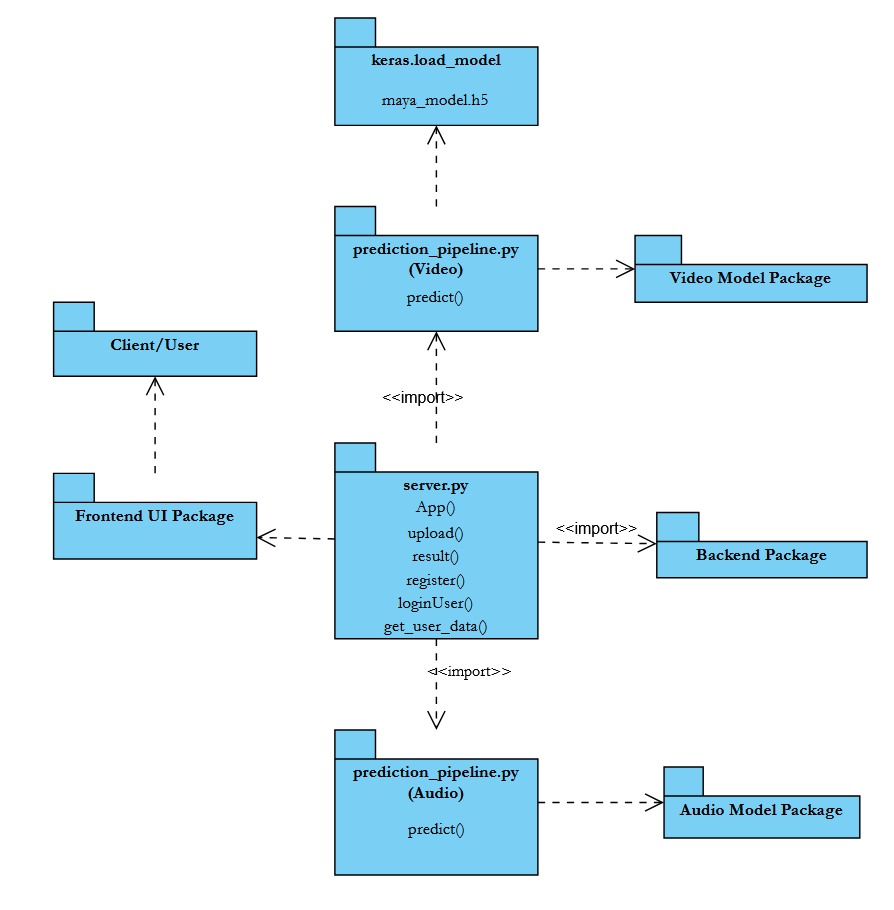


Figure 4: Package diagram of the entire system.

**3.2 Deepfake detection using Machine learning**

Our machine learning project consists of two independently developed models: a video classifier and an audio classifier. The video classifier was constructed using a hybrid architecture incorporating Convolutional Neural Networks (CNNs) [[10]](#b10) and Long Short-Term Memory (LSTM) [[11]](#b11) networks, which are powerful models for tasks that involve both spatial and temporal dependencies. The audio classifier was solely based on a CNN architecture.

[Figure 5](#f5) depicts the process of detection in our system.

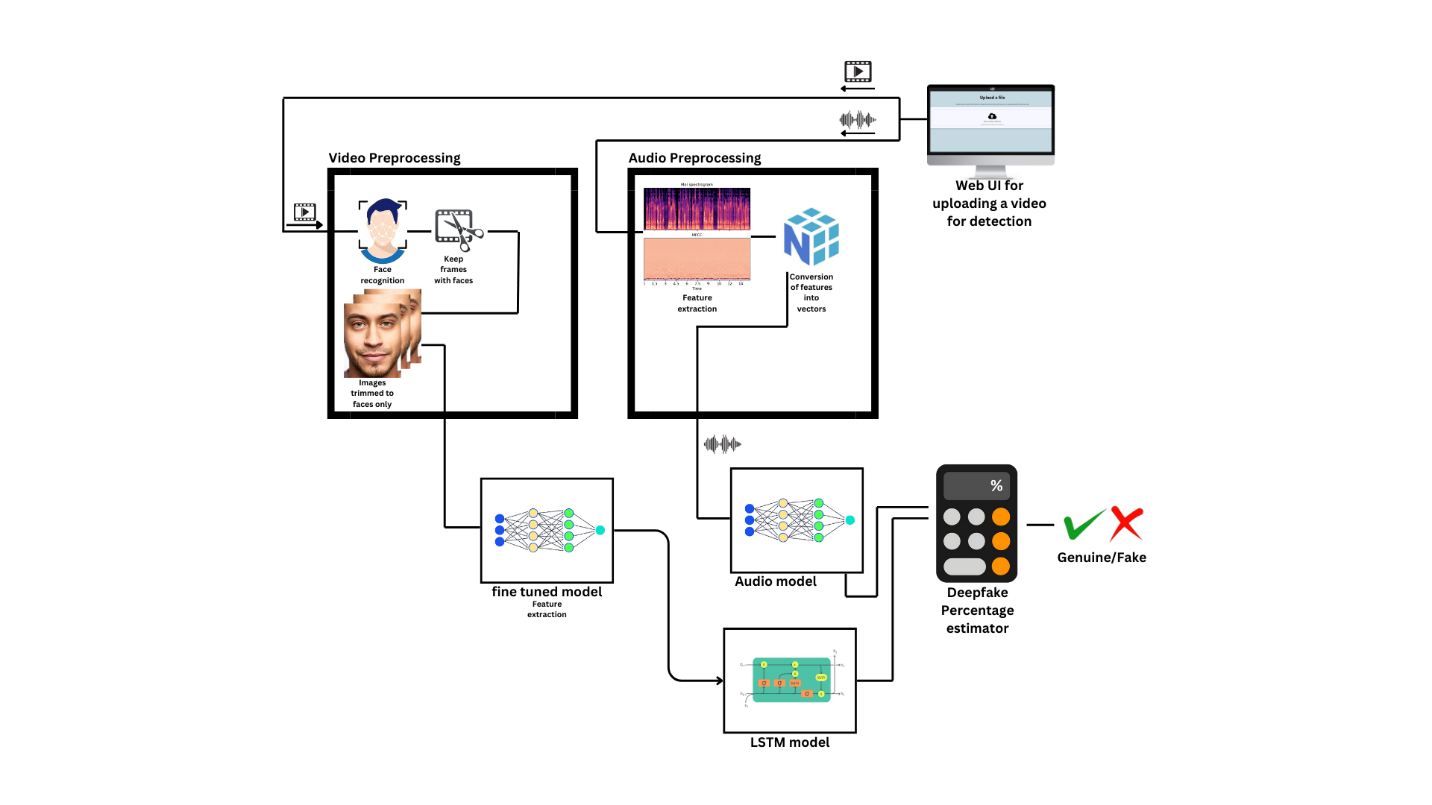
****

Figure 5: Detection process architecture.

**3.3 Video model**

CNNs excel at extracting spatial features. CNNs are a type of deep learning architecture that is particularly well-suited for tasks involving classification. When applied to facial features, CNNs can learn to identify and categorize different aspects of a person's face, such as eyes, nose, mouth, and even specific facial expressions.

LSTMs are designed to handle sequential data. They remember information from previous time steps, making them ideal for tasks such as video classification. Feeding the CNN output to an LSTM allows the network to leverage these learned spatial features for subsequent temporal analysis. LSTM is critical, as this model primary objective is to identify inconsistencies between the facial features.

Our CNN model employed a fine-tuned Inception-V3 architecture (third version of Google’s inception CNN) [[Figure 6](#f6)]. The pre-trained Inception-V3 model, containing approximately 26 million parameters, and was adapted to our specific task of facial image classification. To enhance model performance, the remaining trainable parameters were optimized on a dataset of facial images both real and fake, resulting in a more specialized CNN for our application.

A diagram of a structure

Description automatically generated

Figure 6: InceptionV3 pre-trained model architecture.

Subsequently, the extracted features are fed into a Long Short-Term Memory (LSTM) network [[Figure 7](#f7)], to capture temporal dependencies inherent in video data.

**A diagram of a rectangular object

Description automatically generated**

Figure 7: LSTM model architecture.

**3.4 Audio model**

The audio detection stage of the project consists of a machine learning model trained to recognize synthetic and generic speech. It was created by training the model on a massive collection of fake/synthetic and real/generic speech recordings. These datasets contained ~61k samples of audio recordings which were preprocessed to extract audio features and dependencies out of.

The extracted features are Mel Frequency Cepstral Coefficients (MFCC), delta cepstral, and delta2 cepstral, which are used to visually represent audio, but can also represent dependencies and features of human speech in audio recordings.

[Figure 8](#f8) shows a Mel spectrogram which uses these MFCC’s to represent audio visually.

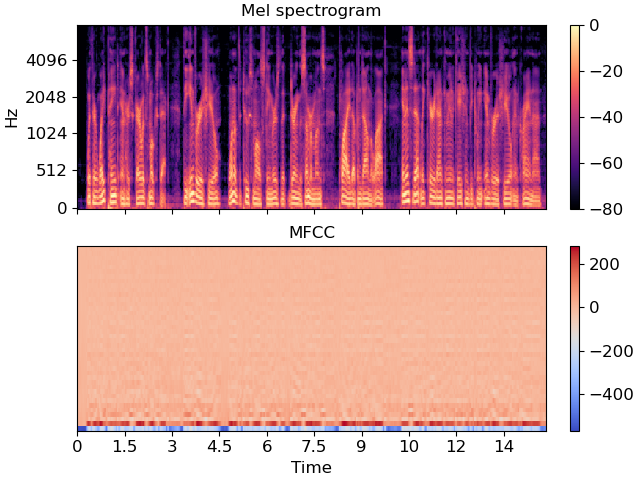


Figure 8: Mel spectrogram built out of MFCC coefficients.

After we preprocessed the datasets, we then created multi-dimensional vectors from the extracted features for each recording and fed them as training, testing, and validation data to a sequential CNN machine learning model.   
The model’s structure is displayed in Figures [9](#f9), and [10](#f10).

A screenshot of a computer

Description automatically generated

Figure 9: Textual representation of the audio detection model’s structure.

A diagram of a graph

Description automatically generated with medium confidence

Figure 10: Visual representation of the audio detection model’s structure.

After training the model, we evaluated the accuracy and additional metrics such as loss, recall, and precision of the model.

The preprocessing stage utilized the following tools for feature extraction:

1. Librosa library: Used to extract MFCC, delta cepstral, and delta2 cepstral features from audio files.
2. Soundfile library: Used to open and read audio files.
3. NumPy library: Used to store extracted audio features in .npy files and handle multi-dimensional data variables.

The training stage utilized the following tools for model training:

1. NumPy library:Used in opening saved datasets as .npy files.
2. Sklearn library:Used for plotting the various results of the trained model, such as confusion matrix and accuracy over epochs.
3. Keras framework:Used as the main framework for CNN model training.
4. Tensorflow framework**:** Used in combination with Keras for the development and training of the audio detection model.

**4. Development process**

**4.1 Video model**

A broad dataset was assembled, including both authentic and deepfake videos using Celeb-DF (v2) dataset. To ensure optimal model performance, videos with the highest perceived deepfake quality were prioritized. The dataset consisted of short video sequences (10-15 seconds) featuring a single individual in each frame, mostly in the format of interview.

Following data collection, the dataset was split into training, testing, and validation subsets ensuring that the testing data remained unseen during model training. Each video underwent preprocessing to isolate the facial region using the Python library "face\_recognition"[[12]](#b12), and the frame rate was standardized to 15 frames per second.

This ensured a representative sample of facial expressions, as high frame rate videos can contain unnecessary information. The extracted facial images were then resized to 224x224 pixels and saved as NumPy arrays. A maximum of 200 frames were retained per video, resulting in NumPy arrays with dimensions of 200x224x224.

The pre-processed video frames were subsequently fed into a fine-tuned Inception V3 model. This model, originally developed by Google, was adapted for the task of deepfake detection through training on a dataset of 10,000 real and fake images. The fine-tuning process enabled efficient and resource-friendly feature extraction from the video frames.

The extracted features were then employed to train a Long Short-Term Memory (LSTM) model with Binary Cross Entropy as a loss function. LSTM networks are well-suited for processing sequential data, such as video frames, and capturing temporal dependencies. The trained LSTM model was tasked with predicting whether a given video was authentic or a deepfake.

**4.2 Audio model**

Due to the small amount of high-quality generated audio currently available online, we opted to use as many audio samples as possible to maximize the model’s familiarity with varying audio quality. Therefore, three datasets (In the wild, Deep Boice, and Fluent speech corpus) collectively containing 50+ hours of audio were chosen to train and develop the audio detection model.

**4.2.1 Pre-processing**

To generalize the data, each audio sample was sampled to 22.05khz from its original sample rate and split into 1 second chunks using the Librosa library. After splitting, the last chunk of each audio recording would be padded if it is at least 75% of a second, otherwise, it was discarded.

These sections went through preprocessing to extract special features and dependencies from the voices of the sampled subjects. Preprocessing was done by the method proposed in [[13]](#b13).   
During preprocessing, 100 MFCC coefficients were extracted from the sample sections.   
Each extracted MFCC coefficient contained 44 different dependencies which totaled to 4400 values per sample section. These 4400 values were then used to calculate delta and delta2 features.

The output from this process yielded a vector of size 13200 per chunk, totaling 193196 vectors from the entire dataset collection. Unlike video, there is no meaning in arranging the extracted features in matrixes, thus, vectors in the form of <MFCC’s, delta, delta2> seemed to be the ideal and easiest way to store the features and feed them as training data to the audio detection model.

Finally, the extracted vectors were stored in NumPy files to be fed into the audio detection model.

**4.2.2 Model training**

To train the model, we first needed to scale the data to deal with outliers. We scaled the data using a Robust Scaler provided by the Sklearn library, and it seemed to give the most promising training results. After scaling, 108k scaled vectors were then fed as training data to the Sequential model, 49k were used as testing data, and 36k were used as validation data.

We chose a batch size of 1024 and 5 epochs setting for the training phase. The reason for picking this exact setting is described in the [Audio experiments](#s7_3) section.   
The 5th epoch resulted in an accuracy score of 99.22%.

**4.3 Web UI**

To create an aesthetically pleasing, easy-to-use, and simple application, we opted on using a web-based UI that will communicate with the backend code and the model. This was done by incorporating React as the client-side framework, and Flask as the Python backend.

Ultimately, we chose this approach over developing a simple Python-based desktop application to allow users from multiple platforms to engage and be able to use our application. This also allows us to easily make improvements to the UI and to the backend logic separately in the future. Learning these frameworks was achieved by using AI tools, various YouTube tutorials, and widely available tutorials online.

**4.3.1 Backend**

For the backend component of our web application, we employed the Python-based Flask framework. Flask served as the backend server, responsible for managing HTTP requests originating from the client-side. This framework’s integration with React JS facilitated the exchange of data via HTTP requests directed to the Flask API.

Given that our models’ pipelines are implemented in Python, the utilization of Flask streamlined the execution of the necessary code for both video and audio preprocessing, model execution, and the dissemination of prediction results.

Furthermore, we established a connection between the backend server and a local SQLite3 database. This database serves as a repository for registered user information and user history, which is subsequently displayed on the user's profile page.

**4.3.2 Frontend**

The frontend was built using the React framework to allow smooth communication between the client and the server. React allowed us to build a dynamic application, pass data between pages, create animations, create a login system, allow the users to upload files, and overall, the website can be easily upgraded in the future.

React allows the users to send requests to the backend via the “Fetch API” which is typically used to communicate with different API’s, databases, or websites.  
Our frontend can communicate seamlessly with our Flask backend to pass large files or requests between the client and the server.

**4.4 Datasets**

**4.4.1 Video dataset**

The Celeb-DF (v2) dataset [Figure 11](#f11), consist of 590 celebrity interview videos (10-14 seconds each), served as the foundation for generating 5639 deepfake videos. A noticeable class imbalance was evident between genuine and synthetic samples. To mitigate this, an extended dataset was constructed by incorporating 300 videos from YouTube, 1000 from C23, and 7500 from DFMNIST+[[14]](#b14). Given the halved frame rate of DFMNIST+ videos, a speed adjustment was applied to match other videos frame rate, which lead to shorter video duration (10-14 seconds to 5-7 seconds), resulting in an effective sample size of approximately 3750 videos. This enlargement process aimed to balance the total video duration between real and fake categories while maintaining consistent video format (10-14 second single-person shots) across all datasets.

The datasets were later split into train, test and validation as follows:

* Train: 8949 videos
* Test: 3009 videos
* Validation: 3069 videos



Figure 11: Examples from Celeb-DF V2 dataset.

**4.4.2 Audio datasets**

**In the Wild** [[15]](#b15)

The In the wild dataset contains 31,781 real and generated audio recordings of 58 celebrities and politicians, averaging 23 minutes of real and 18 minutes of fake audio per speaker.

**Fluent Speech Corpus** [[16]](#b16)

The Fluent Speech Corpus dataset contains 30,043 real audio recordings of 97 speakers performing various voice commands. This dataset is diverse since it contains multiple speakers in different settings such as their home, outside, and recordings with background noise and without. This allowed our model to learn to recognize real human voices regardless of the environment or setting.

**Deep Voice** [[17]](#b17)

The Deep Voice dataset contains 10-minute recordings of 8 real celebrities such as Joe Biden, Elon Musk and more. Each of these recordings is then voice swapped with another celebrity which results in 8 recordings of the same recorded speech (1 real and 7 fake voices) for every celebrity, totaling 64 speeches. [Figure 12](#f12) depicts the creation process of this dataset.

The process of creating deepfake voices for this dataset is done in the following way:

1. The process starts with real speech as the input.
2. The real speech is split into two separate stems: one for the accompaniment (background noise or ambient sounds) and one for the actual speech.
3. The isolated vocal stem is then processed using an RVC (Retrieval-Based Voice Conversion) model. This model is trained on a dataset of voices and can be used to convert the real vocals into a different voice or style.
4. The output of the RVC model is the fake speech. When combined with the original accompaniment stem, it creates the final fake speech.

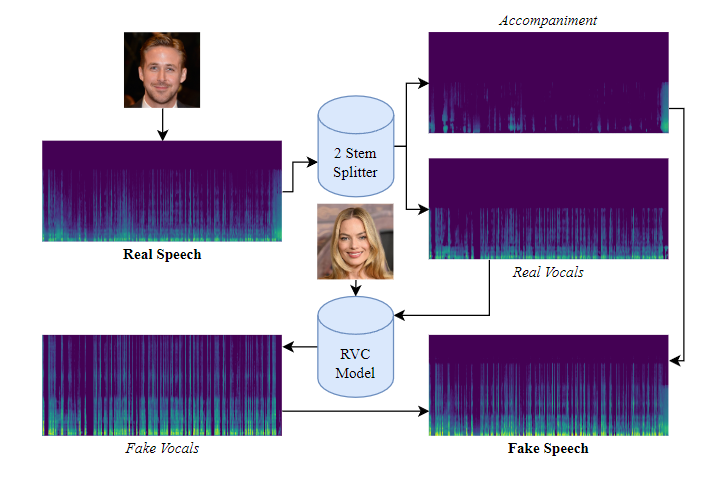


Figure 12: Deep Voice dataset creation process.

1. **Technological infrastructure**

**5.1 Video classification**

Both video-focused models were developed using Google Colaboratory, a cloud-based Jupyter notebook environment providing accessible computational resources including GPUs and TPUs. To accommodate the substantial size of the video dataset, a Colab Pro+ subscription was acquired, ensuring adequate computational power for both data preprocessing and model training phases.

A T4 GPU with high RAM was the primary computational resource employed for this project on Google Colab. While TPU v2 accelerators have demonstrated superior performance on similar tasks, the T4 GPU was selected due to its lower cost. Initial estimates of preprocessing and training time indicated that the T4 GPU would provide adequate computational power for the project’s requirements.

The models were constructed using the Python programming language and the Keras deep learning framework. Keras was employed as an interface for developing artificial neural networks. While the framework facilitates the creation and training of custom models from scratch, it also provides access to pre-trained architectures, such as Inception V3, which was leveraged in this project.

**5.2 Audio classification**

The entire process from preprocessing to training to prediction was written in Python.

The preprocessing stage of the audio datasets was done using our personal computers. Although the datasets were big in size, the stage of feature extraction from the entire dataset of audio recordings proved to be doable without using any external tools. On the contrary, the task of training the audio model for prediction proved to be impossible for a personal computer to handle.

Thus, we have used Google Colab’s A100 Graphical Processing Unit to handle the large requirement of system RAM for handling 193k audio files in one go. Finally, the model was constructed using the Keras framework.

**5.3 Tests and edge cases**

|  |  |  |  |
| --- | --- | --- | --- |
| Case | Test Case | Expected Result | Results |
| 1 | Upload a video file (e.g., mp4) | The video is loaded, a “Detect button” appears. | Passed |
| 2 | Upload an audio file (e.g., mp3) | The audio is loaded, a “Detect button” appears. | Passed |
| 3 | Select other detection type (Audio, Video, or Audio + Video) | The system acknowledges the user’s selection. Highlighting the chosen detection type. | Passed |
| 4 | Press “Detect” | The system displays a clear indication that analyzing has begun using a progress bar. | Passed |
| 5 | User replaces the uploaded file with another | The old filename is swapped to the name of the new one. | Passed |
| 6 | Upload a video/audio in an unsupported format | Error is shown:  “File not supported! Please select a supported format: mp4, mkv, avi, mov, wav, mp3” | Passed |
| 7 | Upload a corrupted video/audio file | System displays an error message indicating bad upload. | Passed |
| 8 | Upload a large video/audio file (exceeding size limit) | System displays an error message indicating bad upload. | Passed |
| 9 | Upload an empty file | System displays an error message indicating bad upload. | Passed |
| 10 | Attempt to detect audio with video detection | Detection button disappears. The user cannot press detect. | Passed |
| 11 | Attempt to detect audio with  video + audio detection | Detection button disappears. The user cannot press detect. | Passed |
| 12 | Attempt to detect audio from a video containing no sound | Server does not detect audio, responds with “null”. Client-side displays “Error: This video does not contain audio!”. | Passed |

1. **Constrains and challenges**

**6.1 Video classification**

We experienced several challenges during the development of the video analysis model.

1. Acquiring a substantial dataset of authentic, short-form (10-15 second) videos featuring a single individual posed a significant challenge. The Celeb-DF v2 dataset, while providing a rich source of high-quality deepfake videos, offered a comparatively limited number of genuine samples. Initially, the emphasis was placed on securing enough high-quality deepfakes, with the assumption that real video acquisition would be relatively straightforward.

However, it became evident that identifying authentic videos conforming to the strict criteria of length and subject isolation was considerably more difficult. While the necessary video samples were eventually obtained, preprocessing was required to align the dataset with the established parameters.

1. The initial preprocessing step involved face detection to isolate frames containing human faces. This process was initially computationally intensive, requiring approximately two minutes per video due to the employed standard face\_recognition algorithm that depends on CPU processing. Given the dataset's size of over 15,000 videos, this would have resulted in approximately 500 hours of processing time.

To expedite this process, a Convolutional Neural Network (CNN)-based face detection model from the face\_recognition library was leveraged, utilizing GPU acceleration through Google Colab. This optimization significantly reduced processing time to an average of 20 seconds per video, culminating in a total processing time of approximately 83 hours.

1. To accelerate the model development process and optimize resource utilization, an initial exploratory phase was conducted using a subset of the dataset. Rather than exhaustively evaluating all potential video parameters (e.g., frame rate reduction from 60 to 15 fps) and model configurations on the entire dataset, a 10% sample comprising 750 training videos, 375 validation and 375 testing videos was employed.

This approach significantly reduced computational costs and time expenditure, enabling rapid prototyping and iterative improvement of model architectures and hyperparameters. Insights obtained from this prior analysis informed subsequent experiments on the full dataset.

1. Initial experiments employed a 3D Convolutional Neural Network (CNN) architecture. However, due to substantial computational demands, training was restricted to video segments of 10 frames or less (approximately 0.67 seconds at 15 frames per second). This limitation prevents the analysis of longer temporal dependencies, a core objective of our research.

To address this, a pre-trained CNN model was fine-tuned on a dataset of facial images, consist of both authentic and deepfake samples. This approach yielded a significantly more efficient and effective feature extractor, enabling the subsequent application of a Long Short-Term Memory (LSTM) network to capture temporal patterns within the video data.

**6.2 Audio classification**

Three major challenges were encountered during the development of the audio model.

1. Deepfake datasets are scarcely available on the internet since this field is still growing and evolving. We managed to find only two datasets with decent quality.

To counter the possibility of the model predicting low-quality deepfakes with high accuracy, and not knowing how to deal with high quality ones, we added a third dataset (Fluent speech corpus) consisting of 30k+ real recordings of different people in different settings. We are certain this step was beneficiary and provided the model with the necessary data to better recognize real human speech.

1. Preprocessing was possible only with 32GB of RAM and would render the computer useless for the duration of the process, for hours at a time, ultimately slowing our progress down.

Moreover, training a model on 193k samples proved to be even harder to achieve, since regular home PCs could not handle the required computational power and RAM requirements for training and would eventually crash.

To solve this issue, we have utilized Google Colab’s best available A100 GPU for the training process, which took a mere 15 minutes to complete.

1. We planned to discard the silent and non-relevant parts of any uploaded audio files. This proved to be too difficult to achieve in the short time span of this project.

Therefore, we decided to consider the entire audio file in the detection process. This method proved to be useful, since generating real-world background silence is currently difficult for artificial intelligence to achieve. Additionally, our model might have learned to distinguish between real-world silence and generated silence as well as the differences in speech.

1. **Results and experiments**

**7.1 Video initial experiments**

To expedite our experimental process, we initially trained our video model on a 10% subset of the full dataset. A significant portion of the preprocessing phase was dedicated to facial recognition. To evaluate the efficacy of various algorithms, we conducted comparative analyses of two widely recommended models, ‘face\_recognition’ and ‘dlib’ [[18]](#b18).

Our face detection model of choice was ‘face\_recognition,’ selected over ‘dlib’ due to its slightly faster performance as shown in [Figure 13](#f13). While each frame may only differ by a seemingly insignificant 0.10 seconds, the cumulative effect across numerous frames can result in a substantial time difference overall.

We also explored different video preprocessing approaches, testing both 15 frames per second with 200 total frames and 30 frames per second with 100 total frames.

Our initial results indicated that a preprocessing rate of 15 frames per second with 200 total frames yielded higher accuracy (71%) compared to 30 frames per second with 100 total frames (58%). Additionally, we compared the performance of GRU[[6]](#footnote-6) and LSTM recurrent neural networks. The LSTM model consistently outperformed the GRU model, achieving an accuracy of 71%.

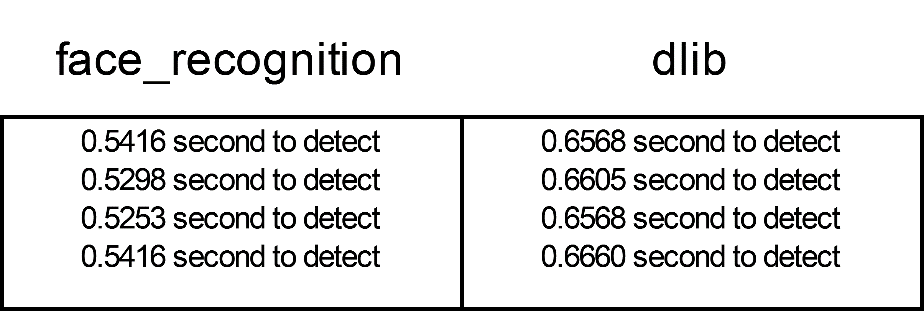


Figure 13: Frame face detection time comparison between face\_recognition and dlib, tested on the same video.

**7.2 Video model results**

The following figures illustrate the training trajectory of the LSTM component within our model.

**Accuracy Function**

During the initial training epochs, concurrent increases in both training and validation accuracy were observed, indicating effective model convergence and generalization to unseen data.   
Following an initial growth phase, the validation accuracy converges to an approximate value of 0.77, suggesting the model has acquired noticeable features from the training data and possesses robust generalization capabilities when presented with previously unencountered instances.

A graph of a graph with blue and orange lines

Description automatically generated

Figure 14: Change in video model accuracy over number of epochs.

**Confusion Matrix**

From a total of 1878 real videos, and 1131 fake videos, we got the following results:

* True Positives (TP): 93% of real instances were correctly classified as class **real**.
* True Negatives (TN): 52% of fake instances were correctly classified as class **fake**.
* False Positives (FP): 48% of fake instances were incorrectly classified as class **real** when they belonged to class **fake**.
* False Negatives (FN): 7% of real instances were incorrectly classified as class **fake** when they belonged to class **real**.

Our model performs reasonably well in predicting class **real** (high TP value). However, there's a noticeable number of false positives (FP), where the model fails to identify the fake videos correctly.

A blue squares with black text

Description automatically generated

Figure 15: Video model confusion matrix.

**Receiver Operating Characteristic (ROC) Curve**

An ideal ROC curve would lie as close as possible to the top-left corner of the graph. This indicates that the model can achieve high TPR (sensitivity) with low FPR (false positive rate).

For our model, the ROC curve is significantly above the diagonal line, indicating that the model is performing better than random guessing. However, it is not perfectly hugging the top-left corner, which suggests room for improvement in terms of balancing sensitivity and specificity.

A blue line graph with numbers

Description automatically generated

Figure 16: ROC curve of the video model.

The accompanying illustrations [Figures [17](#f17), [18](#f18)] present two processed deepfake video samples. The first illustration showcases frames that are accurately identified as a deepfake by the model. The second illustration displays frames that are mistakenly classified as authentic.

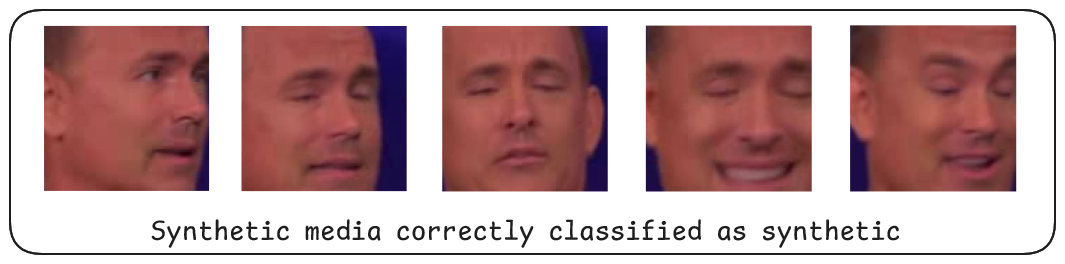


Figure 17: Deepfake video correctly classified as fake by the model.



Figure 18: Deepfake video incorrectly classified as real by the model.

**7.3 Audio initial experiments**

Below is a table of the initial training experiments we have conducted by using different, using just a small percentage of the datasets:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | % of data used | Audio was converted into sample rate | Sample length [sec] | Number of samples | Training input shape (Feature vectors, Dimension) | [Epochs, Batch size, Network Size [Mbytes], Validation size] | CNN train time [min] | Results  [Accuracy, loss, Val accuracy, Val loss] |
| 1 | 2-5% of dataset (3 datasets) | 44.1khz | 1 | 5808 samples, 1 sec each | (3267, 26100) | [E=5, **B=32**, Ns=102MB, V=25%] | 7.2 | [A=0.986, L=0.04, VA=0.935, VL=0.266] |
| 2 | 2-5% of dataset (3 datasets) | 44.1khz | 1 | 5808 samples, 1 sec each | (3267, 26100) | [E=5, **B=64**, Ns=102MB, V=25%] | 6.1 | [A=0.985, L=0.042, VA=0.972, VL=0.119] |
| 3 | 2-5% of dataset (3 datasets) | 44.1khz | 1 | 5808 samples, 1 sec each | (3267, 26100) | [E=5, **B=128**, Ns=102MB, V=25%] | 5.5 | [A=0.985, L=0.048, VA=0.97, VL=0.12] |
| 4 | 2-5% of dataset (3 datasets) | 44.1khz | 1 | 5808 samples, 1 sec each | (3267, 26100) | [E=5, **B=256**, Ns=102MB, V=25%] | 5.1 | [A=0.977, L=0.07, VA=0.966, VL=0.126] |
| 5 | 2-5% of dataset (3 datasets) | 44.1khz | 1 | 5808 samples, 1 sec each | (3267, 26100) | [E=5, **B=512**, Ns=102MB, V=25%] | 4.4 | [A=0.972, L=0.083, VA=0.955, VL=0.133] |
| 6 | 5-10% of dataset (3 datasets) | 44.1khz | **1.5** | 9674 samples, 1 sec each | (5441, 39000) | [E=5, **B=512**, Ns=102MB, V=25%] | 16 | [A=0.964, L=0.103, VA=0.919, VL=0.229] |
| 7 | 5-10% of dataset (3 datasets) | 44.1khz | 1 | 20960 samples, 1 sec each | (11790, 26100) | [E=5, **B=512**, Ns=102MB, V=25%] | 20 | [A=0.886, L=0.208, VA=0.923, VL=0.187] |
| 8 | 10-20% of each dataset (3 datasets) | **22.05khz** | 1 | 27368 samples, 1 sec each | (20526, 13200) | [E=5, **B=512**, Ns=102MB, V=25%] | 11.3 | [A=0.961, L=0.052, VA=0.975, VL=0.069] |

As seen in the table, the 8th test delivers a promising result with a relatively low training time when setting a high batch number and converting the audio to a lower frequency. The loss and value loss show a negligible increase, while the accuracy stays high. The conclusion of these experiments was that increasing batch sizes and decreasing the sample rate results in negligible changes in the results. Therefore, we have chosen to convert the datasets to a 22.05khz sample rate and train the model on a batch size of **1024** and 5 epochs.

**7.4 Audio model results**

**Accuracy**

The model training results yielded exceptional accuracy on the training data. See the table below for the full results.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Epochs  Metrics | Epoch 1 | Epoch 2 | Epoch 3 | Epoch 4 | Epoch 5 |
| Accuracy | 0.8463 | 0.9877 | 0.9896 | 0.9917 | 0.9922 |
| Loss | 0.6263 | 0.0298 | 0.0248 | 0.0195 | 0.018 |
| Precision | 0.8534 | 0.9916 | 0.9953 | 0.9967 | 0.9974 |
| Recall | 0.9437 | 0.9904 | 0.9895 | 0.9911 | 0.9912 |
| Val\_accuracy | 0.991 | 0.9911 | 0.992 | 0.9912 | 0.9932 |
| Val\_loss | 0.0264 | 0.0217 | 0.0177 | 0.0217 | 0.0172 |
| Val\_precision | 0.9948 | 0.998 | 0.9975 | 0.9935 | 0.9979 |
| Val\_recall | 0.9922 | 0.989 | 0.9909 | 0.9938 | 0.9922 |

**Accuracy**: Both training and validation accuracy consistently increase over the epochs, indicating that the model is learning to classify speech effectively.

**Loss**: The training loss decreases over time, while the validation loss fluctuates slightly but overall tends to decrease. This means that the model is improving its ability to generalize to unseen data.

**Precision**: Precision also shows an upward trend, indicating that the model is predicting generic speech as synthetic better over time.

**Recall**: Recall remains relatively stable with minor fluctuations. This means that the model is consistently identifying most of the real audio data.

**Confusion Matrix**

From a total of 41758 real speech samples, and 6601 fake speech samples, we got the following results:

* True Positives (TP): 99% of real instances were correctly classified as class **real**.
* True Negatives (TN): 98% of fake instances were correctly classified as class **fake**.
* False Positives (FP): 1% of fake instances were incorrectly classified as class **real** when they belonged to class **fake**.
* False Negatives (FN): 2% of real instances were incorrectly classified as class **fake** when they belonged to class **real**.

As seen in [Figure 19](#f20), it proved to be exceptionally well at classification of generic speech.

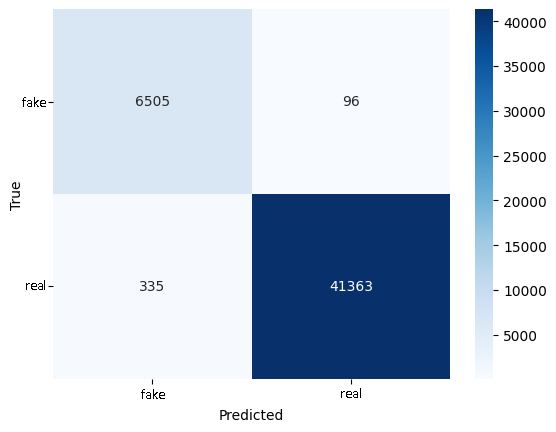


Figure 19: Confusion matrix of the audio model.

**Receiver Operating Characteristic (ROC) Curve**

As seen in the [Figure 20](#f21) below, the ROC curve is exceptionally well, and might mean either an overtrained model or an extremely accurate one. We suspect that the model is accurate in recognizing low-medium quality deepfakes but once we introduce more high quality deepfakes to the validation testing of the model, the ROC curve will certainly worsen.

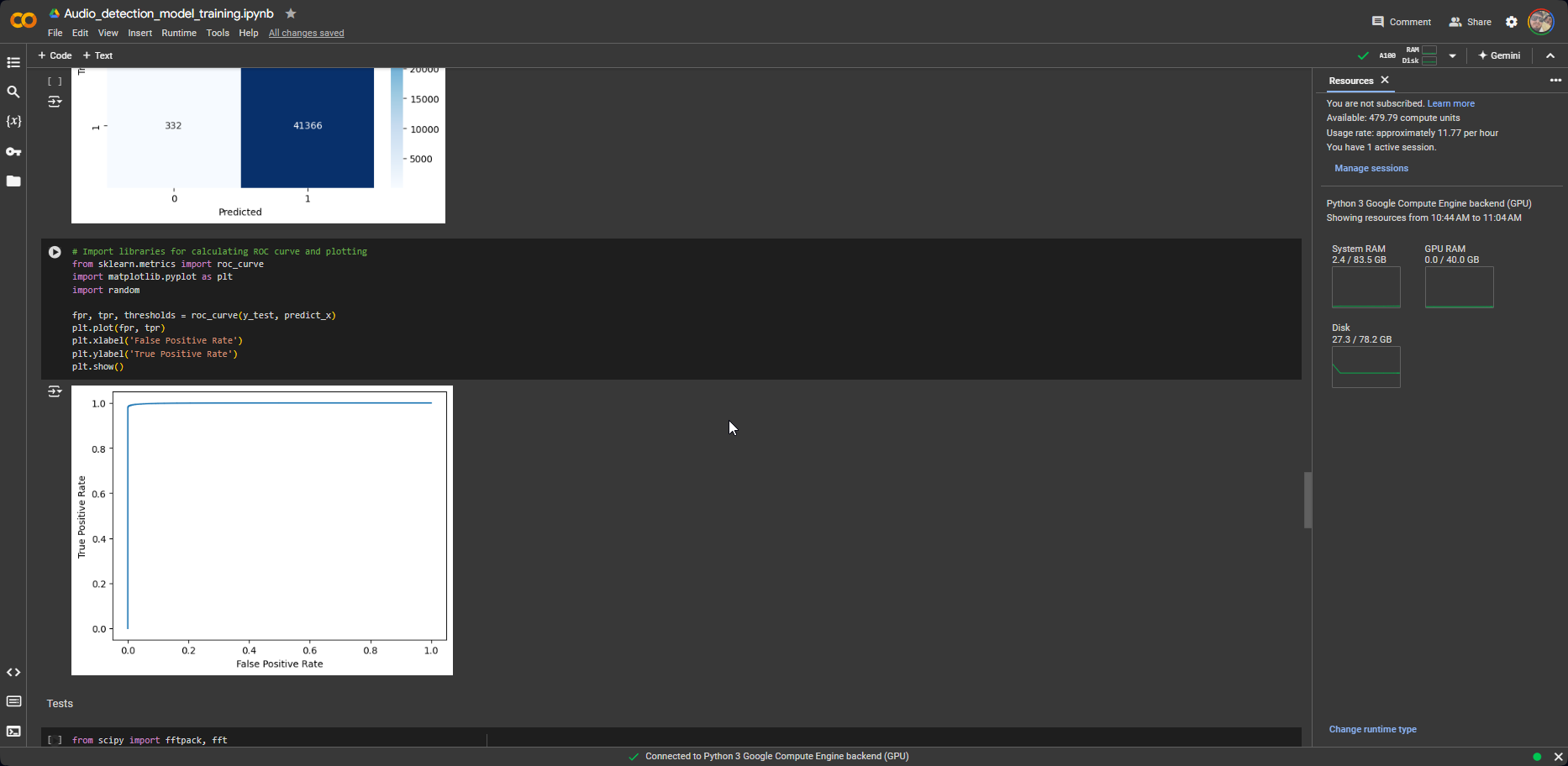


Figure 20: ROC curve of the audio model.

This curve leads us to two main conclusions:

1. Though a batch size of 1024 is very high and might not seem like a factor which could overtrain the model, it still might be too high when training a model on non-high quality deepfakes.   
   This means that in the future, the model should be trained on deepfakes which vary in their quality to cover the greatest number of possibilities and cases for the model to learn.
2. Providing a large amount of various quality deepfakes to the model as training data might alter the model’s performance and lower the accuracy but will prove to be more capable in handling unseen data. When considering the model’s performance on the validation data, it seems to handle partially unseen data well as seen in [Figure 21](#f22_2).

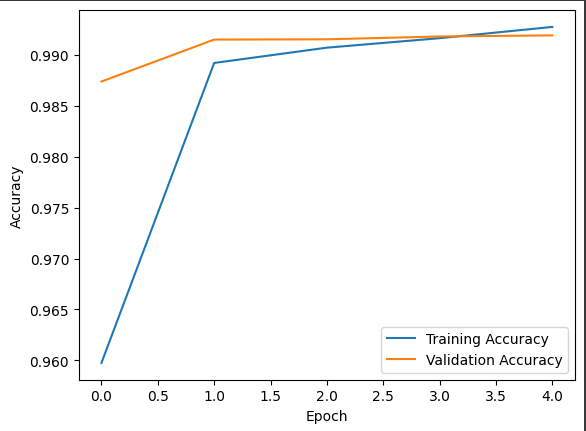


Figure 21: Model accuracy on training and validation data.

1. **Conclusions**

**Project criteria**

In phase A of the Capstone project, we set our detection success criteria to 70-80% for the video model and achieved a final accuracy of ~83%. However, performance was imbalanced, with a detection rate of over 93% for real videos but only 52% for fake videos.

Despite this imbalance, the model demonstrated high specificity for fake videos. This implies that if a video is classified as fake by the model, there is a high probability of it being genuinely fabricated. False positive rates, or instances of real videos being classified as fake, were minimal.

For the audio model, we set our detection criteria to be at the same range of 70-80%. The model proved to be performing well above that range at ~99% for the testing dataset but proves to be inefficient for high-quality deepfakes when tested individually. Combating this inaccuracy on high-quality deepfakes can be greatly improved by training the model on high-quality deepfake datasets which are not yet available to the public.

**Constraint management**

Initial constraint handling was primarily a testing process involving iterative experimentation and comprehensive literature review. While the project's first phase was dedicated to research, practical implementation exposed significant knowledge gaps that necessitated further in-depth study.  
  
For instance, during the development of the video model CNN layer, a prior investigation into video classification techniques identified the (2+1)D convolution as a promising approach.  
However, implementation revealed computational constraints that prevented its utilization. This unforeseen challenge necessitated a return to foundational research to establish an applicable CNN architecture.

**Considerations in decision making**

Time and resource management were critical, yet equally challenging factors in the development of our visual and audio analysis models. The creation of these complex systems required substantial temporal and computational resources. Not only did model development itself demand significant time, but the training process, particularly the feature extraction for video analysis, which consumed approximately five days, also impacted our timeline. To optimize resource allocation, concurrent work on other project components was essential.

Furthermore, computational power directly influenced project duration. As model complexity and data volume increased, so did the processing time. Throughout the development process, a core objective was to achieve optimal model performance while maintaining computational efficiency and within limited computational budgets.

Research [[13]](#b13) suggested extracting bicoherence and bi-spectrum features on top of the already extracted MFCC, delta, and delta2 features. Ultimately, we chose to not extract those features to save on computational speed. Implementing these features as training data as part of continued research can prove beneficial and can improve model accuracy on high-quality generated speech.

**8.1 Key takeaways**

**Video model**

The development of our video analysis model provided valuable insights for potential improvements. Initially, training on the Celeb-DF-V2 dataset, characterized by high-quality and generous fake video content, led to overfitting despite variations in actors and interview settings. This suggests the necessity of incorporating additional datasets, such as FaceForensics++ [[19]](#b19) and DFDC [[20]](#b20), to enhance model generalization. While computational constraints prevented their full utilization, we recognize the potential benefits of employing subsets of these datasets. Furthermore, the lower deepfake quality within these alternative datasets posed an additional challenge.

Fine-tuning on a dataset of high-quality fake and real images produced promising results but was limited by the lack of publicly available deepfake image datasets. Generating synthetic deepfake images through a dedicated model or frame extraction could potentially address this issue, even if at a significant computational cost.

Strict model evaluation was deprived by time and computational limitations, necessitating testing on a reduced dataset and accelerated hyperparameter tuning. Ideally, extensive experimentation and   
in-depth analysis of model components would enable small optimizations.

A critical challenge emerged during the initial model architecture development. A custom 3D convolutional neural network (CNN) exhibited limitations in processing video data, accepting input sequences of less than one second. The development was postponed by this constraint, leading to a prolonged period of investigative research to determine a feasible solution. Ultimately, we concluded that using a pre-trained model for feature extraction proved to be ideal and efficient for this project.

**Audio model**

This model proved to be a success, and we consider it an achievement in this field of fraud-detection using deep learning techniques. Along with the impressive results, we think that additional research and improvements to the audio model and the general preprocessing are still needed.

The lack of sufficient high-quality audio datasets affects the model’s effectiveness over time and might prove to be useless in the coming future since synthetic speech greatly improved over time. We assume that once more high-quality datasets are gathered, the model’s accuracy for very high-quality deepfakes can greatly improve and can also require far less training data to get the optimal effectiveness.

Once more high-quality datasets are gathered, we can proceed to extract additional audio features such as bicoherence and bi-spectrum coefficients and concatenate them on top of the already extracted data for each training vector. This would prove to be computationally possible since we will be using fewer samples overall.   
Additionally, we can deepen the model’s structure by adding extra layers which will be better at detecting different dependencies between different features in synthetic and generic speech samples.

**Overall**

The project proved to be very challenging and complicated, and it was split into three stages of development.

The first stage was preprocessing, where we proceeded to conduct a vast number of tests, experiments, and researched the most optimal ways to preprocess audio and video data.   
We were required figure out ways to fit, scale, normalize, split, pad, and transform the incoming data to different lengths, sizes, and dimensions to find the optimal procedures to preprocess the data for training.

The second stage was the model training stage, where we proceeded to train the video and audio detection models. This required extensive trial and error regarding the depth of each model, the number of layers, activation functions, kernel sizes, and number of filters to apply at each stage.   
This proved to be very time consuming and required many hours of runtime for multiple runs of training different models.

The third stage was developing the web user interface, where we used a JavaScript frontend framework to communicate with a Python backend framework. Connecting between two different languages and establishing a working client-server connection took a decent amount of time to implement and understanding the request-response communication between them proved to be difficult sometimes.

Overall, regardless of the challenges we have faced, we managed to develop a publicly available and user-friendly access to complex deepfake analysis. This will allow us to develop a much more complex system in the future, and we will be able to scale it since the machine learning part is developed in the backend and is disconnected from the frontend.

1. **References**

[1] Florian Zandt, “How dangerous are Deepfakes and Other AI-Powered Fraud?”, <https://www.statista.com/chart/31901/countries-per-region-with-biggest-increases-in-deepfake-specific-fraud-cases/>

[2] Choe Sang-Hun, “South Korean Teenagers Detained Over Deepfake Sexual Images”,   
<https://www.nytimes.com/2024/09/03/world/asia/south-korean-teens-deepfake-sex-images.html>

[3] Heather Barr, “South Korea’s Digital Sex Crime Deepfake Crisis”, <https://www.hrw.org/news/2024/08/29/south-koreas-digital-sex-crime-deepfake-crisis>

[4] Thomas Barrabi, “Elon Musk’s Grok AI chatbot goes viral with mind-blowing deepfakes of Trump, Taylor Swift”, <https://nypost.com/2024/08/22/business/elon-musks-grok-ai-chatbot-goes-viral-with-mind-blowing-deepfakes/>

[5] David Klepper, “Fake babies, real horror: Deepfakes from the Gaza war increase fears about AI’s power to mislead”, <https://apnews.com/article/artificial-intelligence-hamas-israel-misinformation-ai-gaza-a1bb303b637ffbbb9cbc3aa1e000db47>

[6] SolcyréBurga, “How a New Bill Could Protect Against Deepfakes”, <https://time.com/6590711/deepfake-protection-federal-bill/>

[7] Keras 3 API: **InceptionV3**, <https://keras.io/api/applications/inceptionv3/>

[8] Li, Y., Yang, X., Sun, P., Qi, H., & Lyu, S. (2020).   
“**Celeb-DF: A Large-scale Challenging Dataset for DeepFake Forensics**”. In IEEE Conference on Computer Vision and Pattern Recognition (CVPR). **[**[**arXiv:1909.12962**](https://arxiv.org/abs/1909.12962)**]**. <https://github.com/davisking/dlib>

[10] Keras 3 API: **Conv2D layer,** <https://keras.io/api/layers/convolution_layers/convolution2d/>

[11] Keras 3 API: **LSTM layer**, <https://keras.io/api/layers/recurrent_layers/lstm/>

[12] face\_recognition, “The world’s simplest facial recognition API for Python and the command line”, <https://github.com/ageitgey/face_recognition>

[13] [Arun Kumar Singh](https://arxiv.org/search/cs?searchtype=author&query=Singh,+A+K) (1), [Priyanka Singh](https://arxiv.org/search/cs?searchtype=author&query=Singh,+P) (2) ((1) Indian Institute of Technology Jammu, (2) Dhirubhai Ambani Institute of Information and Communication Technology), “Detection of AI-Synthesized Speech Using Cepstral & Bispectral Statistics”, **]arXiv:2009.01934[**

[14] Jiajun Huang, Xueyu Wang, Bo Du, Pei Du, Chang Xu, “DeepFake MNIST+: A DeepFake Facial Animation Dataset”, <https://github.com/huangjiadidi/DeepFakeMnist>

[15] [Nicolas M. Müller](https://arxiv.org/search/cs?searchtype=author&query=M%C3%BCller,+N+M), [Pavel Czempin](https://arxiv.org/search/cs?searchtype=author&query=Czempin,+P), [Franziska Dieckmann](https://arxiv.org/search/cs?searchtype=author&query=Dieckmann,+F), [Adam Froghyar](https://arxiv.org/search/cs?searchtype=author&query=Froghyar,+A), [Konstantin Böttinger](https://arxiv.org/search/cs?searchtype=author&query=B%C3%B6ttinger,+K) , “Does Audio Deepfake Detection Generalize?”, **[**[**arXiv:2203.16263**](https://arxiv.org/abs/2203.16263)**]**.

[16] Fluent speech corpus – speaking recognition voice dataset, <https://www.kaggle.com/datasets/tommyngx/fluent-speech-corpus>

[17] [Jordan J. Bird](https://arxiv.org/search/cs?searchtype=author&query=Bird,+J+J), [Ahmad Lotfi](https://arxiv.org/search/cs?searchtype=author&query=Lotfi,+A) , “Real-time Detection of AI-Generated Speech for DeepFake Voice Conversion”, **[**[**arXiv:2308.12734**](https://arxiv.org/abs/2308.12734)**]**.

[18] dlib: “A toolkit for making real world machine learning and data analysis applications in C++ “, <https://github.com/davisking/dlib>

[19] Andreas Rössler and Davide Cozzolino and Luisa Verdoliva and Christian Riess and Justus Thies and Matthias Nießner. "FaceForensics++: Learning to Detect Manipulated Facial Images", IEEE International Conference on Computer Vision (ICCV), 2019. **[**[**arXiv:1901.08971**](https://arxiv.org/abs/1901.08971)**]**  
<https://github.com/ondyari/FaceForensics>

[20] Brian Dolhansky, Russ Howes, Ben Pflaum, NicoleBaram, and Cristian Canton Ferrer. The deepfake detec-tion challenge (DFDC) preview dataset. arXiv preprintarXiv:1910.08854, 2019. **[**[**arXiv:1910.08854**](https://arxiv.org/abs/1910.08854)**]**<https://ai.meta.com/datasets/dfdc/>

1. **User guide**

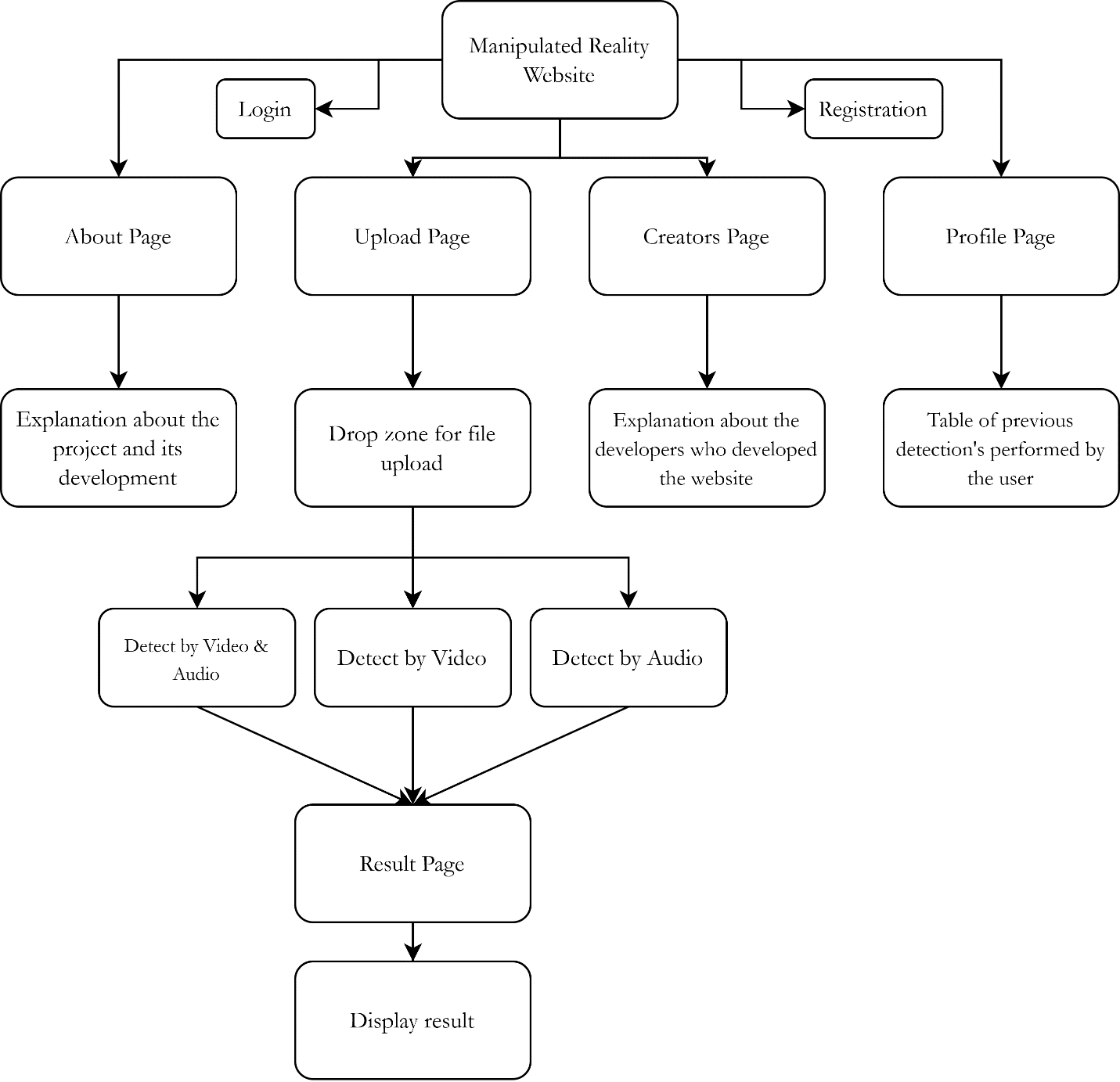
**Introduction**

This section serves as a user guide for our application, a machine learning-based tool designed to detect three distinct types of manipulation within digital content. Users can leverage our pre-trained models to analyze suspicious content and identify potential alterations. To utilize the application, users must first install the necessary software frameworks and libraries. Subsequently, they can launch the web server and interface to fully explore the application's capabilities.

**Website Usage**

Upon successful installation, the application interface will be accessible at http://localhost:3000/. Users must first create an account by providing a unique username, password, and email address. Subsequently, users can select their preferred detection mode: video, audio, or a combination of both.  
After selecting the detection mode, users can upload a compatible media file from their local device. A preview option is available to verify the uploaded content before proceeding. Once the file is validated, the application initiates the detection process. This involves preprocessing the media data and applying the trained model to generate predictions. The results, including the model's output, will be displayed on a dedicated results page.  
Additionally, users can access their profile page to review their detection history. This includes details about the files analyzed, detection modes employed, and corresponding model predictions.

**The website structure:** The entire website structure is depicted in the image below.



**Deepfake detection process**

First, to use this website, the user is required to login or create an account. [Figure 1](#f22) illustrates the login interface, where users can access their accounts or proceed to the registration page if they have not yet created one.

A screenshot of a login screen

Description automatically generated

Figure 1: Login page.

Then, the user can upload video or audio files to undergo deepfake detection [Figure 2](#f23). Upon successful upload, the user can select the detection type (Video, Audio, Both).

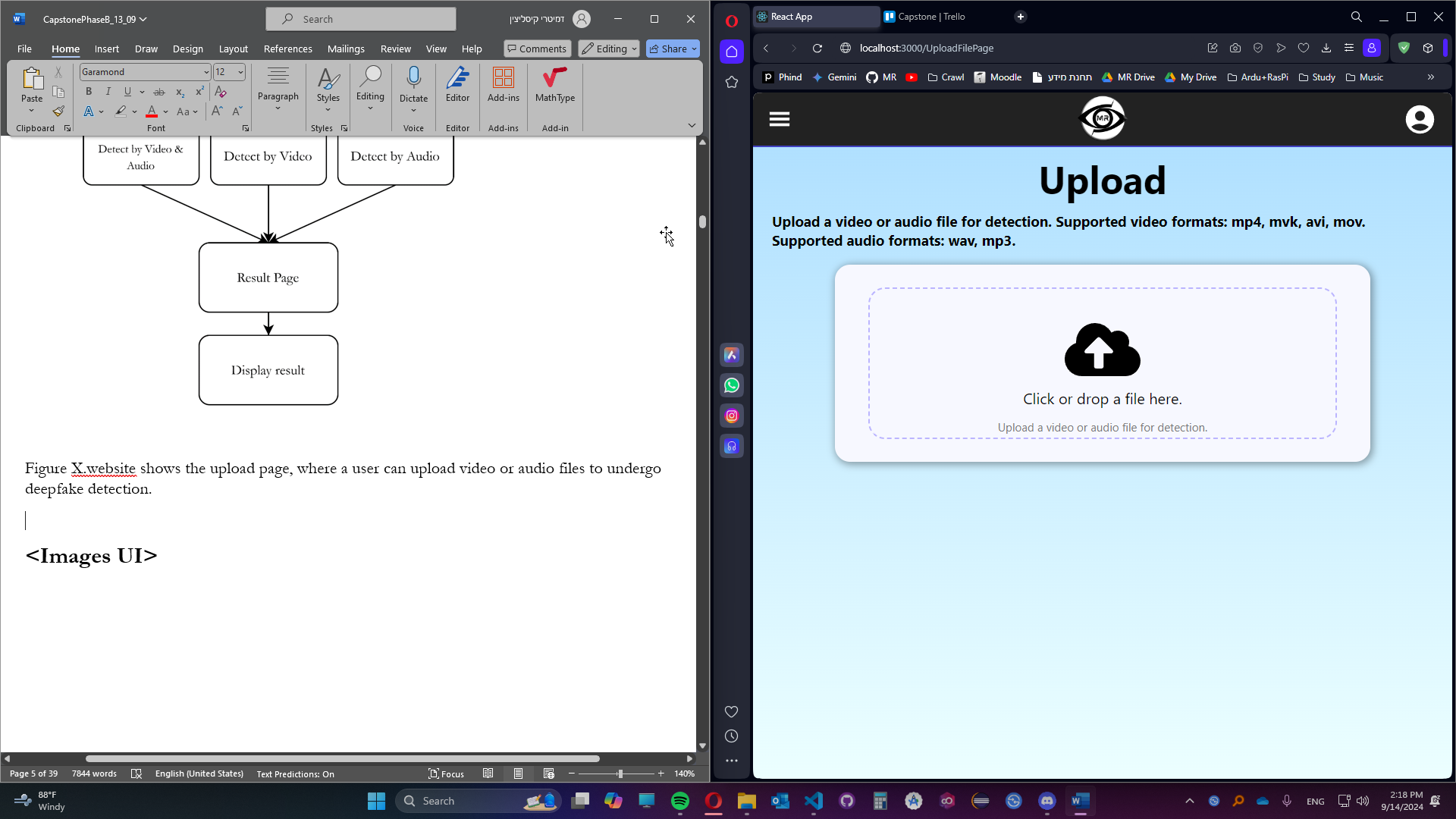


Figure 2: Upload page, user can click or drop a file into the drop zone.

Once the file successfully uploads [Figure 3](#f24), it enables the user to choose from three different options of detection type, video & audio, video only or audio only.

A screenshot of a video file

Description automatically generated

Figure 3: Successfully uploaded file, detection options are visible to the user.

Next, the website sends a request containing the file and detection type to the server. While the server analyzes the file, the user is navigated to the results page [Figure 4](#f25), with a loading bar. Once the video finishes analyzing, the server responds with the percentage estimation for the model being real.

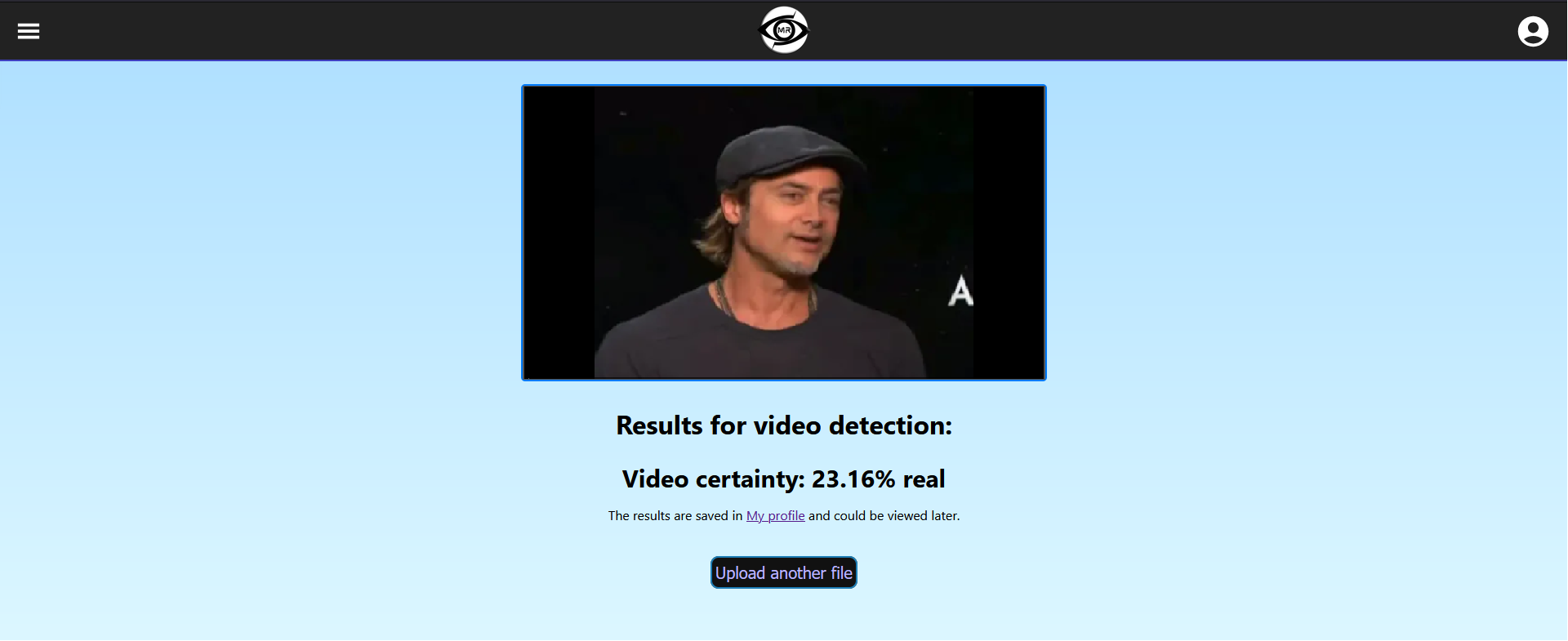


Figure 4: Result page, an example of fake video detection process, getting result of only 23.16% been real.

Once the backend finishes analyzing the file, the results will be saved in the database, and can be viewed by the user from their personal profile [Figure 5](#f26) accessible from the profile icon on the top right corner.

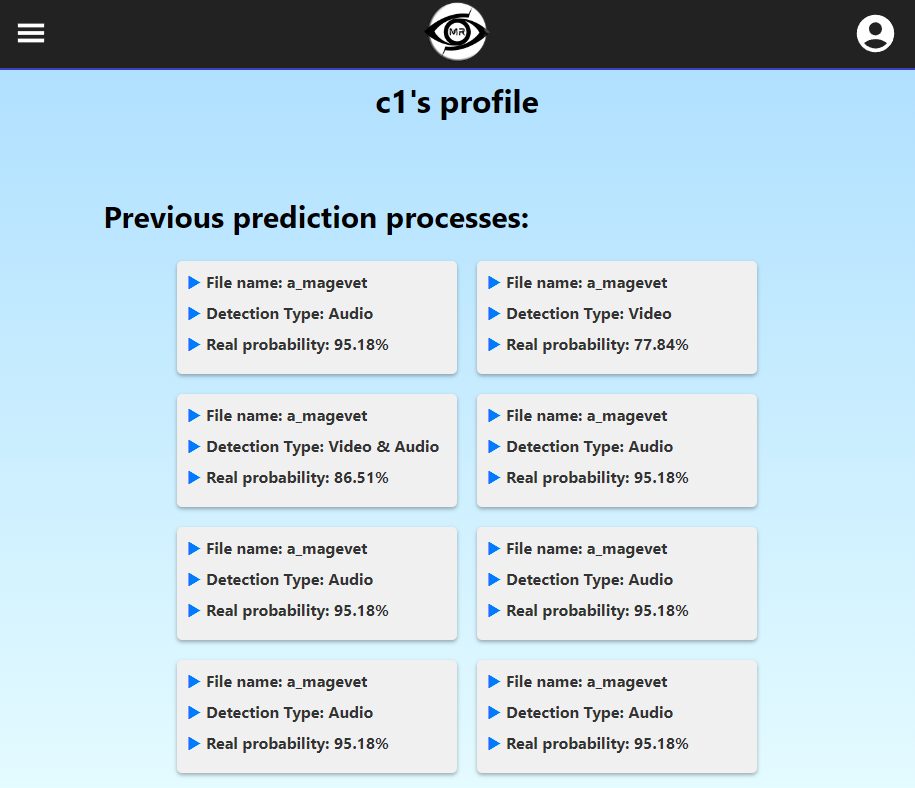


Figure 5: Profile page displays the user his history of previous predictions.

1. **Maintenance guide**

**Description**

This section outlines the procedures for maintaining and potentially enhancing our project. To ensure the application functions correctly on your device, please follow the installation steps described in the **Installation** section. As with any open-source project, you are encouraged to contribute by making modifications or improvements as needed.

**11.1 Installation**

1. To install Manipulated Reality, you need to use **Python 3.11** or higher.
2. Install **Node.js 20.11.1**, which you can download from this [blog release](https://nodejs.org/en/blog/release/v20.11.1).
3. Open Git Bash

Change the current working directory to the location where you want the cloned directory.

Run: git clone <https://github.com/MaximL98/Manipulated-Reality.git>

Press **Enter** to create your local clone.

1. Once you clone the repository, run **pip install -r requirements.txt** (preferable on a virtual environment via [venv](https://docs.python.org/3/library/venv.html) or [Anaconda](https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html))

This will install all the dependencies from the given requirements file.

1. Some users might get an error when installing face\_recognition library, follow the instructions from [this Github repository](https://github.com/z-mahmud22/Dlib_Windows_Python3.x) to install Dlib compiled binary wheels on windows x64 OS.
2. Navigate into the app directory where package-lock.json file is located, meaning the folder named “frontend” (PATH\Manipulated-Reality\frontend).

From there run in the terminal, **npm install**

This command will install all the dependencies related to the website application.

1. Download the models files from the following [Google Drive](https://drive.google.com/drive/folders/1L758Rllh4s8ROEiuiqKHcRh5aefXOxpn?usp=sharing) folder.
2. Place the “Audio\_detection\_model.keras” and “RobustScaler.plk” in the AudioTraining folder, as for the other two files, create a folder named “models” inside the video\_analysis folder and place them there.
3. To run the website, open two terminals. Run **python .\server.py** from the root path (PATH\Manipulated-Reality). In the second terminal navigate to the frontend folder

(cd .\frontend\), and run **npm start**. Which will run the website at <http://localhost:3000/>

**11.2 Video model**

**11.2.1 Video prediction pipeline**

The video deepfake detection pipeline leverages a multitude of libraries and frameworks. [Figure 6](#f6_2) illustrates the dependencies between prediction\_pipeline.py and the external tools it utilizes to accomplish its task. The pipeline entails three key stages: facial frame extraction, feature extraction, and pre-trained model loading.

A blue diagram with text and words

Description automatically generated with medium confidence

Figure 6: Video prediction code structure

**11.2.2 Video model training**

The open-source codebase for video model development was utilized within the Google Colab environment. To expedite training, Google Colab Pro+ was necessary, providing access to hardware accelerators. Alternatively, a local Jupyter Notebook environment equipped with high-performance computing resources could be employed. For data management, Google One Premium was subscribed to, enabling the storage of datasets and intermediate data structures, such as NumPy arrays, in Google Drive.

The video model has three development steps (links to their Google Colab notebook):

1. [video\_preprocessing.ipynb](https://colab.research.google.com/drive/1DPkPfHKcHLx07Ttuu84TYw7KBaWxix-1?usp=sharing) [[Section 1]](file:///C:\Users\Dima\Desktop\Maintenance_Guide%20(1).docx#b1)
2. [fine\_tuning\_pretrained\_inceptionV3.ipynb](https://colab.research.google.com/drive/1mklWJq4dv8FEL36zqVo7fwH4642oVa20?usp=sharing) [[Section 2]](file:///C:\Users\Dima\Desktop\Maintenance_Guide%20(1).docx#b2)
3. [video\_training.ipynb](https://colab.research.google.com/drive/1APlK8I-k4_gIMgdGklSkcba8B_lCpcMa?usp=sharing) [[Section 3]](file:///C:\Users\Dima\Desktop\Maintenance_Guide%20(1).docx#b3)

**Video preprocessing**

The initial stage of our video model development involves a series of preprocessing steps designed to enhance data quality and consistency. These steps include:

1. **Facial Landmark Detection:** Extracting facial regions from video frames using state-of-the-art techniques, such as deep learning-based algorithms, to focus on relevant visual information.
2. **Frame Rate Reduction:** Down sampling the video frame rate to a predetermined level, reducing computational complexity and potentially improving generalization.
3. **Video Duration Capping:** Limiting the maximum duration of video sequences to ensure uniform input sizes and prevent overfitting.
4. **Frame Resizing:** Scaling video frames to a standard resolution, facilitating efficient processing and compatibility with the model architecture.

**Data access and environment setup**

To execute these preprocessing tasks, it is necessary to mount your Google Drive to access the required video data. Follow the instructions below to establish the necessary connection:

# Connect to Google Drive

**from** google**.**colab **import** drive

drive**.**mount**(**‘/content/drive’**)**

For optimal organization within your Google Drive, establish a predetermined quantity of folders. For instance, consider the following structure:

folder\_paths **=** **[**“/content/drive/MyDrive/ManipulatedReality/VideoDatasets/Celeb-DF-V2/Celeb-real”**,**

“/content/drive/MyDrive/ManipulatedReality/VideoDatasets/Celeb-DF-V2/Celeb-synthesis”**]**

save\_folders **=** **[**“/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/Celeb-real”**,**

“/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/Celeb-synthesis”**]**

Structure your dataset into two distinct directories: one containing authentic data and the other synthetic data (**folder\_paths**). Define the output directories where the preprocessed data will be saved as NumPy arrays. Create separate folders for real and fake content (**save\_folders**).

Once the dataset and output directories are configured, proceed to execute the data preprocessing pipeline.

Main**(**folder\_paths**,**save\_folders**)**

To ensure effective model training and evaluation, the processed dataset will be divided into three distinct subsets:

1. **Training Set:** This portion of the data will be used to train the model, allowing it to learn patterns and relationships within the data.
2. **Testing Set:** Once the model is trained, it will be evaluated on this unseen data to assess its performance and generalization capabilities.
3. **Validation Set:** This set is used to fine-tune hyperparameters and prevent overfitting during the training process.

To accommodate these subsets for both real and fake data, six additional folders will be created:

* **Real Data:**
  + Train
  + Test
  + Validation
* **Fake Data:**
  + Train
  + Test
  + Validation

real\_video\_folder\_train **=** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/Celeb-real/train/’

real\_video\_folder\_test **=** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/Celeb-real/test/’

real\_video\_folder\_val **=** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/Celeb-real/val/’

fake\_video\_folder\_train **=** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/Celeb-synthesis/train’

fake\_video\_folder\_test **=** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/Celeb-synthesis/test/’

fake\_video\_folder\_val **=** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/Celeb-synthesis/val/’

Prior to executing the **label\_data** function, it is essential to establish CSV files designated to store the individual path directories for each subcategory.

Paths\_to\_csv **=** **{**

‘train\_df’**:** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/train\_df\_full’**,**

‘test\_df’**:** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/test\_df\_full’**,**

‘val\_df’**:** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/val\_df\_full’

**}**

The **extract\_video\_frame** function offers customizable parameters for frame size, target frames per second, and the total number of saved frames.

This process is computationally intensive, requiring a high-performance graphics processing unit (GPU), we used T4 GPU. Even with a powerful GPU, the operation may take approximately 1-2 days to complete.

**11.2.3 Fine tuning pretrained InceptionV3 model**

To extract features from the dataset, we will leverage the pre-trained InceptionV3 model. This model has been trained on a massive dataset of images, making it well-suited for our task.

To access the necessary files, mount your Google Drive to your Colab notebook using the appropriate previously explained method.

Establish the following directory structure to organize your project:

train\_images\_path **=** ‘/content/drive/MyDrive/ManipulatedReality/ImageDataset/rvf10k/train/’

valid\_images\_path **=** ‘/content/drive/MyDrive/ManipulatedReality/ImageDataset/rvf10k/valid/’

***# Set path to train and validation images that are stored in the drive***

resized\_train\_images\_path **=** ‘/content/drive/MyDrive/ManipulatedReality/ImageDataset/rvf10k/resized\_train/’

resized\_valid\_images\_path **=** ‘/content/drive/MyDrive/ManipulatedReality/ImageDataset/rvf10k/resized\_valid/’

Allocate two separate directories, one for training images and another for validation images.

If necessary, resize all images to a standard dimension (e.g., 224x224 pixels) using image processing libraries. This step is crucial for ensuring consistent input to the model.

Partition the dataset into training and validation sets. Specify the desired proportion of images for each set:

# Define number of train and validation samples

train\_samples **=** 7000

validation\_samples **=** 3000

The fine-tuning step allows to examine verity of different hyperparameters, such as:

# Dimensions of our images

img\_width**,** img\_height **=** 224**,** 224

***# Set parameters***

batch\_size **=** 16

num\_classes **=** 2

epochs **=** 50

activation **=** ‘relu’

min\_delta**=**0

patience**=**4

dropout**=**0.2

learning\_rate**=**0.0001

# Data augmentation for training images

train\_datagen **=** ImageDataGenerator**(**

preprocessing\_function**=**preprocess\_input**,** ***# Preprocess images for InceptionV***

rotation\_range**=**40**,** ***# Randomly rotate images by up to 40 degrees***

width\_shift\_range**=**0.2**,** ***# Randomly shift images horizontally by up to 20% of the width***

height\_shift\_range**=**0.2**,** ***# Randomly shift images vertically by up to 20% of the height***

shear\_range**=**0.2**,** ***# Shear images by up to 20 degrees***

zoom\_range**=**0.2**,** ***# Randomly zoom images by up to 20%***

horizontal\_flip**=**True**,** ***# Randomly flip images horizontally***

fill\_mode**=**’nearest’**,** ***# Fill in newly created pixels with the nearest pixel value***

validation\_split**=**0.2 ***# Reserve 20% of data for validation***

**)**

***# Data augmentation for validation images (same as training for consistency)***

val\_datagen **=** ImageDataGenerator**(**

preprocessing\_function**=**preprocess\_input**,**

rotation\_range**=**40**,**

width\_shift\_range**=**0.2**,**

height\_shift\_range**=**0.2**,**

shear\_range**=**0.2**,**

zoom\_range**=**0.2**,**

horizontal\_flip**=**True**,**

fill\_mode**=**’nearest’**,**

validation\_split**=**0.2

**)**

***# No data augmentation for testing***

test\_datagen **=** ImageDataGenerator**(**

rescale**=**1**/**255

**)**

***# Generate training data***

train\_generator **=** train\_datagen**.**flow\_from\_directory**(**

resized\_train\_images\_path**,**

target\_size**=(**img\_width**,** img\_height**),**

shuffle**=**True**,**

seed**=**20**,**

batch\_size **=** batch\_size**,**

class\_mode**=**’binary’**,**

**)**

***# Generate validation data***

validation\_generator **=** val\_datagen**.**flow\_from\_directory**(**

resized\_valid\_images\_path**,**

target\_size**=(**img\_width**,** img\_height**),**

seed**=**20**,**

batch\_size**=**batch\_size**,**

class\_mode**=**’binary’**,**

**)**

The fine-tuned model and its associated weights will be stored independently in designated Google Drive directories, as specified:

# Save model weights

model**.**save\_weights**(**‘/content/drive/MyDrive/ManipulatedReality/Fine-TunedModel/inceptionv3\_tuned\_weights\_full\_22\_07.weights.h5’**)**

***# Save model itself***

model**.**save**(**‘/content/drive/MyDrive/ManipulatedReality/Fine-TunedModel/inceptionv3\_tuned\_model\_full\_22\_07.keras’**)**

This process is computationally intensive, requiring a high-performance graphics processing unit (GPU), we used T4 GPU. Even with a powerful GPU, the operation may take approximately 12 hours to complete.

**11.2.4 Video training**

**Improved Text:**

This final stage of the video model development involves feature extraction and LSTM modelling. Our fine-tuned model from the previous step is used to extract features from the processed data. These extracted features are subsequently fed into the LSTM model.

Prior to proceeding, we mount Google Drive to the notebook to access the necessary CSV files. These files, created in the initial step, contain the processed data. For instance:

# Paths to csv’s file that are located in the drive

paths\_to\_csv **=** **{**

‘train\_df’**:** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/train\_df\_full.csv’**,**

‘test\_df’**:** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/test\_df\_full.csv’**,**

‘val\_df’**:** ‘/content/drive/MyDrive/ManipulatedReality/NumpyConvertedDatabases/Celeb-DF-v2/val\_df\_full.csv’

**}**

After loading the data frames, we load the fine-tuned model:

# Load fine-tuned model from Google drive

**from** keras**.**models **import** load\_model

path\_to\_model **=** ‘/content/drive/MyDrive/ManipulatedReality/Fine-TunedModel/inceptionv3\_tuned\_model\_full\_22\_07.keras’

fine\_tuned\_model **=** load\_model**(**path\_to\_model**)**

fine\_tuned\_model**.**summary**()**

Following the successful loading of the dataset and model, the feature extraction process was initiated. This computationally intensive task necessitates a high-performance GPU. With adequate GPU resources, the process is anticipated to require approximately 2-3 days for completion. To mitigate the limitations imposed by Google Colab’s runtime constraints, the Python **pickle** library was employed to implement checkpointing, enabling the process to be paused and resumed at a later time.

**Try:**

# Attempt to resume from a previous checkpoint

start\_idx**,** frame\_features**,** labels **=** resume\_from\_checkpoint**(**save\_path**)**

print**(**f”Resuming from checkpoint … “**)**

**except** FileNotFoundError**:**

***# Start from scratch if no checkpoint found***

print**(**f”No checkpoint found … “**)**

start\_idx **=** 0

frame\_features **=** np**.**zeros**(**shape**=(**num\_samples**,** MAX\_SEQ\_LENGTH**,** NUM\_FEATURES**),** dtype**=**”float16”**)**

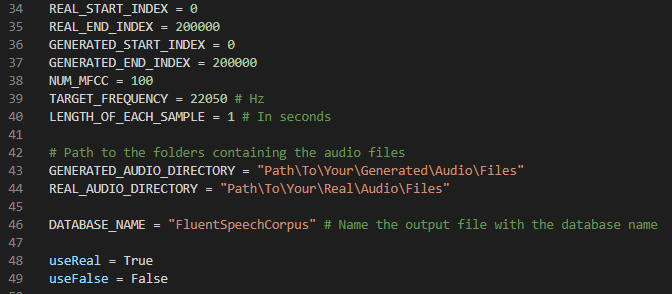
Following the completion of the feature extraction process, the extracted data, stored in NumPy array format, will be persisted to Google Colab to avoid redundant execution of this step prior to LSTM model training. Subsequently, the LSTM model architecture will be defined, the pre-extracted feature data will be loaded, and the model will be trained. The training process is anticipated to conclude within a few minutes. However, it is essential that this training be conducted on an A100 GPU due to the substantial memory requirements of the model, which approximate 70GB of random-access memory (RAM) when loading the entire dataset.

**11.3 Audio model**

**11.3.1 Preprocessing datasets**

The process of preprocessing audio to create the dataset for training the model is simple.The file “Manipulated-Reality\AudioTraining\Pre\_process\_audio.py” is responsible for preprocessing an entire dataset into two NumPy files: feature set and label set.

It needs two paths to two folders, one folder containing fake audio recordings, the other folder containing real audio recordings. The code goes into each folder and preprocesses multiple audio files. At the top level of the code, there are multiple variables which we will now explain:

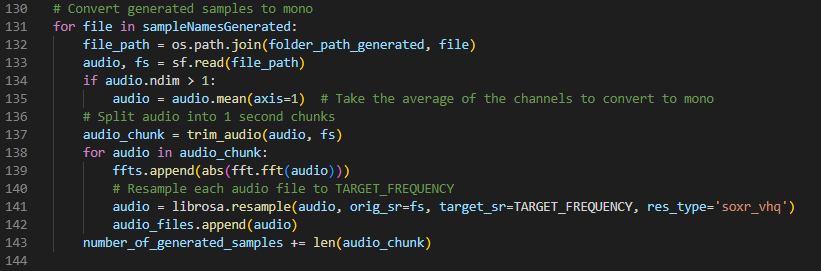


1. **REAL\_START\_INDEX**: In the folder of real audio data, features will be extracted from this file number onwards. Index out of range is handled in the code.
2. **REAL\_END\_INDEX**: In the folder of real audio data, features will be extracted up to this file number. Index out of range is handled in the code.
3. **GENERATED\_START\_INDEX**: In the folder of fake audio data, features will be extracted from this file number onwards. Index out of range is handled in the code.
4. **GENERATED\_END\_INDEX**: In the folder of real audio data, features will be extracted up to this file number. Index out of range is handled in the code.
5. **NUM\_MFCC**: The number of MFCC coefficients to extract from each sample.  
   MFCC coefficients represent different dependencies of frequencies between other frequencies.
6. **TARGET\_FREQUENCY**: The frequency to which you want to normalize the audio to. For example, if the original audio frequency is 44.1khz and the given target frequency is 22.05khz. The audio file will undergo down sampling from 44.1khz to 22.05khz before feature extraction.
7. **LENGTH\_OF\_EACH\_SAMPLE**: To ensure that the preprocessed data will be of the same dimension, each audio file is split into multiple chunks of LENGTH\_OF\_EACH\_SAMPLE time. For example, an audio recording of length 3.77 seconds and LENGTH\_OF\_EACH\_SAMPLE=1, it will split the recording into 4 equal 1 second chunks, where the last chunk will be padded with 0’s.

Note that any chunk with less than 75% of the desired length of each sample will be discarded.

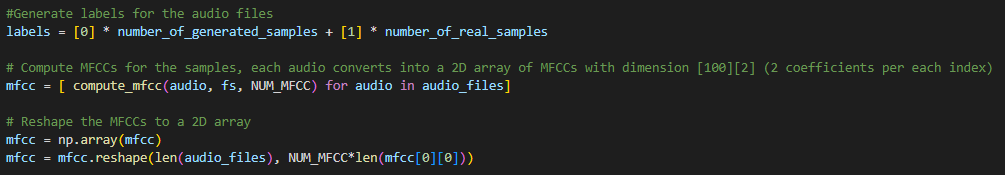
1. **GENERATED\_AUDIO\_DIRECTORY**: The path to the folder containing only fake audio speech.
2. **REAL\_AUDIO\_DIRECTORY**: The path to the folder containing only real audio speech.
3. **DATABASE\_NAME**: The name of your database, can be named any name you’d like.
4. **useReal**: Choose true if you want to extract features from the real audio folder. Otherwise, the real speech audio won’t be preprocessed.
5. **useFake**: Choose true if you want to extract features from the fake audio folder. Otherwise, the fake speech audio won’t be preprocessed.

Next, the code goes over every audio recording in each folder, splits it into LENGTH\_OF\_EACH\_SAMPLE chunks, resamples is to TARGET\_FREQUENCY, and adds the chunks into a list. There is a single list for generated and real audio chunks.



At the end of the process, audio\_files is a list containing N number of generated chunks and M number of real chunks. E.g. [generated, generated, …, generated, real, real, …, real].

Then, a list of labels is created and the first N values in the list are “0” for “fake” and the M final values in the list are “1” for “real”.



Next, every audio chunk goes through the MFCC feature extraction. It is very important to **keep the number of MFCC features the same for all datasets** you want to preprocess.

A screen shot of a computer code

Description automatically generated

After the MFCC feature extraction finishes, the same MFCC features are used to create the first order and second order derivatives called delta and delta2.

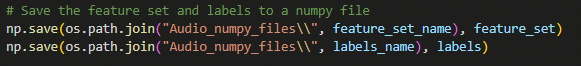
A computer screen shot of text

Description automatically generated

After finishing, it then creates a single NumPy array by concatenating the features together.



Finally, the code saves the feature set and labels set as .npy files to be used for training a model.



**Example:**

Given the following values:

1. **REAL\_START\_INDEX**: 0
2. **REAL\_END\_INDEX**: 1000
3. **GENERATED\_START\_INDEX**: 0
4. **GENERATED\_END\_INDEX**: 1000
5. **NUM\_MFCC**: 100
6. **TARGET\_FREQUENCY**: 22050
7. **LENGTH\_OF\_EACH\_SAMPLE**: 1
8. **GENERATED\_AUDIO\_DIRECTORY**:
9. **DATABASE\_NAME**: SomeNewDataset.

The output files name will look like this:

0-1000\_(REAL)\_0-1000\_(FAKE)\_1000ms\_for\_sample\_22050hz\_frequency\_ SomeNewDataset \_dataset\_**feature\_set**.npy

0-1000\_(REAL)\_0-1000\_(FAKE)\_1000ms\_for\_sample\_22050hz\_frequency\_ SomeNewDataset \_dataset\_**labels\_set**.npy

Now you are ready to train your model.

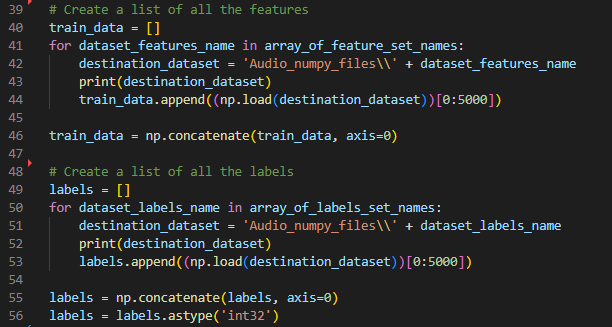
**11.3.2 Training the audio model**

The file “Train\_audio\_model.py” contains the code to train a sequential CNN model from the preprocessed datasets we have created.   
The first step is to write the relative path of the saved datasets.

A black screen with white text

Description automatically generated

Next, the code creates a list of the feature sets and a list for the labels set.

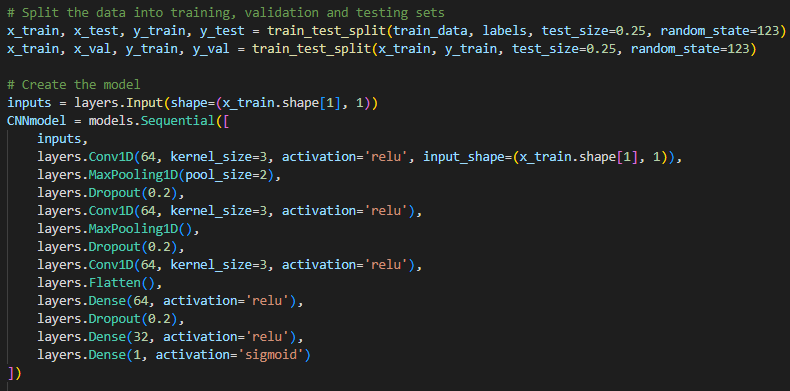


Next, the code creates a scaler to deal with outliers in the data. This is done so that far outliers won’t affect the entire skewness of the data. It is then saved so it can be used later, in the prediction phase.

A computer screen with text

Description automatically generated

Then, the code splits the code into test, train, and validation. Additionally, the model structure is created.



It is recommended to play around with the layers to get different results during further research. Try switching the number of filters, the kernel size and the activation functions.

**Important:** The last layer **must** have a “Sigmoid” activation function, and 1 filter.

Finally, the model is compiled using the metrics we have set, with a Binary cross entropy loss function, the data is fed to the model and the model is trained and saved.

A computer screen with text on it

Description automatically generated

**11.3.3 Preprocessing audio for detection**

The code structure for the audio prediction process is depicted in the picture below. It describes the usage of the Prediction\_pipeline.py file’s predict() function to call the predictSingleAudioFile.py file’s predict\_single\_audio\_file.py function. This file imports both the feature extracting file and the audio model. It then extracts the audio features from the feature extractor and loads the audio model to predict the audio features that were extracted.

A diagram of a audio library

Description automatically generated

Figure 7: Audio prediction code structure

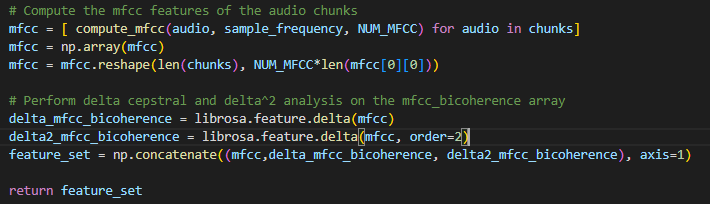
The file “Manipulated-Reality\AudioTraining\Feature\_Extraction\_from\_sample.py” has a function audio\_file\_feature\_extractor(audio\_path, num\_mfcc, target\_freq, length\_of\_each\_sample).

The function receives 4 input variables:

* 1. **audioFilePath**: the path in which the audio file is currently stored. This essentially goes into the audio file’s location and reads it from there.
  2. **NUM\_MFCC**: the number of MFCC coefficients to extract from each sample.  
     MFCC coefficients represent different dependencies of frequencies between each other.
  3. **TARGET\_FREQUENCY**: the frequency to which you want to normalize the received audio to. For example, if the original audio frequency is 44.1khz and the given target frequency is 22.05khz. The audio file will undergo down sampling from 44.1khz to 22.05khz before feature extraction.
  4. **LENGTH\_OF\_EACH\_SAMPLE**: To ensure that the preprocessed data will be of the same dimension, each audio file is split into multiple chunks of LENGTH\_OF\_EACH\_SAMPLE time. For example, an audio recording of length 3.77 seconds and LENGTH\_OF\_EACH\_SAMPLE=1 will split the recording into 4 equal 1 second chunks, where the last chunk will be padded with 0’s.  
     Note that any chunk with less than 75% of the desired length of each sample will be discarded.

Similarly to the [preprocessing datasets section](#a1), the code is thoroughly commented and is responsible for multiple preprocessing steps. The code loads an audio file, extracts its frequency, splits it into chunks, and then resamples the audio into the desired TARGET\_FREQUENCY.

Then, each chunk undergoes feature extraction.  
The feature extraction includes 3 different feature types: MFCC, delta, and delta2. The function returns a 2-dimensional Numpy array of the concatenated features.



**Complete example:**

Audio file: 3.77 second recording.  
NUM\_MFCC: 100.  
TARGET\_FREQUENCY: 22050.  
LENGTH\_OF\_EACH\_SAMPLE: 1.

The code will produce 4 vectors of size 13200, a vector for each second of the audio file. The vector will look like [4400 MFCC features, 4400 delta features, 4400 delta2 features].

**Good to know:** Choosing 100 MFCC features as the default for this project was for the sake of simplicity. You might choose to add more MFCC features to increase the vector size and in turn, this might aid in improving the model’s accuracy. Based on the final model’s results, adding more MFCC’s is pointless since the model’s accuracy is very good. You might also choose to lower the sampling frequency to lower computational costs without impacting the accuracy. The choice is yours.

**Important:** MFCC features **must** be the same across all data. Changing the MFCC value will change the vector size and will not fit to the model’s input layer.

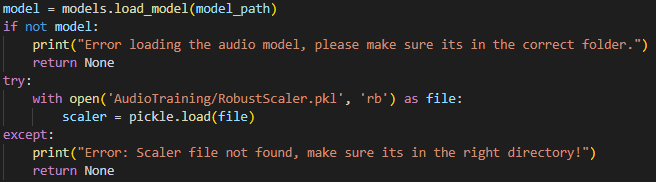
The file “Manipulated-Reality\AudioTraining\predictSingleAudioFile.py” takes three variables as input, an audio file path, the path of the detection model and the path of the scaler.



It then preprocesses the audio file according to the file path provided and saves the preprocessed audio as a list of vectors (A vector per each second of the audio) in the variable “pre\_processed”.



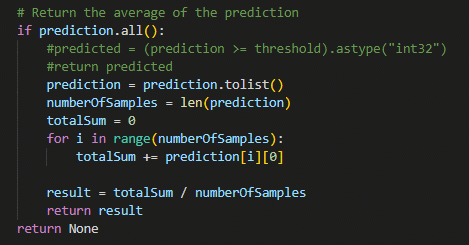
Next, the model and the scaler are loaded.



Finally, the preprocessed chunks go through rescaling using the saved scaler and is predicted by the model.



Then, the average of the results is calculated and a value in the range of 0-1 is returned.



**11.4 Frontend**

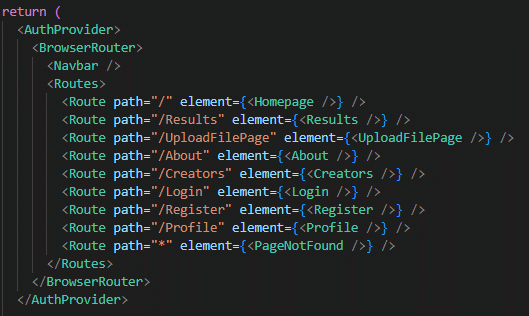
The code structure of our frontend is accessed from the file App.js which initiates and connects between all pages of our application. The client/user gets access to App.js when starting the application and React is employed to allow the logic of navigation and file uploading in the different pages of the app.

A blue diagram with black text

Description automatically generated

Figure 8: Frontend code structure

The root directory of the frontend is “Manupulated-Reality\frontend”. According to the file App.js, all the available paths of the website are located in the return section of the function. The paths are returned as React components and can be edited by clicking Ctrl + click on the green text.



Available pages are: Homepage, Results, UploadFilePage, About, Creators, Login, Register, Profile, and PageNotFound.

**11.5 Backend**

The code structure of our backend for the web application includes server.py which includes:

* **File Upload:** Upload video or audio files directly for processing.
* **Automatic Audio Extraction (if applicable):** For video uploads, the system can automatically extract the audio stream for analysis.
* **Model Predictions:** Leverage powerful video and audio prediction models to generate insights from uploaded content.
* **User Management:** Securely register and login users using a Python script (db\_control.py) that facilitates connection to a local SQLite3 database.
* **Data Delivery:** The backend server efficiently retrieves and transmits the necessary data to the frontend application for user display.

A diagram of a software development

Description automatically generated with medium confidence

Figure 9: Backend code structure

The primary backend component designated "server.py", functions as an API that interacts with the frontend to facilitate user requests. This interaction entails the exchange of data connected to the user's current location within the application (i.e., the specific page being accessed) and the desired actions they intend to execute.

For instance, consider the upload page scenario. Here, the user initiates a file upload operation and concurrently selects a specific detection type. This selection serves as a directive for the server, enabling it to confirm the appropriate actions to execute and the relevant data to be returned to the frontend.

**if** detection\_type **==** 'va'**: # VA stands for video & audio**

**if** allowed\_file\_video**(**file**.**filename**):**

print**(**"Video and audio"**)**

file**.**save**(**video\_path**)**

**# Extract the audio from the video**

paths **=** utils**.**extract\_audio**(**video\_path**,** audio\_path**)**

**if** **not** paths**:**

**return** jsonify**([**video\_path**]),** 200

**return** jsonify**([**video\_path**,** audio\_path**]),** 200

**if** detection\_type **==** 'v'**: # V stands for video only**

**if** allowed\_file\_video**(**file**.**filename**):**

print**(**"Video only"**)**

file**.**save**(**video\_path**)**

**return** jsonify**([**video\_path**]),** 200

**else: # Else A which stands for audio only**

**if** allowed\_file\_audio**(**file**.**filename**):**

print**(**"Audio only"**)**

file**.**save**(**audio\_path**)**

**return** jsonify**([**audio\_path**]),** 200

**# If its a video file and the user want only to test the audio**

**elif** allowed\_file\_video**(**file**.**filename**):**

file**.**save**(**video\_path**)**

**# Extract audio from the video**

paths **=** utils**.**extract\_audio**(**video\_path**,** audio\_path**)**

os**.**remove**(**video\_path**)**

**if** **not** paths**:**

**return** jsonify**(),** 200

**return** jsonify**([**audio\_path**]),** 200

The data is subsequently transmitted to the results page, where the detection process is initiated, dependent on the specified detection type.

# If the detection type is video & audio

**if** video\_path **!=** "undefined" **and** audio\_path **!=** "undefined"**:**

video\_result **=** prediction\_pipeline**.**predict**(**video\_path**)**

audio\_result **=** predictSingleAudioFile**.**predict\_single\_audio\_file**(**audio\_path**)**

data **=** **[**video\_result**,** audio\_result**]**

video\_path\_insert **=** video\_path**.**replace**(**'/'**,** '.'**)**

**# Inserting the data after the processes**

append\_data**(**username**,** "Video & Audio"**,** video\_path\_insert**.**split**(**'.'**)[-**2**],** "some path"**,(**video\_result **+** audio\_result**)/**2**)**

# If the detection type is video only

**elif** video\_path **!=** "undefined"**:**

video\_result **=** prediction\_pipeline**.**predict**(**video\_path**)**

data **=** **[**video\_result**]**

video\_path\_insert **=** video\_path**.**replace**(**'/'**,** '.'**)**

**# Inserting the data after the processes**

append\_data**(**username**,** "Video"**,** video\_path\_insert**.**split**(**'.'**)[-**2**],** "some path"**,(**video\_result**))**

# If the detection type is audio only

**elif** audio\_path **!=** "undefined"**:**

print**(**f"audio\_path={audio\_path}"**)**

audio\_result **=** predictSingleAudioFile**.**predict\_single\_audio\_file**(**audio\_path**)**

data **=** **[**audio\_result**]**

audio\_path\_insert **=** audio\_path**.**replace**(**'/'**,** '.'**)**

print**(**audio\_path\_insert**)**

**# Inserting the data after the processes**

append\_data**(**username**,** "Audio"**,** audio\_path\_insert**.**split**(**'.'**)[-**2**],** "some path"**,(**audio\_result**))**

**else:**

print**(**"Error, both of the paths are None!"**)**

**return** "Error, both of the paths are None!"

The user registration and login functionalities are implemented within the backend component. This is achieved through the db\_control.py script, which provides the necessary functions for interacting with the local SQLite3 database.

Register user:

**if** username **and** password **and** email**:**

**if** register\_user**(**username**,** password**,** email**):**

**return** jsonify**(**"USER REGISTERED"**),** 200 ***# Redirect to login page after successful registration***

**else:**

***# Handle registration failure (e.g., user already exists)***

**return** jsonify**(**"ERROR, username or email already used."**),** 200

**else:**

**return** jsonify**(**"ERROR, please fill up all fields!"**),** 200

Login user:

**if** username **and** password**:**

**if** authenticate\_user**(**username**,** password**):**

***# Redirect to protected area or home page***

**return** jsonify**(**"USER FOUND"**),** 200

**else:**

***# Handle login failure (e.g., incorrect credentials)***

**return** jsonify**(**"USER NOT FOUND"**),** 200

**else:**

**return** jsonify**(**"ERROR, please fill up all fields!"**),** 200

**Thanks for reviewing this Capstone project!**

We hope you enjoyed,

Maxim L. and Dima K.

1. Generative adversarial networks are two models of neural networks that compete against each other. The first network generates images, and the second one identifies the mistakes. They then work in a continuous loop to try and improve each other. [↑](#footnote-ref-1)
2. An Autoencoder is a type ofneural network model that is trained to reconstruct its output data (such as images, text, audio, etc.). The model flow includes compressing the input, decompressing the output, and comparing both input and output using a loss function. [↑](#footnote-ref-2)
3. LSTM (Long Short-Term Memory) networks are a type of recurrent neural network (RNN) specifically designed to handle long-term dependencies in sequential data. Unlike traditional RNNs, they have a special memory cell that can store information for longer periods. [↑](#footnote-ref-3)
4. MFCC are features extracted from an audio signal, widely used in speech and audio processing. They represent the spectral characteristics of sound in a manner that is suitable for various machine learning tasks. The coefficients capture the shape of the power spectrum of a sound signal, emphasizing features of the audio signal that are important for human speech perception. [↑](#footnote-ref-4)
5. 3 Deltas are used to capture the rate of change of the spectral characteristics of an audio signal over time. They provide information about how the sound's spectral properties are evolving, which is crucial for representing dynamic aspects of speech, such as pitch contour and formant transitions.

   Essentially, deltas help to model the dynamic features of an audio signal, making it possible to differentiate between sounds that have similar static spectral properties but differ in their temporal evolution. [↑](#footnote-ref-5)
6. GRU (Gated Recurrent Unit) is a type of recurrent neural network (RNN) designed to handle sequential data.

   Unlike traditional RNNs, GRUs use gates to control the flow of information, allowing them to effectively process long-term dependencies. GRUs are often considered a simpler alternative to LSTM networks but can achieve similar performance in many cases. [↑](#footnote-ref-6)