User Manual

Project Mozart
iOS Personal Voice/PlayHT/FastSpeech2 Synthetic Voice Detection and the Impacts of Noise

---

Contributing Authors: Branch Hill and Jacob Pomatto
Dec 7, 2023

Table of Contents

Table of Figures

Table of Tables

Table of Scripts

Table of Acronyms

Acronym	Meaning
AD	Audio Deepfake
ANN	Artificial Neural Network
AR-DAD	Arabic Diversified Audio Dataset
ASV	Automatic Speaker Verification
AWGN	Additive White Gaussian Noise
BPTT	Backpropagation Through Time
CNN	Convolution Neural Network
CQCC	Constant Q Cepstral Coefficients
DBiLSTM	Deep Bidirectional Long Short-Term Memory
DL	Deep Learning
DNN	Deep Neural Network
DT	Decision Tree
DTW	Dynamic Time Warping
EER	Equivalent Error Rate
FoR	Fake or Real
FQT	Final Qualification Test
GPG	GNU Privacy Guard
HLL	Human in Loop Learning
HMM	Hidden Markov Model
HTTPS	Hypertext Transfer Protocol Secure
KNN	K-Nearest Neighbors
LFCC	Linear Frequency Cepstral Coefficient
LGBM	Light Gradient-Boosting Machine
LLR	Log-Likelihood Ratios
LR	Linear/Logistic Regression
LSTM	Long Short-Term Memory
MFCC	Mel Frequency Cepstral Coefficient
RF	Random Forest

RNN	Recurrent Neural Network
SNR	Signal-Noise Ratio
SSAD	Semi-Supervised Anomaly Detection
SSH	Secure Shell Protocol
STFT	Short-Time Fourier Transform
STN	Spatial Transformer Network
SVM	Support Vector Machine
TCN	Temporal Convolution Network
WSL	Windows Subsystem for Linux

Table 1, Table of Acronyms

Introduction

The user manual outlines the major features of the software and demonstrates how to effectively use the system. This document will cover the background, purpose, system usage, results, and possible future work in this area.

Project Background

Deep Fake Speech detection is an area of research that is starting to emerge more as the tools for manipulating audio or synthesizing fake audio become more prevalent. When our team initially started this project, we aimed to create a system that could predict whether audio was fake or authentic with a high degree of accuracy. Once we developed the system and created a baseline for the metrics, we started to explore the impact of introducing noise to the data. We chose to research this area because it seemed like a simple way to manipulate the data and it had a significant impact on the accuracy of the system. In some instances, the presence of noise decreased the accuracy of the model by 20-30%.

User Manual Purpose

The user manual will provide guidance for setting up the project, manipulating data, using scripts to train the model, and analyzing the results of the software. It will also cover some of the measures we took to mitigate the issues that come along with introducing noise into the dataset. This document will demonstrate how to reproduce the project environment and results for anyone that wants to use this project as a starting point for further research.

Project Overview

Our team leveraged Python, TensorFlow, Keras Models, and other frameworks to develop the classification model. We also used data that had been used in previous research related to this area of machine learning. The technology and data will be covered in later sections.

Project Objectives and Tasks

- Develop a classification model for fake and real speech recognition
- Determine the impact of adding noise into the dataset
- Mitigate the issues that come along with introducing noise into the dataset

Scope and Target Audience

This project is intended for researchers and data scientists that are working on machine learning and deep fake audio detection. This project and user manual can be used to continue research in this area and addresses some of the issues that are posed by introducing noise to the data.

Systems Requirements

This document assumes that the user is operating on a machine running Windows 11 or Windows 10 Build 19045 or later. Although it is not required, it is recommended that the machine have at least 16 GB of RAM and an NVIDIA GPU for best performance.

Setting Up Development Environment

Setting up the development environment requires installing several programs before downloading the codebase and installing dependencies for the system.

Installing WSL

Windows Subsystem for Linux, or WSL, allows developers to run a Linux environment on their Windows machine without the need for a separate virtual machine. WSL is a dependency of Docker Desktop. WSL can be installed using the following steps.

1. Open the Windows Start menu.
2. Select the search bar and search “Powershell”.
3. Right click the powershell application and select “Run as administrator”.
4. Enter the “wsl --install” command into the powershell terminal.
5. Restart the computer.
6. Once the computer has been restarted, open the command prompt application and run the wsl command to verify that it has been installed.

Installing Git for Windows

Git for Windows is a common tool for managing source code and documentation. Git for Windows can be installed using the following steps.

1. Download the Git for Windows installer from the following link: https://github.com/git-for-windows/git/releases/download/v2.42.0.windows.2/Git-2.42.0.2-64-bit.exe
2. Start the installer once it is downloaded and follow the prompts to install Git for Windows.

Installing Docker Desktop

Docker Desktop is a secure, out of the box containerization software offering developers a toolkit for building and running applications. This project leverages containers to provide an

environment-in-a-box for python development. Docker Desktop can be installed using the following steps.

1. Download Docker Desktop for Windows installer from the following link: https://desktop.docker.com/win/main/amd64/Docker%20Desktop%20Installer.exe?_gl=1*s8yam4*_ga*MTMyMjA0NjM4OS4xNjkzMDc1MDc5*_ga_XJWPQMJYHQ*MTY5MzA3NTA3OC4xLjEuMTY5MzA3NTUxMS42MC4wLjA
2. Start the installer once it is downloaded and follow the prompts to install Git for Windows.

Installing Visual Studio Code

Visual Studio Code is a modern source-code editor that offers numerous plugins to enhance developer productivity. Visual Studio Code can be installed using the following steps.

1. Download the VSCode for Windows installer by clicking on the following link: https://code.visualstudio.com/sha/download?build=stable&os=win32-x64-user
2. Open the installer once it is downloaded and follow the prompts to install Visual Studio Code.

Installing Dev Containers

The Dev Containers extensions allows developers to use a container as a full-featured development environment. In the scope of this project, it provides all the necessary dependencies, tooling, and configuration to develop with Python. DevContainers can be added to Visual Studio Code using the following steps.

1. Open the extensions menu in VS Code.
2. Search for Dev Containers
3. Install the Extension the first extension in the list

Cloning Code Repository

The code for this project can be cloned using the following link:
https://github.com/GuassianFlux/CS657-Audio-Deep-Fake-Detector

The git command can be used in the Git Bash for Windows to download the repository. The git command is shown in Figure 1 below.

Terminal Command

```
$ git clone https://github.com/GuassianFlux/CS657-Audio-Deep-Fake-Detector
```

Figure 1, Command for cloning the code repository

Opening Code in Visual Studio Code

Open the project folder in Visual Studio Code and select “Reopen in Container” in the search bar.

Installing Software Dependencies

Technology such as Python, TensorFlow, and other libraries must be installed to compile and run the software. Open the scripts folder in the project repository and run the bash script entitled "install.sh". This script will install all required dependencies that are listed in the requirements text file of the code repository.

Creating a Dataset

A dataset must be downloaded for training, validation, and testing before the software is compiled. The data that was used for this project has also been used in previous research into fake audio detection.

Downloading Data

The data used for this research can be downloaded from the following links.

- Fake Audio: https://zenodo.org/records/5642694 (Over 100,000 generated audio files created using different synthesis algorithms)
- Real Audio: https://keithito.com/LJ-Speech-Dataset/ (Over 10,000 real audio files that were used to create the fake data associated with this project)

The GitHub repo and research paper associated with these datasets can be found at the following links.

- GitHub Repo: https://github.com/RUB-SysSec/WaveFake
- Research Paper: https://arxiv.org/pdf/2111.02813.pdf (Explains synthesis algorithms)

Create a folder for the dataset and add two subfolders named "real" and "fake" for best results. Place the wave files downloaded from the links above in the appropriate folders. A subset of the total data can be used for testing. The fake data downloaded from the source above will also have several subfolders for different synthesis algorithms. They can be placed in the dataset folder with the structure shown in the Figure 2 below. The subfolders will not have any impact on the classes that are created by TensorFlow.

Folder Structure

```
workspace                                                    (code repository)
├── data_sets                        (data folder)
│   ├── fake                        (fake speech)
│   │   ├── generated_audio
│   │   │   ├── common_voices_prompts
│   │   │   │   ├── *.wav
│   │   │   ├── jsut_multi_band_melgan
│   │   │   │   ├── *.wav
│   │   │   ├── jsut_parallel_wavegan
│   │   │   │   ├── *.wav
│   │   │   ├── ljspeech_full_band_melgan
│   │   │   │   ├── *.wav
│   │   │   ├── ljspeech_hifiGAN
```

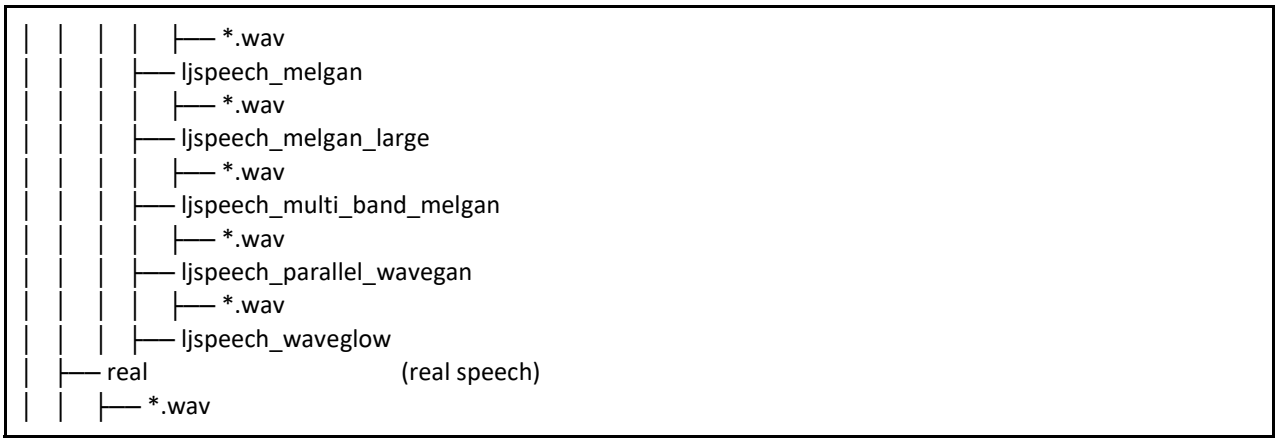

Figure 2, Dataset folder structure

Loading Data with the System

To verify that the new dataset can be loaded, run script1, load_data.py script in the "src" file and pass the path of the dataset folder as an argument to the script. The command is shown in figure 3 below.

Terminal Command

```
$ python ./load_data.py <DATA_SET_DIR>
```

Figure 3, Command for verifying that data can be loaded

The output in the terminal should show the number of files loaded, the number of classes (which should always be 2 - real and fake), and the split for training and validation. The expected output is shown in figure 3 below.

Terminal Output

```
Loading data sets from /workspaces/data_sets
Found 407 files belonging to 2 classes.
Using 326 files for training.
Using 81 files for validation.
```

Figure 4, Terminal output after loading the dataset

Creating the Model

After verifying that the development environment is properly configured and the training data has been downloaded, the next step is fitting the model. This is the process of creating an instance of a model and iteratively teaching it how to classify speech with the fake and real data classes created from the dataset.

Fitting the Model

Fitting is the process of training and adjusting the parameters of the model to improve its prediction accuracy. To begin fitting the model, run the python script entitled script2, fit_model.py, and pass the path of the dataset as an argument to the script. It is also important to specify the path where the keras model will be saved using the -tmp argument. The command is shown in Figure 4 below.

Terminal Command

```
$ python ./fit_model.py <DATA_SET_DIR> -tmp <TRAINED_MODEL_DIR>
```

Figure 5, Command for fitting the model with the dataset

The output in the terminal will show the metrics at each epoch and will also display the path and name of the file where the keras model was saved. The expected output is shown in Figure 5 below.

Terminal Output

```
Loading data sets from /workspaces/data_sets
Found 407 files belonging to 2 classes.
Using 326 files for training.
Using 81 files for validation.
Epoch 1/25
6/6 [==============================] - 9s 1s/step - loss: 5.5127 -
accuracy: 0.6718 - val_loss: 5.2377 - val_accuracy: 0.5625
Epoch 2/25
6/6 [==============================] - 5s 848ms/step - loss: 2.2354 -
accuracy: 0.7117 - val_loss: 2.1009 - val_accuracy: 0.7656
Epoch 3/25
6/6 [==============================] - 5s 858ms/step - loss: 0.7950 -
accuracy: 0.8129 - val_loss: 0.9753 - val_accuracy: 0.7344
Epoch 4/25
6/6 [==============================] - 5s 855ms/step - loss: 0.3061 -
accuracy: 0.8834 - val_loss: 1.0071 - val_accuracy: 0.6719
Epoch 5/25
6/6 [==============================] - 5s 877ms/step - loss: 0.1089 -
accuracy: 0.9571 - val_loss: 0.7420 - val_accuracy: 0.7812
Epoch 6/25
6/6 [==============================] - 6s 898ms/step - loss: 0.0420 -
accuracy: 0.9847 - val_loss: 0.8466 - val_accuracy: 0.7969
Epoch 7/25
6/6 [==============================] - 6s 922ms/step - loss: 0.0384 -
accuracy: 0.9908 - val_loss: 0.7634 - val_accuracy: 0.8256
Epoch 8/25
6/6 [==============================] - 6s 950ms/step - loss: 0.0164 -
accuracy: 1.0000 - val_loss: 0.7019 - val_accuracy: 0.8269
Epoch 9/25
```

```
6/6 [==============================] - 5s 879ms/step - loss: 0.0098 -
accuracy: 1.0000 - val_loss: 0.8120 - val_accuracy: 0.8812
Epoch 10/25
6/6 [==============================] - 6s 959ms/step - loss: 0.0065 -
accuracy: 1.0000 - val_loss: 0.8831 - val_accuracy: 0.7812
Epoch 11/25
6/6 [==============================] - 6s 1s/step - loss: 0.0051 -
accuracy: 1.0000 - val_loss: 0.8719 - val_accuracy: 0.7812
Epoch 11: early stopping
Saving model to path /workspaces/trained_model/model.keras
```

Figure 6, Terminal output after fitting the model

The loss and accuracy metrics are related to data that the model has been trained on. The val_accuracy and val_loss are related to how the model is performing with validation data that it has not been trained on.

Saving the Trained Model

The trained model will automatically be saved in the directory passed to the fit_model.py script. This folder will contain the keras model, a log that shows the metrics of the model after each epoch, an image that shows the plot of the validation accuracy of the model over each epoch, and a plot that shows the validation loss of the model over each epoch. The folder structure is shown below in figure 6.

Folder Structure

```
trained_model                                          (trained model path)
├── history_accuracy.log              (accuracy plot)
├── history_loss.log                  (loss plot)
├── history.log                       (metrics log)
├── model.keras                       (serialized keras model)
```

Figure 7, Folder structure for model and model metric logs

Evaluating Model Predictions

Making predictions involves creating a dataset with wav files that the model has not been trained on and letting the model classify each file.

Making Predictions

The model can be used to make predictions on a test dataset once it has been trained. The test dataset can be created using other sources or can be a subset of the total dataset. The test dataset should not include any files that were used to train the model - this will result in inaccurate results. The accuracy of the model with the test dataset should be comparable to the validation accuracy shown while training the model.

To begin making predictions, run the python script entitled script 3, make_predictions.py and pass the path of the model path and test dataset path as arguments to the script. A path can also be specified for prediction data using the -pdp argument. The prediction data will contain a sample output of each layer in the model.

Terminal Command

```
$ python ./make_predictions.py <DATASET_DIR>
-tmp <TRAINED_MODEL_DIR> -pdp <PREDICTION_DATA_DIR>
```

Figure 8, Command for making predictions with the trained model

The output in the terminal will show information about the loaded model and prediction results. The expected output is shown in figure 8 below.

Terminal Output

```
Loading data sets from /workspaces/data_sets
Found 407 files belonging to 2 classes.
Using 326 files for training.
Using 81 files for validation.
    epoch  accuracy      loss  val_accuracy  val_loss
0       0  0.671779  5.512682      0.562500  5.237662
1       1  0.711656  2.235432      0.765625  2.100936
2       2  0.812883  0.795020      0.734375  0.975301
3       3  0.883436  0.306129      0.671875  1.007065
4       4  0.957055  0.108932      0.781250  0.742034
5       5  0.984663  0.042005      0.796875  0.846626
6       6  0.990798  0.038382      0.765625  0.763375
7       7  1.000000  0.016404      0.796875  0.701945
8       8  1.000000  0.009798      0.781250  0.811953
9       9  1.000000  0.006492      0.821250  0.883100
10     10  1.000000  0.005088      0.881250  0.871945
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 1122, 127, 32)     320

 max_pooling2d (MaxPooling2  (None, 561, 63, 32)       0
 D)

 dropout (Dropout)           (None, 561, 63, 32)       0

 flatten (Flatten)           (None, 1130976)           0

 dense (Dense)               (None, 128)               144765056
```

```
 dense_1 (Dense)             (None, 1)                 129

=================================================================
Total params: 144765505 (552.24 MB)
Trainable params: 144765505 (552.24 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Making dataset predictions...
1/1 [==============================] - 0s 151ms/step
WAV 1 Prediction: real, Actual: real, Confidence: 90.9701%
conv2d_input (1, 1124, 129, 1)
conv2d (1, 1122, 127, 32)
max_pooling2d (1, 561, 63, 32)
dropout (1, 561, 63, 32)
flatten (1, 1130976)
dense (1, 128)
dense_1 (1, 1)
WAV 2 Prediction: fake, Actual: fake, Confidence: 99.7879%
WAV 3 Prediction: fake, Actual: fake, Confidence: 99.9994%
WAV 4 Prediction: fake, Actual: fake, Confidence: 99.2258%
WAV 5 Prediction: real, Actual: real, Confidence: 100.0000%
WAV 6 Prediction: real, Actual: real, Confidence: 99.9963%
WAV 7 Prediction: real, Actual: real, Confidence: 100.0000%
WAV 8 Prediction: real, Actual: real, Confidence: 96.9024%
WAV 9 Prediction: fake, Actual: fake, Confidence: 52.6905%
WAV 10 Prediction: fake, Actual: real, Confidence: 66.7773%
WAV 11 Prediction: fake, Actual: fake, Confidence: 99.8045%
WAV 12 Prediction: fake, Actual: real, Confidence: 56.9043%
WAV 13 Prediction: fake, Actual: fake, Confidence: 99.9636%
WAV 14 Prediction: fake, Actual: fake, Confidence: 92.2934%
WAV 15 Prediction: fake, Actual: fake, Confidence: 94.3780%
WAV 16 Prediction: fake, Actual: fake, Confidence: 99.9397%
WAV 17 Prediction: fake, Actual: fake, Confidence: 99.8213%
Number correct: 13 out of 17
Accuracy: 88.2352%
```

Figure 9, Output of making predictions with a trained model

Prediction data will also be generated in the specified path showing the output of some of the model layers. The folder structure is shown in figure 9 below.

Folder Structure

```
prediction_data                                    (prediction data path)
├── 0_conv2d_input.png                  (model input)
├── 1_conv2d.png                  (convolutional layer)
├── 2_max_pooling2d.png                (max pooling layer)
├── 3_dropout.png                  (dropout layer)
├── 4_flatten.png                  (flatten layer)
```

```
├── 5_dense.png                    (dense layer)
├── 6_dense_1.png                  (dense layer)
├── output.txt                     (all output)
```

Figure 10, Folder structure for prediction data

Interpreting Model Output

The prediction for each of the files in the test dataset will be displayed in the terminal output after making predictions. Each file will have a prediction, actual classification, and confidence percentage. An example of the prediction output is shown in figure 10 below.

`Terminal Output`

```
Making dataset predictions…
1/1 [==============================] - 0s 151ms/step
WAV 1 Prediction: real, Actual: real, Confidence: 90.9701%
conv2d_input (1, 1124, 129, 1)
conv2d (1, 1122, 127, 32)
max_pooling2d (1, 561, 63, 32)
dropout (1, 561, 63, 32)
flatten (1, 1130976)
dense (1, 128)
WAV 2 Prediction: fake, Actual: fake, Confidence: 99.7879%
WAV 3 Prediction: fake, Actual: fake, Confidence: 99.9994%
WAV 4 Prediction: fake, Actual: fake, Confidence: 99.2258%
WAV 5 Prediction: real, Actual: real, Confidence: 100.0000%
WAV 6 Prediction: real, Actual: real, Confidence: 99.9963%
WAV 7 Prediction: real, Actual: real, Confidence: 100.0000%
WAV 8 Prediction: real, Actual: real, Confidence: 96.9024%
WAV 9 Prediction: fake, Actual: fake, Confidence: 52.6905%
WAV 10 Prediction: fake, Actual: real, Confidence: 66.7773%
WAV 11 Prediction: fake, Actual: fake, Confidence: 99.8045%
WAV 12 Prediction: fake, Actual: real, Confidence: 56.9043%
WAV 13 Prediction: fake, Actual: fake, Confidence: 99.9636%
WAV 14 Prediction: fake, Actual: fake, Confidence: 92.2934%
WAV 15 Prediction: fake, Actual: fake, Confidence: 94.3780%
WAV 16 Prediction: fake, Actual: fake, Confidence: 99.9397%
WAV 17 Prediction: fake, Actual: fake, Confidence: 99.8213%
Number correct: 13 out of 17
Accuracy: 88.2352%
```

Figure 11, Model prediction output

Interpreting Confidence Levels

Each prediction will show a confidence level in the output of the model. The raw output is on an interval between 0 and 1, where 0 is the real class and 1 is the fake class. The interpretation of the output is shown in the table below.

Output	Confidence Level
0	100% confident that the speech is real
0.3	70% confident that the speech is real
0.5	Unknown if it is real or fake
0.7	70% confident that the speech is fake
1	100% confident that the speech is fake

Table 2, Prediction confidence levels

Since the confidence levels are calculated as shown above, a prediction should never show a confidence level below 50%. The closer the confidence level is to 50%, the less sure the model is of its prediction. The closer the confidence level is to 100%, the more sure it is in its prediction. Low confidence levels can be an indication that the model is being over or under trained. The amount of training that needs to be done can change based on the size and variation of the dataset, but the system should handle changes in the dataset.

Adding Noise to Audio Files

Model confidence and accuracy can be negatively impacted when noise is introduced into the validation dataset; however, this issue can be mitigated to some degree by adding noisy data to the training data. The following sections show how white noise and burst noise can be added to datasets.

Additive Gaussian White Noise

To add white noise to a set of wav files, run the script entitled script 4, make_noise_dataset.py, and pass input and output directory to the script as arguments. To add white noise, you also must pass the string “white” to the type argument. The signal to noise ratio can also be specified in the arguments.

`Terminal Command`

```
$ python ./make_noise_dataset.py <INPUT_DIR> <OUPUT_DIR> -t white
-r <SIGNAL_TO_NOISE_RATIO>
```

Figure 12, Command for creating a dataset with added white noise

Running the script will not show any output in the terminal if it is successful. The script can be verified by opening the output data path and verifying that the files were generated. Once it is

verified that the data was created, the files in the dataset can be opened and played in Visual Studio Code to verify that the noise has been added.

Burst Noise

To take a dataset and add burst noise to all of the files, run the script entitled script 4, make_noise_dataset.py and pass input and output directory to the script as arguments. To add burst noise, you also must pass the string "burst" to the type argument. The signal to noise ratio can also be specified in the arguments.

Terminal Command

```
$ python ./make_noise_dataset.py <INPUT_DIR> <OUPUT_DIR> -t burst
-r <SIGNAL_TO_NOISE_RATIO>
```

Figure 13, Command for creating a dataset with added burst noise

Running the script will not show any output in the terminal if it is successful. The script can be verified by opening the output data path and verifying that the files were generated. Once the noisy dataset has been generated, the files in the dataset can be opened and played in Visual Studio Code to verify that the noise has been added.

Testing with Noisy Data

Once the model has been trained and a noisy dataset has been created, the first test is to make predictions on the noisy dataset without training the model with the noise added. Based on the findings in this research, it is expected that this will have a lower prediction accuracy. To start making predictions on the noisy dataset, use the same script that was used to make predictions in the sections above, but pass the path to the noise dataset to the script. This can be done with white or burst noise data.

Terminal Command

```
$ python ./make_predictions.py <NOISY_DATASET_DIR>
-tmp <TRAINED_MODEL_DIR> -pdp <PREDICTION_DATA_DIR>
```

Figure 14, Command for making predictions on noisy data

The command line output will look the same as it did when making predictions with the default dataset, but the accuracy is expected to be lower.

Terminal Output

```
Loading data sets from /workspaces/data_sets
Found 407 files belonging to 2 classes.
Using 326 files for training.
Using 81 files for validation.
    epoch  accuracy      loss  val_accuracy  val_loss
```

```
0        0  0.671779  5.512682      0.562500  5.237662
1        1  0.711656  2.235432      0.765625  2.100936
2        2  0.812883  0.795020      0.734375  0.975301
3        3  0.883436  0.306129      0.671875  1.007065
4        4  0.957055  0.108932      0.781250  0.742034
5        5  0.984663  0.042005      0.796875  0.846626
6        6  0.990798  0.038382      0.765625  0.763375
7        7  1.000000  0.016404      0.796875  0.701945
8        8  1.000000  0.009798      0.781250  0.811953
9        9  1.000000  0.006492      0.821250  0.883100
10      10  1.000000  0.005088      0.881250  0.871945
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 1122, 127, 32)     320

 max_pooling2d (MaxPooling2  (None, 561, 63, 32)       0
 D)

 dropout (Dropout)           (None, 561, 63, 32)       0

 flatten (Flatten)           (None, 1130976)           0

 dense (Dense)               (None, 128)               144765056

 dense_1 (Dense)             (None, 1)                 129
=================================================================
Total params: 144765505 (552.24 MB)
Trainable params: 144765505 (552.24 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
Making dataset predictions…
...
...
...
1/1 [==============================] - 0s 151ms/step
WAV 1 Prediction: real, Actual: real, Confidence: 90.9701%
conv2d_input (1, 1124, 129, 1)
conv2d (1, 1122, 127, 32)
max_pooling2d (1, 561, 63, 32)
dropout (1, 561, 63, 32)
flatten (1, 1130976)
dense (1, 128)
dense_1 (1, 1)
WAV 2 Prediction: fake, Actual: fake, Confidence: 99.7879%
WAV 3 Prediction: fake, Actual: fake, Confidence: 99.9994%
WAV 4 Prediction: fake, Actual: fake, Confidence: 99.2258%
WAV 5 Prediction: real, Actual: real, Confidence: 100.0000%
```

```
WAV 6 Prediction: fake, Actual: real, Confidence: 99.9963%
WAV 7 Prediction: real, Actual: real, Confidence: 100.0000%
WAV 8 Prediction: fake, Actual: real, Confidence: 96.9024%
WAV 9 Prediction: fake, Actual: fake, Confidence: 52.6905%
WAV 10 Prediction: real, Actual: real, Confidence: 66.7773%
WAV 11 Prediction: fake, Actual: fake, Confidence: 99.8045%
WAV 12 Prediction: fake, Actual: real, Confidence: 56.9043%
WAV 13 Prediction: fake, Actual: fake, Confidence: 99.9636%
WAV 14 Prediction: fake, Actual: fake, Confidence: 92.2934%
WAV 15 Prediction: fake, Actual: fake, Confidence: 94.3780%
WAV 16 Prediction: fake, Actual: fake, Confidence: 99.9397%
WAV 17 Prediction: fake, Actual: fake, Confidence: 99.8213%
Number correct: 12 out of 17
Accuracy: 70.5882%
```

Figure 15, Output of making predictions on noisy dataset

Training with Noisy Data

Once you have the results of making predictions with noisy data, you can train the model with the noisy data and test again. Use the same script that was used to fit the model with the default dataset but pass the path to the noisy data.

Terminal Command

```
$ python ./fit_model.py <NOISY_DATA_SET_DIR> -tmp <TRAINED_MODEL_DIR>
```

Figure 16, Command for fitting the model with the noisy dataset

The terminal output will look the same as it did when training previously, but the validation accuracy will be slightly worse.

Terminal Output

```
Loading data sets from /workspaces/data_sets
Found 407 files belonging to 2 classes.
Using 326 files for training.
Using 81 files for validation.
Epoch 1/25
6/6 [==============================] - 9s 1s/step - loss: 5.5127 -
accuracy: 0.6718 - val_loss: 5.2377 - val_accuracy: 0.5625
Epoch 2/25
6/6 [==============================] - 5s 848ms/step - loss: 2.2354 -
accuracy: 0.7117 - val_loss: 2.1009 - val_accuracy: 0.7656
Epoch 3/25
6/6 [==============================] - 5s 858ms/step - loss: 0.7950 -
accuracy: 0.8129 - val_loss: 0.9753 - val_accuracy: 0.7344
Epoch 4/25
```

```
6/6 [==============================] - 5s 855ms/step - loss: 0.3061 -
accuracy: 0.8834 - val_loss: 1.0071 - val_accuracy: 0.6719
Epoch 5/25
6/6 [==============================] - 5s 877ms/step - loss: 0.1089 -
accuracy: 0.9571 - val_loss: 0.7420 - val_accuracy: 0.7812
Epoch 6/25
6/6 [==============================] - 6s 898ms/step - loss: 0.0420 -
accuracy: 0.9847 - val_loss: 0.8466 - val_accuracy: 0.7969
Epoch 7/25
6/6 [==============================] - 6s 922ms/step - loss: 0.0384 -
accuracy: 0.9908 - val_loss: 0.7634 - val_accuracy: 0.8256
Epoch 8/25
6/6 [==============================] - 6s 950ms/step - loss: 0.0164 -
accuracy: 1.0000 - val_loss: 0.7019 - val_accuracy: 0.8269
Epoch 9/25
6/6 [==============================] - 5s 879ms/step - loss: 0.0098 -
accuracy: 1.0000 - val_loss: 0.8120 - val_accuracy: 0.8812
Epoch 10/25
6/6 [==============================] - 6s 959ms/step - loss: 0.0065 -
accuracy: 1.0000 - val_loss: 0.8831 - val_accuracy: 0.7812
Epoch 11/25
6/6 [==============================] - 6s 1s/step - loss: 0.0051 -
accuracy: 1.0000 - val_loss: 0.8719 - val_accuracy: 0.7812
Epoch 11: early stopping
Saving model to path /workspaces/trained_model/model.keras
```

Figure 17, Terminal output after fitting the model with noisy dataset

Once the model is trained, making predictions on the noisy dataset with the same script that was used in the last section. The accuracy will improve but won't be better than it was when training and making predictions with the default dataset.

Filtering Noise from Audio Files

To filter noise from all the files in a dataset, run the script entitled script 5, filter_noise_dataset.py and pass input and output directory to the script as arguments.

Terminal Command

```
$ python ./filter_noise_dataset.py <INPUT_DIR> <OUPUT_DIR>
```

Figure 18, Command for creating a dataset with noise filtered

Once the filtered dataset has been generated, the files in the dataset can be opened and played in Visual Studio Code to verify that the noise has been removed.

Testing with Filter Noise Files

Once the noise has been filtered from the wav files, the next test is to repeat the training and predictions from the sections above. When we started this project, we thought that this would be the best way to handle noise in the test dataset; however, this algorithm made the accuracy of the model worse when testing with the default model and testing with a model that had been trained on filtered files. Training and testing with noisy data produced better results.

Producing Data Plots

Waveform and spectrogram data plots can be generated to visualize the changes made to the files when adding and filtering noise during these tests. To create data plots for a specific wav file, run the script entitled script 6, generate_plots.py and pass the path of the file to the script as an argument.

`Terminal Command`

```
$ python ./generate_plots.py <FILE_PATH>
```

Figure 19, Command for generating plots for a wav file

The plot of the waveform, spectrogram, and dual wave spectrogram will be saved to file in the working directory.

Waveform

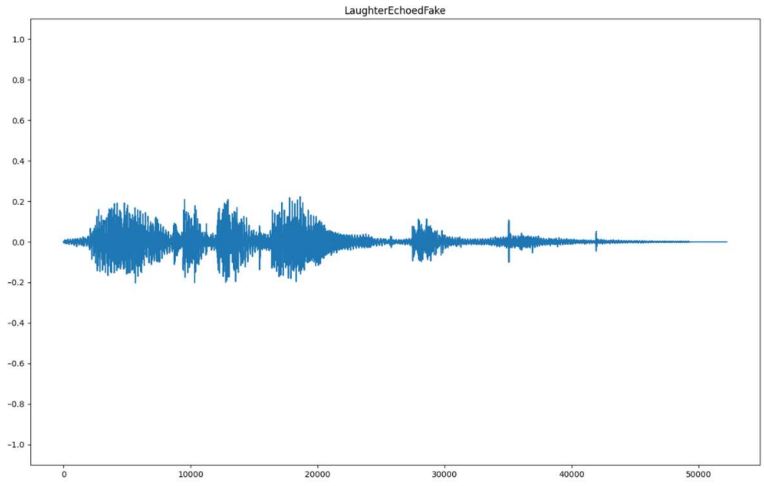

Figure 20, Waveform Plot

Spectrogram

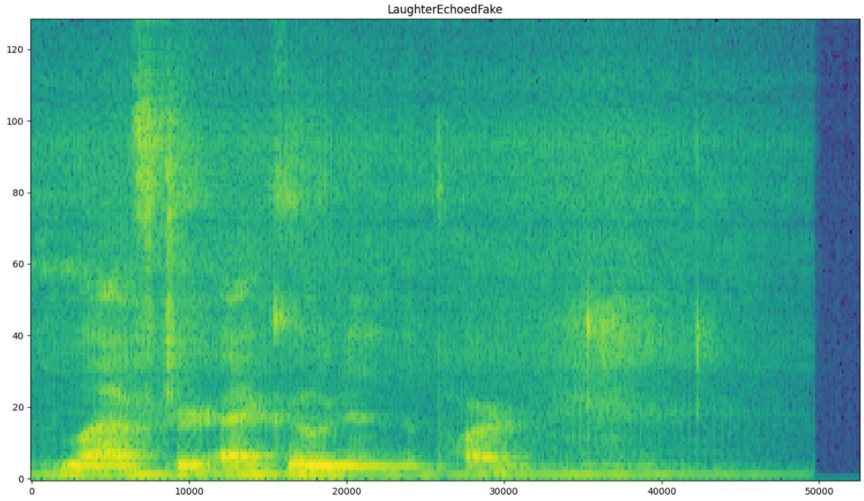

Figure 21, Spectrogram Plot

Dual Waveform Spectrogram

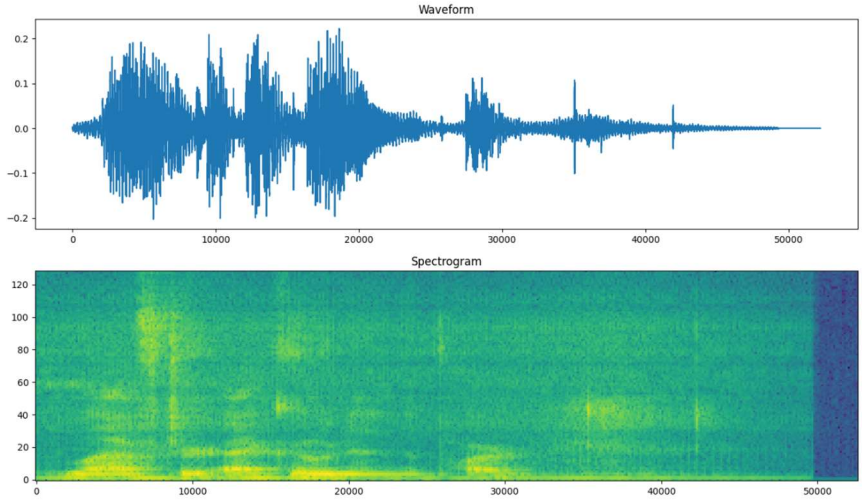

Figure 22, Dual Waveform Spectrogram Plot

Results and Conclusion

When evaluating the performance of the detection model, four groupings of data were constructed: one for each generative algorithm and one that combined all of the generative algorithms. For each grouping, several scenarios were constructed:

1. A baseline trained on nominal data and predicted against nominal data
2. A model trained on nominal data and predicted against white noise data
3. A model trained on white noise data and predicted against white noise data
4. A model trained on nominal data and predicted against filtered white noise data
5. A model trained on nominal data and predicted against burst noise data
6. A model trained on burst noise data and predicted against burst noise data
7. A model trained on nominal data and predicted against filtered burst noise data

These scenarios were chosen to determine the impact of different noise models on the prediction accuracy of the project's detection model and how it may be mitigated. Each scenario was executed for 40 runs, where the average prediction accuracy and equivalent error rate are demonstrated in Table 3.

When viewing the results from Table 3, a few observations may be made. When comparing the baseline performance for each algorithm, the model performs best against Apple's Personal Voice, followed by PlayHT, and then by FastSpeech2. When comparing noise, burst noise appears to impact the prediction accuracy more severely than white noise for an equivalent SNR. This may be mitigated by introducing the noise into the training set, and in general, filtering the noise before predictions reduces accuracy. When comparing the combined group to its individual counterparts, the baseline prediction accuracy is nearly equivalent to the averaged individual baselines (84.4345%); however, the impact of noise is not as extreme, such that the accuracy deltas are on average lower in magnitude. Though this may not be true with additional test cycles, the combined group performed best when trained with white noise and predicted against white noise.

When comparing the results of this experiment to DeepSonar, an alternative detection method, the combined baseline results for this project's prediction model are 13.1385% lower; however, DeepSonar was very susceptible to noise, such that prediction accuracy would be as low as 73% when it was present. While this project's nominal prediction accuracy is lower than DeepSonar's, its noisy prediction accuracy is nearly 11% greater on average, and the worst accuracy loss when noise is present is nearly 2.5% rather than DeepSonar's 25%. Under ideal conditions DeepSonar certainly has the ability to perform better, but this project's detection model performs most consistently.

Scenario	Avg Prediction Acc.	EER	Accuracy Delta
FastSpeech2			
Baseline	75.0000	11.3281	0.0000
White Noise - No Training	73.1818	12.2656	-1.8182
White Noise - Training	75.6818	10.9766	0.6818
White Noise - Filtered	68.7879	17.1094	-6.2121
Burst Noise - No Training	63.7879	17.1094	-11.2121
Burst Noise - Training	70.2273	13.7891	-4.7727
Burst Noise - Filtered	66.2879	15.8203	-8.7121
iOS Personal Voice			
Baseline	96.4286	0.0000	0.0000
White Noise - No Training	92.9464	1.8056	-3.4822
White Noise - Training	96.4286	0.0000	0.0000
White Noise - Filtered	96.0715	0.1852	-0.3571
Burst Noise - No Training	80.9822	8.0093	-15.4464
Burst Noise - Training	96.4286	0.0000	0.0000
Burst Noise - Filtered	94.5536	0.9722	-1.8750
PlayHT			
Baseline	81.8750	9.0605	0.0000
White Noise - No Training	76.2480	11.8745	-5.6270
White Noise - Training	78.1250	10.9375	-3.7500
White Noise - Filtered	69.3750	15.3115	-12.5000
Burst Noise - No Training	71.8750	14.0625	-10.0000
Burst Noise - Training	75.6250	12.1875	-6.2500
Burst Noise - Filtered	71.8500	14.0625	-10.0250
Combined (FastSpeech2, Personal Voice, PlayHT)			
Baseline	84.9615	6.8555	0.0000
White Noise - No Training	83.9231	7.3828	-1.0384
White Noise - Training	85.5000	6.5820	0.5385
White Noise - Filtered	81.2308	8.7500	-3.7307
Burst Noise - No Training	82.3846	8.1641	-2.5769
Burst Noise - Training	80.5000	9.1211	-4.4615
Burst Noise - Filtered	78.8846	9.9414	-6.0769

Table 3, Results tables

Reviewing the objectives set for the project, the team was able to successfully achieve the initial plan. The combined group detection accuracy of 84.9615% and EER of 9.0605 exceed the desired minimum threshold of 80% detection accuracy. Additionally, the model was fairly resilient to noise interference; when compared to a modern counterpart, the performance loss was reduced by a factor of 10. And last, these results were collected against modern, publicly available generative algorithms providing contemporary data points for audio deep fakes.

Future Work

However successful in the scope of the semester, the project is far from solving the issues presented by audio deep fakes. If given more time, objectives for improving the detection accuracy, adding the ability to classify different synthesis architectures, and increasing the productization of the model would outline the next steps for the project moving forward. Furthermore, There are still major major issues related to maintaining a high detection accuracy when noise is added to the test data. Training with a larger variation of data and introducing noise to the validation dataset improves the accuracy to some degree, but there is still a significant decrease from the baseline.

Scripts

Script 1, load_data.py

```
##########################################
# Put this at the top of each file before
# importing TensorFlow to suppress warnings
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
##########################################

from data_processor.data_processor import Data_Processor
import tensorflow as tf
import warnings
import argparse

if __name__ == '__main__':
    # Ignore Python and TensorFlow Warning and Debug Info
    warnings.filterwarnings("ignore")
    tf.get_logger().setLevel('ERROR')

    # Arguments and help tips
    parser = argparse.ArgumentParser(description='Script for loading
a dataset')
    parser.add_argument('dataset_path', help='Path to the dataset.
The dataset directory should contain a folder named fake and a folder
name real.')
    args = parser.parse_args()
```

```
    # Dataset path
    dataset_path = args.dataset_path

    # Create initialize data processor and load datasets
    processor = Data_Processor()
    processor.load_datasets(dataset_path)
    processor.make_spectrogram_datasets()
```

Script 2, fit_model.py

```
##############################################
# Put this at the top of each file before
# importing TensorFlow to suppress warnings
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
##############################################

from data_processor.data_processor import Data_Processor
from deepfake_detector.deepfake_detector import DeepFake_Detector
import tensorflow as tf
import warnings
import argparse

if __name__ == '__main__':
    # Ignore Python and TensorFlow Warning and Debug Info
    warnings.filterwarnings("ignore")
    tf.get_logger().setLevel('ERROR')

    # Arguments and help tips
    parser = argparse.ArgumentParser(description='Script for fitting
the model')
    parser.add_argument('dataset_path', help='Path to the dataset.
The dataset directory should contain a folder name fake and a folder
name real.')
    parser.add_argument('--trained_model_path', '-tmp', help='Path
where the trained model and metrics will be saved.')
    args = parser.parse_args()

    # Dataset path
    dataset_path = args.dataset_path

    # Trained model path
    trained_model_path = args.trained_model_path

    # Create initialize data processor and load datasets
    processor = Data_Processor()
    processor.load_datasets(dataset_path)
    processor.make_spectrogram_datasets()
```

```
    # Trains a new model with the loaded datasets.
    processor.fit_model(trained_model_path)
```

Script 3, make_predictions.py

```
##############################################
# Put this at the top of each file before
# importing TensorFlow to suppress warnings
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
##############################################

from data_processor.data_processor import Data_Processor
from deepfake_detector.deepfake_detector import DeepFake_Detector
import tensorflow as tf
import warnings
import argparse

if __name__ == '__main__':
    # Ignore Python and TensorFlow Warning and Debug Info
    warnings.filterwarnings("ignore")
    tf.get_logger().setLevel('ERROR')

    # Arguments and help tips
    parser = argparse.ArgumentParser(description='Script for making
predictions with a trained model')
    parser.add_argument('dataset_path', help='Path to the dataset.
The dataset directory should contain a folder name fake and a folder
name real.')
    parser.add_argument('--trained_model_path', '-tmp', help='Path
where the trained model and metrics will be saved.')
    parser.add_argument('--prediction_data_path', '-pdp', help='Path
where prediction data, heatmaps, and activation maps will be saved.')
    args = parser.parse_args()

    # Dataset path
    dataset_path = args.dataset_path

    # Trained model path
    trained_model_path = args.trained_model_path

    #Prediction data path
    prediction_data_path = args.prediction_data_path

    # Create initialize data processor and load datasets
    processor = Data_Processor()
    processor.load_datasets(dataset_path)
```

```
    processor.make_spectrogram_datasets()

    # Load the trained model and make predictions on the
    # on the test dataset
    detector = DeepFake_Detector()
    detector.load_model_from_file(trained_model_path)
    test_ds = processor.get_test_dataset()
    class_names = processor.get_class_names()
    detector.predict_dataset(test_ds, class_names,
prediction_data_path)
```

Script 4, make_noise_dataset.py

```
import sys, getopt
import os
from scipy.io import wavfile
import numpy as np
from noise_generator.noise_generator import Noise_Generator
import argparse
import shutil

# Add function reference to noise option here.
# Main will check against this data structure to determine if valid
options are passed
noise_dict = {
     'white': Noise_Generator.add_agwn,
     'burst': Noise_Generator.add_burst
}

def main(argv):
    # snr = 15
    # type = 'white'
    # opts, args = getopt.getopt(argv,'hr:t:',['snr=,type='])
    # for opt, arg in opts:
    #     if opt == '-h':
    #         print('make_noise_dataset.py <input_dir>
<output_dir>\n-r, --snr   Signal to Noise ratio')
    #         sys.exit()
    #     elif opt in ('-r', '--snr'):
    #             snr=np.int16(arg)
    #     elif opt in ('-t', '--type'):
    #         if arg in noise_dict.keys():
    #             type = arg
    #         else:
    #             print('Invalid noise option.')
    #             print('Valid options are:')
    #             print('[' + ", ".join(list(noise_dict.keys())) +
']')
```

```
    #             print('Defaulting to white noise.')

    # Arguments and help tips
    parser = argparse.ArgumentParser(description='Script for making
white noise or burst noise dataset')
    parser.add_argument('input_dir', help='Path to the input
directory')
    parser.add_argument('output_dir', help='Path where the trained
model and metrics will be saved.')
    parser.add_argument('--type', '-t', help='Noise type - White
noise or bust noise')
    parser.add_argument('--snr', '-r', help='Sinal to noise ratio')
    args = parser.parse_args()

    input_dir = args.input_dir
    output_dir = args.output_dir
    type = args.type
    snr = args.snr

    if os.path.exists(output_dir):
        try:
            shutil.rmtree(output_dir)
        except OSError as e:
            print(e)
    os.makedirs(output_dir)

    for filename in os.listdir(input_dir):
        inputfile = os.path.join(input_dir,filename)
        outputfile = os.path.join(output_dir,filename)
        if os.path.isfile(inputfile):
            samplerate, wav = wavfile.read(inputfile)
            wav = noise_dict[type](wav, snr)
            wavfile.write(outputfile,samplerate,wav)

if __name__ == '__main__':
    main(sys.argv[1:])
```

Script 5, filter_noise_dataset.py

```
import sys, getopt
import os
from scipy.io import wavfile
import argparse
from noise_filter.noise_filter import Noise_Filter

def main(argv):
    # opts, args = getopt.getopt(argv,"h")
    # for opt, arg in opts:
```

```
    #     if opt == '-h':
    #         print('make_noise_dataset.py <input_dir>
<output_dir>\n-r, --snr   Signal to Noise ratio')
    #         sys.exit()


    # Arguments and help tips
    parser = argparse.ArgumentParser(description='Script for making
white noise or burst noise dataset')
    parser.add_argument('input_dir', help='Path to the input
directory')
    parser.add_argument('output_dir', help='Path where the trained
model and metrics will be saved.')
    args = parser.parse_args()

    input_dir = args.input_dir
    output_dir = args.output_dir

    for filename in os.listdir(input_dir):
        inputfile = os.path.join(input_dir,filename)
        outputfile = os.path.join(output_dir,filename)
        if os.path.isfile(inputfile):
            samplerate, wav = wavfile.read(inputfile)
            wav = Noise_Filter.filter(wav,samplerate)
            wavfile.write(outputfile,samplerate,wav)

if __name__ == "__main__":
    main(sys.argv[1:])
```

Script 6, generate_plots.py

```
import sys
from scipy.io import wavfile
from plotter.plotter import Plotter

def get_label(filepath):
    filename = filepath.split('/')[-1]
    return filename.split('.')[0]

def main(argv):
    input_filepath = argv[-1]
    label = get_label(input_filepath)

    samplerate, wav = wavfile.read(input_filepath)

    Plotter.plot_waveform(waveform=wav, label=label)
    Plotter.plot_spectrogram(waveform=wav, label=label)
    Plotter.plot_dual_wave_spec(waveform=wav, label=label)
```

```
if __name__ == "__main__":
    main(sys.argv[1:])
```

Project Repository

The source code for this project is public on GitHub and can be found at the following location:
https://github.com/GuassianFlux/CS657-Audio-Deep-Fake-Detector

Repository Contents

The project repository contains the following contents:

- Instructions for setting up the development environment
- DevContainer configuration files
- Scripts for installing all dependencies
- Source code for loading datasets, fitting the model, visualizing data, and making predictions
- High level workflow scripts
- Documentation:
    - Design Manual: Outlines the software design and project architecture
    - User Manual: Explains and demonstrates how the software is used
    - Final Qualification Test: Explains how to setup the development environment and reproduce results
    - Research Paper: Summaries the project from beginning to end and presents the final results

How to Contribute

Use the following command to clone the code repository.

```
$ git clone https://github.com/GuassianFlux/CS657-Audio-Deep-Fake-Detector
```