Caterpillar: Machine Learning Algorithm for Weld Acoustics Monitoring

Diana Pham, Nate Williamson, Kevin Mackey,

Oscar Nelsen, Rohit Patil, Holly Roach, Soham Manjrekar

TMGT 461

Spring Semester 2024

CATERPILLAR WELD ACOUSTICS MONITORING PROGRAM

Project Team: Nate Williamson

Soham Manjrekar

Rohit Patil

Holly Roach

Diana Pham

Kevin Mackey

Oscar Nelsen

Project Advisor: Professor Molly Hathaway Goldstein

Company Sponsor: Caterpillar, Inc.

Peoria, Illinois 61629

Company Advisors: Mr. William Barnickel

Mr. Tazio Grivetti

TMGT 461

Spring Semester 2024

The Hoeft Technology and Management Program

University of Illinois at Urbana-Champaign

Abstract/Executive Summary:

Throughout the development of this project, our team collaborated closely with Caterpillar's welding experts to understand the intricacies of the welding process and the specific challenges associated with weld quality assessment. By leveraging microphones designed for a welding environment and machine learning (ML) algorithms, we developed a robust feedback system capable of real-time weld quality evaluation.

The project commenced with extensive research into welding acoustics and existing ML applications in industrial settings. This foundational understanding enabled us to design a comprehensive approach for data collection, analysis, and feedback generation. We established clear objectives and deliverables, emphasizing the importance of creating a user-friendly solution adaptable to Caterpillar's diverse manufacturing environment.

Over the course of the semester, our team made significant strides in implementing the proposed solution. We received audio samples collected from Caterpillar's welding laboratory and refined our ML model to accurately classify weld quality based on acoustic signals. Through iterative testing and optimization, we achieved promising results, demonstrating the feasibility and potential impact of our approach.

The economic analysis revealed compelling cost-saving opportunities for Caterpillar if they decide to implement the ML-driven weld quality monitoring system. By reducing rework, the implementation of our solution has the potential to generate substantial returns on investment while enhancing overall operational efficiency.

In conclusion, the Caterpillar Machine Learning Algorithm for Weld Acoustics Monitoring project represents a significant advancement in weld quality assessment methodologies. Our findings underscore the transformative potential of machine learning in industrial settings, paving the way for enhanced productivity, cost savings, and product quality. Moving forward, we envision continued collaboration between academia and industry to drive innovation and address complex challenges in manufacturing processes.

Key Words:

Welding, Acoustics, Sound Monitoring, Time Domain, Frequency Domain, Machine Learning, Caterpillar, Microphone, Algorithm, Artificial Intelligence, Testing Automation.

Acknowledgements:

The University of Illinois team extends its sincere gratitude to the Hoeft Technology and Management Program for providing us with the invaluable opportunity to undertake this senior capstone project. We are grateful for the guidance and mentorship provided by our esteemed faculty advisor, Professor Molly Goldstein, as well as the invaluable support from course directors Professor Harrison Kim and Professor Arkadiy Sakhartov.

We would like to express our appreciation to our sponsor, Caterpillar, for their unwavering support throughout this project. We extend our thanks to William Barnickel, Tazio Grivetti, and Mark Manzi for their dedicated time, guidance, and insights into weld operations and the economic aspects of our endeavors. Your willingness to share your expertise has greatly enriched our project experience and facilitated our learning.

Table of Figures

List of Tables

Table of Contents

Introduction:

This report provides a summary of the outcomes from the current semester's Caterpillar Capstone project. The primary goal of the project is to use machine learning to construct a monitoring system that can provide real-time feedback for weld quality. Welding defects such as porosity, contamination, inconsistent wire feed, shield gas flow/mix, and burn-through can significantly affect the quality of welds, often requiring costly and inefficient rework. The machine learning algorithm was built to detect these common welding defects. This report is structured into several key sections, including our problem statement, objectives, and deliverables.

Problem Statement/Opportunities Definition:

Welding is a critical aspect of Caterpillar manufacturing. One of the challenges in automating this process is monitoring the quality of the welds. The current methods for assessing weld quality lack efficiency, relying on manual inspection or rudimentary techniques. In welding acoustics, where sound carries valuable information about welding integrity, machine learning algorithms could add great value. An algorithm must analyze weld acoustics data in real-time, accurately identifying defects while filtering out background noise. To train an algorithm like this, we estimate needing at least 10 times as many data points as there are classifications[4]. For example, if we have 4 different types of welds we would like to identify, we will want at least 40 samples per type of weld, with an ideal of 80. If implemented successfully, the algorithm should be adaptable across different welding processes and materials, providing insights into defect causes for process optimization and quality control. Solving this challenge will streamline inspection, enhancing productivity and safety in industrial welding.

With Caterpillar owning 48 facilities globally and approximately 147 welding stations at each of their manufacturing sites, there are 7000+ welding stations across Caterpillar. Welding by hand is a labor-intensive and time-consuming process that takes welders years to master. While most welds are of high quality, there can be some abnormalities in the welding process. There is a major opportunity for creating highly efficient weld feedback systems that provide welders with real-time feedback to enhance productivity.

Another opportunity for Caterpillar within this project is to obtain technical expertise in microphone and audio monitoring. Audio verification can be applied to manufacturing, material validation, construction, and other existing monitoring systems. The machine learning algorithm from this project can also be applied to predict more accurately how and when errors might occur in the larger manufacturing and assembly process.

Objectives and Deliverables:

Our team identified multiple objectives to accomplish throughout this project. We have a core objective to learn and research the topic of welding acoustics and engineering principles related to testing. Another main objective is to facilitate an efficient and practical weld acoustics ML model to Caterpillar by creating an optimal weld acoustics data collection system. To do this, we transformed the testing files to the optimal size and format for data analysis. This algorithm can accurately analyze welding audio to identify welding errors in real time and should be easy for Caterpillar to use and implement across many worksites. It should also be able to run with few errors and be considered a "blackbox" (Figure 1).

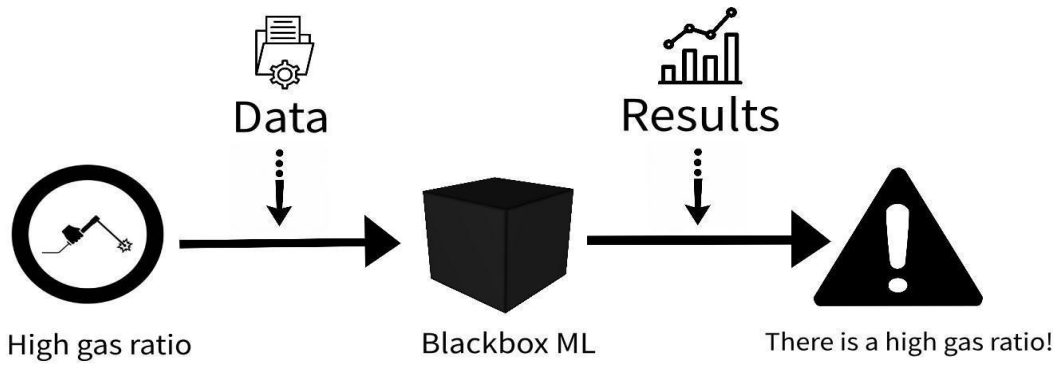

FIGURE 1: EXAMPLE OF MACHINE LEARNING BLACKBOX APPLICATION

Our final main objective of this project is to predict and classify bad welds in real time. We plan for our ML algorithm to give feedback so welders will know in real time whether their welds are acceptable or not. This is important since this will save welders time and effort. To do this, we developed an interface to convey algorithm responses. Our ML algorithm analyzes the welds and gives feedback in real time. We have created instructions for CAT to continue using and refining the algorithm to their needs.

Background:

Article Research -

According to the article titled "Acoustic signal-based automated control of welding penetration using digital twin technology[5]" written by J. Tao, N.M. Nor, and A.B.B. Abdullah, there is much that goes into the research and experimentation of finding the most optimal weld quality and categorizing weld quality.

Outline -

Their team determined that welding penetration control is a critical aspect to ensure the quality of robotic welding. This relates to our project because when classifying different types of welds, we need to know what portions of the welding process are most important to create a successful weld. The article also outlines a major challenge that they faced, which is retrieving reliable data from a complex welding environment in "a noncontact and non-destructive manner as well as achieving real-time computing." We too have faced similar issues in our project. There is a large amount of feedback noise and external environment audio.

Solutions and Takeaways-

In the article, they employed various sensors to monitor the welding penetration. We are currently implementing the use of dish microphones to focus on the audio of each welding booth. Placing ultrasonic audio sensors to measure the welding penetration "by employing the direct reflection of longitudinal or shear waves from the bottom of the weld," just as they did in their experiment, is another strategy that can be utilized.

Solution Methods:

Initial Framework-

To begin development of the ML model, we needed to consider several crucial factors. Given the complexity of weld quality assessment and the need for real-time feedback in industrial settings, our approach required careful planning and consideration.

Selecting an appropriate coding language was of prime importance for the project's success. Prompted by the recommendations of the CAT team and informed by industry standards, we opted to utilize Python for our ML model development. Python's versatility, extensive library support, and popularity in the field of ML made it an ideal choice for our project.

Additionally, building a robust dataset was essential for training our ML algorithm effectively. To achieve this, we collaborated closely with the CAT team to gather real-world audio samples representing different weld defect categories, including porosity, contamination, inconsistent wire feed, shield gas flow/mix issues, and burn-through. Our goal was to collect sufficient audio samples to ensure the algorithm's accuracy and reliability in identifying various weld defects.

After obtaining a sufficient initial dataset, we planned to leverage simple ML techniques and frameworks to develop our algorithm. Initially, we aimed to use Deep Audio Classification, an open-source Jupyter notebook specifically designed for classifying audio samples in .wav format. Additionally, we intended to implement a digital Noise Reduction Program to enhance the quality of our audio data, thereby improving the performance of our ML model. This approach was initially favored for its accessibility and cost-effectiveness. Jupyter notebooks are user-friendly, easily transferable, and require minimal coding experience to navigate. However, they are not very flexible and have limited extensibility.

The lessons learned from this approach led us to pivot due to several factors. Firstly, while Jupyter notebooks offered ease of use, their limitations in handling large datasets and scalability became apparent as our project progressed. Additionally, the complexity of our ML model's requirements demanded more advanced features and customization than Jupyter notebooks could provide. Consequently, we transitioned to a more scalable and customizable development environment to meet our evolving needs.

Current Framework-

All relevant files/folders mentioned below can be found in “finalCAT” folder in the SharePoint

Program setup

If downloading 'finalCAT' directly from sharepoint, you can skip Step 3 and Step 4, since the audio samples are already spliced. If adding new audio files, revisit Step 3 and Step 4.

Step 1: Install python3

On Macintosh, we had to create a virtual environment to be able to run a python script. We did this by running the following command in terminal:

```
python3 -m venv myenv
```

Then to activate the virtual environment (which we did every time we started a new session), we ran:

```
source myenv/bin/activate
```

Step 2: Install required libraries

Below are the required libraries:

- Numpy
- Librosa
- Soundfile
- Os
- Matplotlib
- Sklearn
- Watchdog
- Queue
- Time

To install any new libraries, run the following command in terminal:

```
pip3 install library_name
```

Step 3: Prepare to build Main classifier: Splice and organize audio samples

Create a folder (named whatever you like) that will act as the main directory that hosts this program. Place splice_audio.py and final_main.py in this main directory. Also within this main directory, create a folder named "samples". Within "samples", create 4 folders named the following: "acceptable", "acceptable_old", "bad", "bad_old".

The code assumes 10 second segments of audio, therefore each sample it ingests should be 10 seconds long. splice_audio.py will take any length audio sample and splice it into 10 second segments. splice_audio.py will splice every audio sample in folders "acceptable_old" and "bad_old" and deposit all 10 second segments into folders "acceptable" and "bad", respectively. To run the script, enter the following common in terminal:

```
python3 splice_audio.py
```

Splicing only needs to happen once per raw audio file; we only ran splice_audio.py whenever we downloaded new raw audio files. The segments will be named according to the following convention, where x is an integer: [original file name]_segment_x.

You should now have two folders named “acceptable” and “bad”, each filled with 10-second audio samples of the respective weld quality; there should only be 10 second segments of bad welds in the “bad” folder, and there should only be 10 second segments of acceptable welds in the “acceptable” folder.

Create a new folder called “new samples”. “new samples” will be the folder that the main program watches for new audio files. Whenever a new audio file is placed in this folder, it will be classified. For now, “new samples” should be empty.

Summary: The main directory should host splice_audio.py, final_main.py, “samples”, and “new samples”. Before running the program, the “samples” directory should host two folders, “acceptable” and “bad”, which contain 10 second segments of the relevant quality audio sample.

Step 4: Prepare to build bad-type classifier

Note: the bad-type classifier follows similar folder structure as the main classifier.

Within the main directory, create a folder called “bad_classifier”. Within “bad_classifier”, create a folder named “samples”. Within “samples”, create folders for each category of bad weld. Currently, the code supports the following categories: uncategorized, bad angle, unclean, and undercut. So, to begin, create four folders within “samples” named “uncategorized”, “bad angle”, “unclean”, and “undercut”. Note: In the Category Modifications section below, we explain how to change or expand these categories (for example, to add a ‘porosity’ category). Like the main classifier, fill each category’s folder with 10 second samples of the respective type of weld. So, for example, there would only be 10 second audio files of “undercut” welds inside the samples/undercut folder.

Recommendation: Copy over audios from the main classifier’s *samples/bad* folder into the appropriate folder for the type of weld. Splice_audio.py preserved the raw audio file name in the file name for each 10 second segment, so if the original file name contained information about the type of bad weld, this should be simple. Note: Since the main classifier and bad-type classifier are separate classifiers, it is okay if they have training samples in common.

Summary: The main directory should also host a folder named “bad_classifier”. Within “bad_classifier”, there should be a folder named “samples”. Within “samples” there should be four folders named “uncategorized”, “bad angle”, “unclean”, and “undercut”, each filled with 10 second audio samples of their respective type of weld.

Step 5: Run the program

If on Mac, make sure the virtual environment is activated (see Step 1).

In terminal, navigate to the main directory where the whole program sits. Then, run the following command in terminal:

```
python3 final_main.py
```

Step 6: Classify new audio

Note: Mac users have a hidden .DS_Store file in every folder they create. The code is written to ignore/skip such files, which is why you see the "Hidden file. Skipping …" notification in terminal below.

When the classifiers are finished building/training, you will see the following output in terminal (Figure 2). Notice the output "Monitoring directory: 'new samples'". This means you can now place new 10-second audio files to be classified in 'new samples'.

```
(myenv) hollyroach@Hollys-MacBook-Air-2 finalCAT % python3 final_main.py
Hidden file. Skipping: samples/acceptable/.DS_Store
Hidden file. Skipping: samples/bad/.DS_Store
Classifier training finished.
Hidden file. Skipping: bad_classifier/samples/uncategorized/.DS_Store
Hidden file. Skipping: bad_classifier/samples/bad angle/.DS_Store
Hidden file. Skipping: bad_classifier/samples/unclean/.DS_Store
Hidden file. Skipping: bad_classifier/samples/undercut/.DS_Store

 Bad Classifier training finished.
Monitoring directory: 'new samples'
```

FIGURE 2: OUTPUT WHEN CLASSIFIER TRAINING FINISHED

Upon opening a new audio sample inside 'new samples', a text notification of the classification will appear in terminal, and a python window will appear with a red/green text box depending on the classification. For example, in Figure 3 below, a sample classified as 'acceptable' will display:

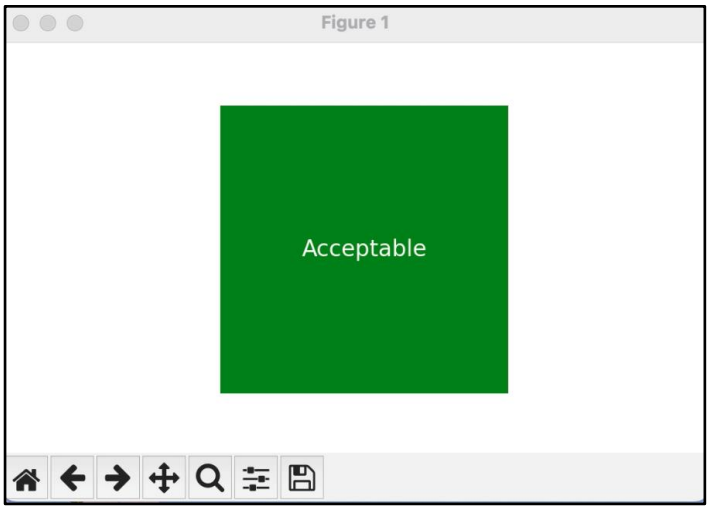

FIGURE 3: HMI OUTPUT OF AN 'ACCEPTABLE' CLASSIFICATION

And a sample classified as 'bad angle' will display:

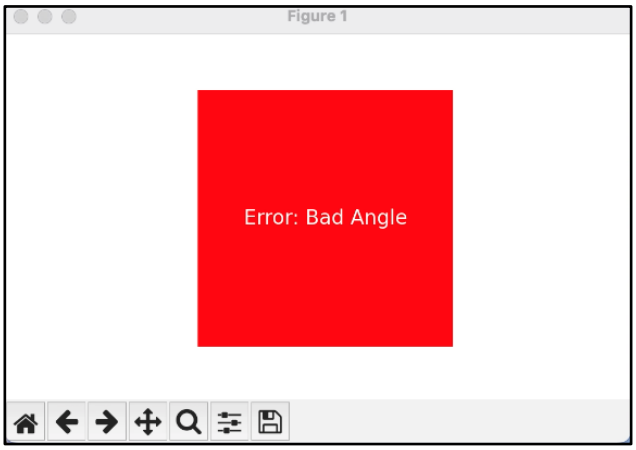

FIGURE 4: HMI OUTPUT OF A 'BAD ANGLE' CLASSIFICATION

To quit the program, hit Ctrl+C.

Category Modifications

First, reorganize the folder structure within your 'main directory'/bad classifier/samples folder. There are currently 4 folders named "uncategorized", "bad angle", "unclean", and "undercut". To modify these categories of the bad-type classifier, create folders with the name of each desired category. Make sure to fill each folder with 10 second audio samples of the respective type of weld.

For example, if you wanted to add a "porosity" category to the current categories, there would now be 5 folders within 'main directory'/bad classifier/samples, named the following: "uncategorized", "bad angle", "unclean", "undercut", AND "porosity".

To update the script itself, make the following edits to final_main.py:

1) In the update_plot() function (Figure 5), which updates the red/green box in the python plot window, update the text notification code with the appropriate category name. As seen below on line 25, if the classification was “bad” and “uncategorized” the text would be set to “Error: Uncategorized”. Alter/Add these if-statements as necessary.

```
20  # Function to update the plot
21  def update_plot(pred_label_name, bad_classification):
22      # Determine color based on prediction
23      if pred_label_name == "bad":
24          color = 'r'
25          if bad_classification == "uncategorized":
26              text = 'Error: Uncategorized'
27          if bad_classification == "bad angle":
28              text = 'Error: Bad Angle'
29          if bad_classification == "unclean":
30              text = 'Error: Unclean'
31          if bad_classification == "undercut":
32              text = 'Error: Undercut'
33      elif pred_label_name == "acceptable":
34          color = 'g'
35          text = 'Acceptable'
```

FIGURE 5: UPDATE_PLOT() SNIPPET

2) In the badtype_train() function, add/alter the categories seen on line 146 below.

```
143 # Train function
144 def badtype_train():
145     base_path_train = "bad_classifier/samples"
146     categories = ["uncategorized", "bad angle", "unclean", "undercut"]
```

FIGURE 6: BADTYPE_TRAIN() SNIPPET

3) In the bad_classify(sample) function, add/alter the categories seen on line 189 below.

```
183 def bad_classify(sample):
184     global running
185     running = True
186
187
188     # Define categories
189     bad_categories = ["uncategorized", "bad angle", "unclean", "undercut"]
```

FIGURE 7: BAD_CLASSIFY(SAMPLE) SNIPPET

How We Built and Tested the Program

After struggling to implement open-source code, our team utilized ChatGPT to build a first draft and to aid debugging.

The real-time implementation of the classifier (where a user can place new raw samples into the "new samples" folder, and it will be classified in real-time) cannot track accuracy because there is currently no feedback feature to correct a mistake. However, we were able to test our classifiers (both the main classifier and bad-type classifier) using different scripts which build the classifiers in an identical manner, test them with a set-aside testing set, and track accuracy.

audio_classifier.py, if placed in the main directory, will build and train the main classifier (identically to the final_main.py script), independently setting aside some samples to test the classifier with. Once the classifier is trained, it will go through each sample in the testing set and record its accuracy. The following terminal output is an example of an audio_classifier.py run (Figure 8).

```
(myenv) hollyroach@Hollys-MacBook-Air-2 finalCAT % python3 audio_classifier.py
Hidden file. Skipping: samples/acceptable/.DS_Store
Hidden file. Skipping: samples/bad/.DS_Store

Predictions:
Sample 1: Predicted Label: acceptable, True Label: bad
Sample 2: Predicted Label: acceptable, True Label: bad
Sample 3: Predicted Label: acceptable, True Label: bad
Sample 4: Predicted Label: acceptable, True Label: acceptable
Sample 5: Predicted Label: acceptable, True Label: acceptable
Sample 6: Predicted Label: acceptable, True Label: acceptable
Sample 7: Predicted Label: acceptable, True Label: acceptable
Sample 8: Predicted Label: bad, True Label: bad
Sample 9: Predicted Label: acceptable, True Label: acceptable
Sample 10: Predicted Label: bad, True Label: bad
Sample 11: Predicted Label: acceptable, True Label: acceptable
Sample 12: Predicted Label: acceptable, True Label: acceptable

Accuracy: 75.0%
```

FIGURE 8: TERMINAL OUTPUT OF AUDIO_CLASSIFIER.PY

**Note: accuracy was 75% with this testing set but may vary with other testing sets.*

Orig_bad_audio_classifier.py will do the same if placed in the 'main directory'/bad_classifier folder. The following terminal output is an example of an audio_classifier.py run.

```
(myenv) hollyroach@Hollys-MacBook-Air-2 bad_classifier % python3 orig_bad_audio_
classifier.py
Hidden file. Skipping: samples/bad angle/.DS_Store
Hidden file. Skipping: samples/uncategorized/.DS_Store
Hidden file. Skipping: samples/undercut/.DS_Store
Hidden file. Skipping: samples/unclean/.DS_Store

Predictions:
Sample 1: Predicted Label: bad angle, True Label: bad angle
Sample 2: Predicted Label: bad angle, True Label: bad angle
Sample 3: Predicted Label: unclean, True Label: unclean
Sample 4: Predicted Label: bad angle, True Label: bad angle
Sample 5: Predicted Label: undercut, True Label: undercut
Sample 6: Predicted Label: bad angle, True Label: bad angle
Sample 7: Predicted Label: bad angle, True Label: bad angle
Sample 8: Predicted Label: unclean, True Label: unclean
Sample 9: Predicted Label: uncategorized, True Label: uncategorized
Sample 10: Predicted Label: bad angle, True Label: bad angle
Sample 11: Predicted Label: bad angle, True Label: bad angle
Sample 12: Predicted Label: uncategorized, True Label: uncategorized

Accuracy: 100.0%
```

FIGURE 9: TERMINAL OUTPUT OF ORIG_BAD_AUDIO_CLASSIFIER.PY

**Note: accuracy was 100% with this testing set but may vary with other testing sets.*

Currently both testing scripts place 80% or 50% of the provided samples into the training set, and 20% or 50% into the testing set. This can be changed by modifying the test_size and train_size variables in the following line. This is line 94 for audio_classifer.py, and line 106 for orig_bad_audio_classifier.py.

```
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, train_size=0.8, random_state=0)
```

FIGURE 10: TESTING CODE LINE TO MODIFY

For a detailed description of the code implementation, please refer to the comments within the scripts.

Human Machine Interface (HMI) -

The goal of the HMI subsystem is to show the quality of the weld identified by the ML algorithm to the welder as they are welding. To build the HMI subsystem, we constructed a hardware-based HMI using a traffic light system with three colors: green, yellow, and red (Figure 11). These colors are indicated by LED lights that represent the status of the weld. We utilized a breadboard, resistors, and an Arduino connected to Arduino software to receive input strings indicating whether the weld was a "good weld" or "bad weld" and output the corresponding light (Figure 12). This was a preliminary proof of concept to show that it was possible to display the status of a weld. However, while making this initial version, we ran into challenges with merging the Arduino code with the compiled machine learning algorithm. The Arduino code, written in C, and the machine learning algorithm, developed in Python, required a bridge to communicate effectively. Additionally, the information on the breadboard is discrete and restricted to displaying information through the three colored lights.

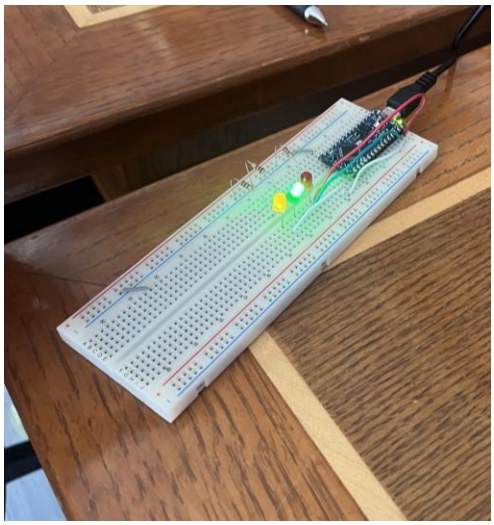

FIGURE 11: PRELIMINARY BREADBOARD ALERT SYSTEM

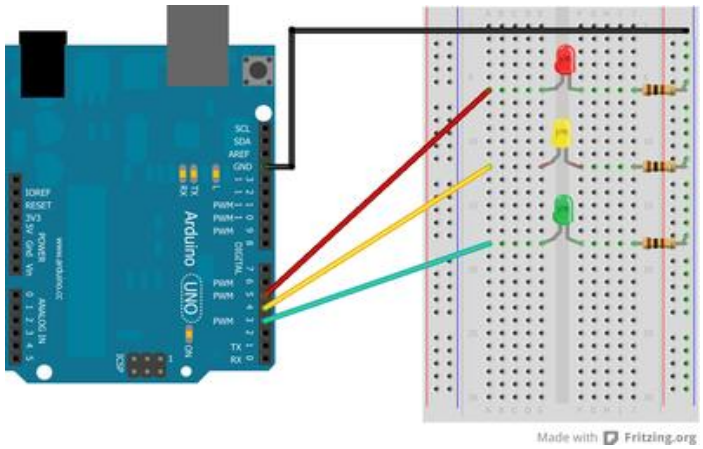

FIGURE 12: ALERT SYSTEM FRITZING FOR THE ARDUINO

After reflecting on integrating the algorithm's code in Python with the Arduino code in C, we pivoted to a software-based notification system in Python. The Python code took a string as an input and outputted a large green or red box indicating the weld's status, as well as listing the error type and accuracy. The effective workflow of the Machine Learning Algorithm and HMI subsystem is displayed in Figures 3 and 4.

The statuses create a new popup window once the machine learning algorithm has reached a conclusion about the weld audio file. This popup window refreshes when a new audio file is inputted. For further work on the HMI, a drop-down feedback option could be added to the popup window for the welder to provide feedback to the machine learning algorithm for it to learn from its

mistakes and improve its accuracy. The HMI could also include a yellow color indicating an inconclusive result if the accuracy value is not acceptable or there was an error in the audio file.

Economic Analysis:

In Caterpillar's production process, if a poor-quality weld is not immediately detected, it will need to be reworked later on the production line. As the bad weld goes further through production, and the overall product becomes larger and more developed, it becomes more difficult and thus more expensive for CAT to rework (up to 3 times the original cost of the weld)[6]. These costs encompass labor, materials expense and the delay in project completion. By creating an ML Algorithm to help welders and welding robots catch poor-quality welds as they happen, CAT will reduce rework and the costs associated. An ML algorithm can also reduce rework and material waste by providing recommendations to welders, depending on the type of issue detected, to prevent welders repeating mistakes.

Another advantage of developing a machine learning model for welding acoustics is its potential to assist incoming welders. With over 160,000 welders nearing retirement age, there is an anticipated surge in new welding professionals (Figure 13). Implementing a machine learning algorithm to detect errors will reduce new welders' mistakes and help new welders learn how to identify issues based on sound.

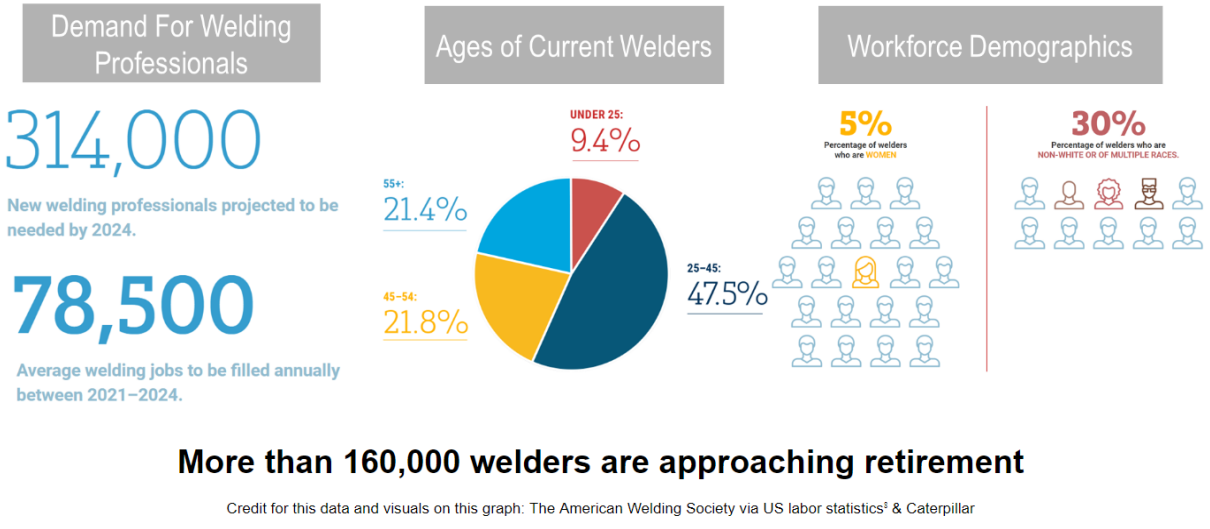

FIGURE 13: THE FUTURE OF WELDING – A CRUCIAL ISSUE

Critical design targets include reducing rework and material waste by improving weld inspection accuracy and training new welders. Constraints include budget limitations and the need for a user-friendly, efficient solution. Functional requirements include real-time monitoring, accurate detection of bad welds, and integration with existing processes. Savings analysis is based on robot rework savings, labor rework savings, raw materials savings, robotic setup expense, and manual setup expense.

Cost savings from reduced rework and material waste are conservatively estimated to be 30% of rework costs but could potentially be as high as 75% according to CAT professionals. Direct costs include $3,250 per station for Xiris WeldMics (Table 2), approximately $200 for monitors, and no additional software costs due to the use of open-source software. As stated, CAT currently operates nearly 7,000 stations. Within the almost 7,000 weld stations, CAT operates roughly 500 robotic weld stations (Figure 14). Although there are vastly more manual weld stations, the robotic weld stations produce more welds per station and provide a greater opportunity for rework savings.

TABLE 1: MICROPHONE PRICING

Part #	Description	Qty	Price (US$)
880-01781	**WeldMic Standalone System,** including: • Microphone with protective sleeve and 1M cable. • Audio Processing Module • USB cable interface to PC • Microphone Mounting Bracket	1	$3,250

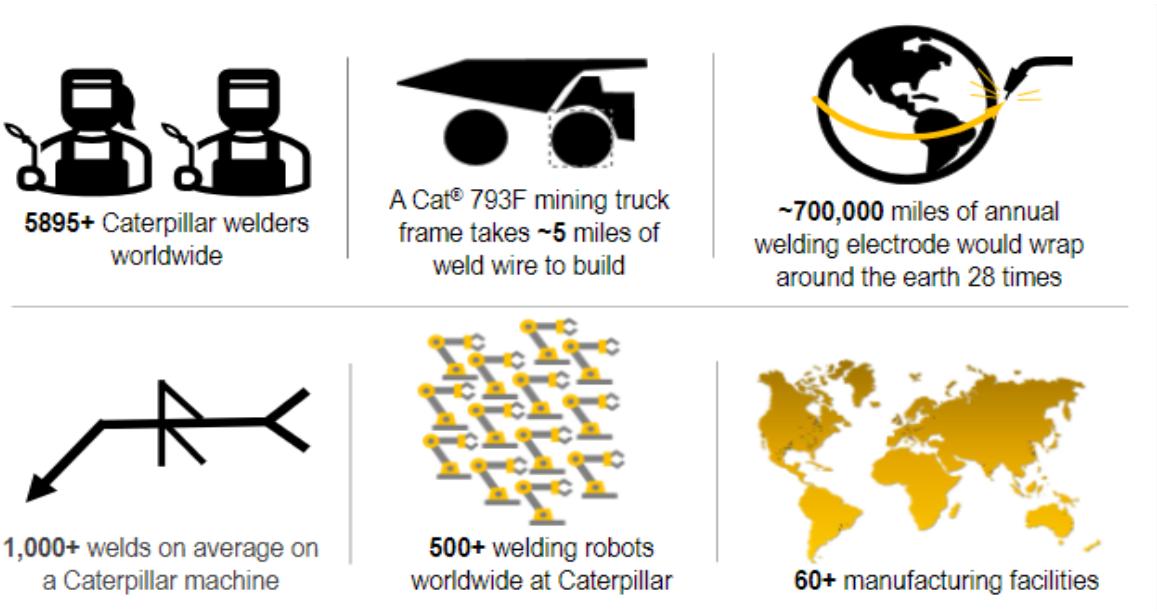

FIGURE 14: CAT WELDING SUMMARY STATISTICS

We estimate that this project will be conservatively implemented in 25% of manual welding stations and nearly 100% of robotic welding stations. Estimates assume that there is already a computer at each welding station, hence why we have only a 25% implementation rate at manual stations. We estimate rework savings on an annual basis. After the project is fully implemented, we estimate the savings on robotic rework to range from $10.5M to $26.25M annually (Table 2). CAT does not record manual rework expenses so, we conservatively estimated manual rework to be 10-15% of weld labor hours. Additionally, we included the machine learning algorithm's 73% detection rate in our manual rework savings. Savings from manual rework can be broken down into labor and raw materials costs. Using the 25% implementation rate within manual welding stations, we estimate labor rework savings to range from $3,285,000 to $9,855,000 annually[B-1] and raw materials savings to range from $231,264 to $693,792 (Table 2). Implementation costs are estimated to be one-time costs at a total of $7.3M (Table 3) with additional expenses incurred as hardware breaks from wear and tear over the years. In our analysis, once ML stations are implemented in 25% of manual weld stations and near 100% of robotic weld stations, estimated savings minus expenses will be $13,735,952 to $36,518,480 annually (Table 3, Figure 15). Estimates have been presented to and approved by Caterpillar Welding.

TABLE 2: COST SAVINGS ANALYSIS

Annual Savings	Robot Rework	Labor Rework	Raw Material	Total Savings
Lower Bound	$35M * 30% = **$10.5M**	**$3,285,000** [B-1]	$3,285,000 * 7.04%[B-1] = **$231,264**	$10.5M + $3,285,000 + $231,264= **$14,016,764**
Upper Bound	$35M * 75% = **$26.25M**	**$9,855,000**	$9,855,000 * 7.04% = $**693,792**	$26.25M + $9,855,000 + $693,792 = **$36,798,792**

TABLE 3: ROLLOUT ANALYSIS

Lower Bound										
Total manual stations est.	1625									
	2024	2025	2026	2027	2028	2029	2030	2031	Total	
Robot Rework Savings	$ 1,050,000.00	$ 4,200,000.00	$ 8,400,000.00	$ 10,500,000.00	$ 10,500,000.00	$ 10,500,000.00	$ 10,500,000.00	$ 10,500,000.00	$ 66,150,000.00	
Labor Rework Savings	$ -	$ 328,500.00	$ 985,500.00	$ 1,971,000.00	$ 2,628,000.00	$ 2,956,500.00	$ 3,120,750.00	$ 3,285,000.00	$ 15,275,250.00	
Raw Materials Savings		$ 23,126.40	$ 69,379.20	$ 138,758.40	$ 185,011.20	$ 208,137.60	$ 219,700.80	$ 231,264.00	$ 1,075,377.60	
Robotic Setup Expense	$ (172,500.00)	$ (517,500.00)	$ (690,000.00)	$ (345,000.00)	$ -	$ -	$ -	$ -	$ (1,725,000.00)	
Manual setup expense	$ -	$ (560,625.00)	$ (1,121,250.00)	$ (1,681,875.00)	$ (1,121,250.00)	$ (560,625.00)	$ (280,312.50)	$ (280,312.50)	$ (5,606,250.00)	
Total Savings/(Expense)	**$ 877,500.00**	**$ 3,473,501.40**	**$ 7,643,629.20**	**$ 10,582,883.40**	**$ 12,191,761.20**	**$ 13,104,012.60**	**$ 13,560,138.30**	**$ 13,735,951.50**	$ 75,169,377.60	
								Implementation Cost	$ (7,331,250.00)	
Manual setup %	0%	10%	30%	60%	80%	90%	95%	100%		
Upper Bound										
Total manual stations est.	1625									
	2024	2025	2026	2027	2028	2029	2030	2031	Total	
Robot Rework Savings	$ 2,625,000.00	$ 10,500,000.00	$ 21,000,000.00	$ 26,250,000.00	$ 26,250,000.00	$ 26,250,000.00	$ 26,250,000.00	$ 26,250,000.00	$ 165,375,000.00	
Labor Rework Savings	$ -	$ 985,500.00	$ 2,956,500.00	$ 5,913,000.00	$ 7,884,000.00	$ 8,869,500.00	$ 9,362,250.00	$ 9,855,000.00	$ 45,825,750.00	
Raw Materials Savings		$ 69,379.20	$ 208,137.60	$ 416,275.20	$ 555,033.60	$ 624,412.80	$ 659,102.40	$ 693,792.00	$ 3,226,132.80	
Robotic Setup Expense	$ (172,500.00)	$ (517,500.00)	$ (690,000.00)	$ (345,000.00)	$ -	$ -	$ -	$ -	$ (1,725,000.00)	
Manual setup expense	$ -	$ (560,625.00)	$ (1,121,250.00)	$ (1,681,875.00)	$ (1,121,250.00)	$ (560,625.00)	$ (280,312.50)	$ (280,312.50)	$ (5,606,250.00)	
Total Savings/(Expense)	**$ 2,452,500.00**	**$ 10,476,754.20**	**$ 22,353,387.60**	**$ 30,552,400.20**	**$ 33,567,783.60**	**$ 35,183,287.80**	**$ 35,991,039.90**	**$ 36,518,479.50**	$ 207,095,632.80	

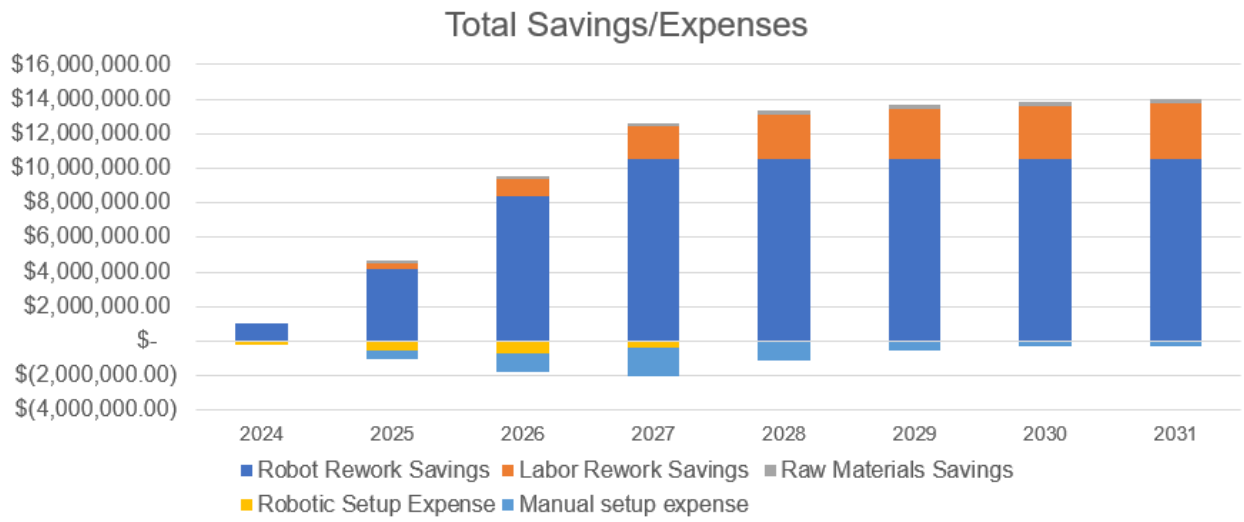

FIGURE 15: GRAPH OF ESTIMATED TOTAL SAVINGS AND EXPENSES EACH YEAR

Conclusions:

It was determined that the machine learning algorithm developed for the use of classifying Caterpillar's welds is functional with a success rate of 84%. This code was run as a series of Python scripts that worked to input acoustic data and output data as electrical signals with an alarm system. By use of this program, Caterpillar welders can reduce their errors with real-time monitoring feedback on weld quality, creating $13.7M to $36.5M of potential net savings each year.

The diversity of the group consisting of both engineering and business students allowed for a unique opportunity to approach the project with members of varying subjects of expertise. These varying perspectives are great for enhancing project quality but could have been utilized even more by working early to share knowledge between different subgroups so that each member could have a more generalized understanding of the final product. This would have allowed business students to understand technical engineering areas of the project further, and the engineers to understand financial analysis further, contributing to a more well-rounded comprehension by each member and a potential increase in overall team productivity if the project were to be continued.

Additionally, it would have been more efficient had we integrated the alarm software with the machine learning program from the beginning, instead of developing it separately and then merging it. This could have shortened the time it takes to properly translate that code and would have allowed

for alarm testing to be done immediately upon finishing the classification software, as opposed to beginning it afterwards. Earlier testing of different welding samples would have allowed for quicker optimization of the machine learning program and more time to gather results.

Moving forward, the group hopes that Caterpillar can implement this program into everyday welding activities and potentially share the technology with additional contractors or welding laboratories that can use it for all types of welding. Continuous optimization of a weld quality monitoring program can allow for an increase in weld quality and therefore an increase in savings.

Recommendations:

The team has recommendations for any similar machine learning work with a welding system in the future. First, we recommend training the machine learning algorithm with more data sets to be able to identify more categories and improve its accuracy. While this is difficult due to time constraints of Caterpillar welders, the number of datasets for early testing of the machine learning program would have benefited heavily from additional data points. We have no minimum or maximum number of data samples per algorithm, ideally, we suggest ~1,000 minutes of each classification of weld to ~10,000 minutes. This can help the data classification work more efficiently and save time during development. This can also be easily achieved, at the scale Caterpillar welds, in the workflow we have established in the solution method block diagram (Figure 16).

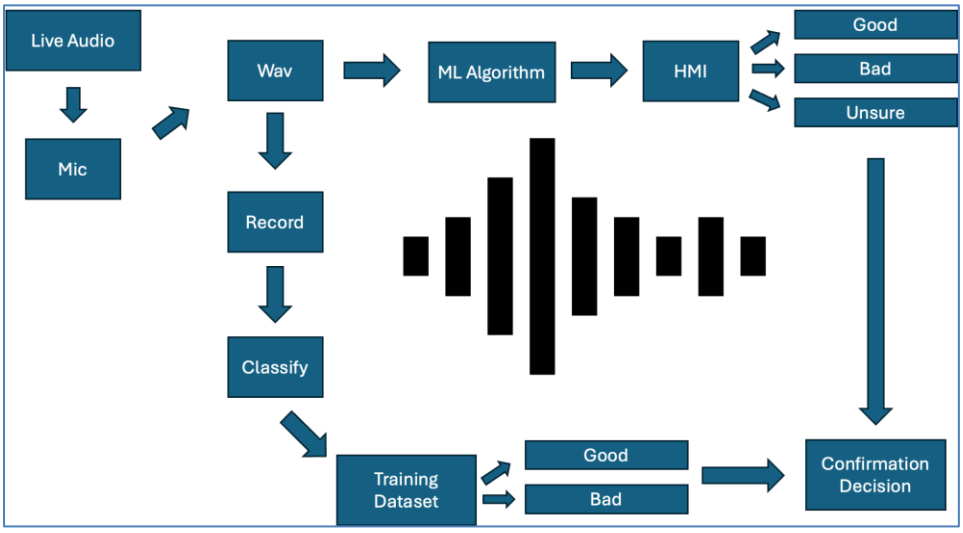

FIGURE 16: IMPLEMENTATION STACK

To further improve the ML algorithm, Caterpillar could implement a feature to receive feedback from welders on classification accuracy of the algorithm. This feedback can be used to train the model further.

If we estimate that these steps were to increase categorical accuracy by 5%, then this would lead to an estimated $225,000 in additional savings annually[B-1].

References:

[1] "Deep Audio Classification." *GitHub*, GitHub, github.com/nicknochnack/DeepAudioClassification/blob/main/AudioClassification.ipynb. Accessed 8 Mar. 2024.

[2] "Amazon.Com: Hausbell Plastic Dish, Replacement Dish, Sound Collecting Amplifier Dish : Electronics." Amazon, www.amazon.com/HAUSBELL-Plastic-Replacement-Collecting-Amplifier/dp/B085HDB8PC. Accessed 30 Apr. 2024.

[3] "Noise Reduction in Python Using Spectral Gating (Speech, Bioacoustics, Audio, Time-Domain Signals)." *GitHub*, GitHub, github.com/timsainb/noisereduce. Accessed 8 Mar. 2024.

[4] Smolic, Hrvoje. "How Much Data Is Needed for Machine Learning? ." *Graphite Note*, Graphite Note, 15 Dec. 2022, graphite-note.com/how-much-data-is-needed-for-machine-learning#:~:text=Generally%20speaking%2C%20the%20rule%20of,(columns)%20in%20your%20dataset.

[5] Tao, J., et al. *Acoustic Signal-Based Automated Control of Welding Penetration Using Digital Twin Technology*, University of Illinois Urbana-Champaign Library Resources, 15 Feb. 2024, eds-p-ebscohost-com.proxy2.library.illinois.edu/eds/detail/detail?vid=0&sid=4c949a5e-3a4f-490e-9f46-0fdd6aa9924f%40redis&bdata=JnNpdGU9ZWRzLWxpdmUmc2NvcGU9c2l0ZQ%3D%3D#db=edselc&AN=edselc.2-52.0-85178151252.

[6] "Understanding the True Cost of Weld Defects." *Miller Electric*, Miller Electric, 17 Dec. 2015, www.millerwelds.com/resources/article-library/the-real-cost-of-missed-or-defective-welds#:~:text=Weld%20failures%20may%20be%20slightly,that%20of%20the%20 original%20 fabrication.

[7] "Arduino Street Traffic Light - Breadboard Edition." Instructables, Instructables, 22 Oct. 2017, www.instructables.com/Arduino-Street-Traffic-Light/.

[8] "Welders, Cutters, Solderers, and Brazers : Occupational Outlook Handbook." *U.S. Bureau of Labor Statistics*, U.S. Bureau of Labor Statistics, 17 Apr. 2024, www.bls.gov/ooh/production/welders-cutters-solderers-and-brazers.htm#tab-1

Appendix:

Appendix A

A-1: final_main.py

```
import numpy as np

import librosa

import soundfile as sf

import os

import matplotlib.pyplot as plt

from sklearn.svm import SVC

from sklearn.preprocessing import StandardScaler

from watchdog.observers import Observer

from watchdog.events import FileSystemEventHandler

import queue

import time


# Initialize the figure and axes for the plot

fig, ax = plt.subplots()


# Define the square vertices

square_vertices = [(0, 0), (0, 1), (1, 1), (1, 0), (0, 0)]

x_values, y_values = zip(*square_vertices)


# Function to update the plot

def update_plot(pred_label_name, bad_classification):

# Determine color based on prediction

if pred_label_name == "bad":

color = 'r'

if bad_classification == "uncategorized":

text = 'Error: Uncategorized'
```

```
if bad_classification == "bad angle":

text = 'Error: Bad Angle'

if bad_classification == "unclean":

text = 'Error: Unclean'

if bad_classification == "undercut":

text = 'Error: Undercut'

elif pred_label_name == "acceptable":

color = 'g'

text = 'Acceptable'


# Clear the axes

ax.clear()

# Plot the square with the corresponding color

ax.fill(x_values, y_values, color)

ax.axis('off')

ax.set_aspect('equal', adjustable='box')

ax.text(0.5, 0.5, text, ha='center', va='center', fontsize=12, color='white')


# Redraw the plot

fig.canvas.draw()

plt.pause(0.01)


# Function to extract features from an audio file

def extract_features(file_path):

target_sample_rate = 44100

try:

audio_data, original_sample_rate = sf.read(file_path)

resampled_audio = librosa.resample(audio_data, orig_sr=original_sample_rate,
target_sr=target_sample_rate)
```

```
# Extract MFCC features

mfccs = librosa.feature.mfcc(y=resampled_audio, sr=target_sample_rate, n_mfcc=40)

return np.mean(mfccs.T, axis=0)

except Exception as e:

print(f"Error while parsing file: {file_path}\nError: {e}")

return None


# Train function

def train():

base_path_train = "samples"

categories = ["acceptable", "bad"]


features_train = []

labels_train = []


for label, category in enumerate(categories):

category_path = os.path.join(base_path_train, category)

for file_name in os.listdir(category_path):

file_path = os.path.join(category_path, file_name)

if os.path.basename(file_path).startswith('.'):

print(f"Hidden file. Skipping: {file_path}")

continue

mfccs = extract_features(file_path)

if mfccs is not None:

features_train.append(mfccs)

labels_train.append(label)

#print("\nExtract features finished for training set.")


features_train = np.array(features_train)

labels_train = np.array(labels_train)
```

```
scaler = StandardScaler()
features_train = scaler.fit_transform(features_train)

#print("\nFeature scaling finished for training set.")

# Initialize and train the classifier
model = SVC(kernel='linear', probability=True)
model.fit(features_train, labels_train)

print("\nClassifier training finished.")

return model, scaler

# Define a custom event handler for monitoring new files
class NewSampleEventHandler(FileSystemEventHandler):
def __init__(self, model, scaler, categories, plot_queue):
self.model = model
self.scaler = scaler
self.categories = categories
self.plot_queue = plot_queue

def on_created(self, event):
# Check if the event is a file and ends with .wav
if not event.is_directory and event.src_path.endswith('.wav'):
file_path = event.src_path
# Extract features from the file
mfccs = extract_features(file_path)
if mfccs is not None:
# Scale the features
```

```
mfccs_scaled = self.scaler.transform([mfccs])

# Predict the label

pred_label = self.model.predict(mfccs_scaled)[0]

pred_label_name = self.categories[pred_label]

print(f"New sample added: {file_path}")

print(f"Predicted Label: {pred_label_name}")

# Add prediction to the queue

# Check if the prediction is "bad" and call program #2

if pred_label_name != 'acceptable':

try:

# Call program #2 using subprocess

bad_classification = bad_classify(file_path)

except Exception as e:

print(f"Error while calling Bad Classifier: {e}")

else:

bad_classification = None

self.plot_queue.put((pred_label_name,bad_classification))




# BAD CLASSIFIER FUNCTIONS BELOW


# Train function

def badtype_train():

base_path_train = "bad_classifier/samples"

categories = ["uncategorized", "bad angle", "unclean", "undercut"]


features_train = []

labels_train = []


for label, category in enumerate(categories):
```

```
category_path = os.path.join(base_path_train, category)

for file_name in os.listdir(category_path):

file_path = os.path.join(category_path, file_name)

if os.path.basename(file_path).startswith('.'):

print(f"Hidden file. Skipping: {file_path}")

continue

mfccs = extract_features(file_path)

if mfccs is not None:

features_train.append(mfccs)

labels_train.append(label)


# print("\nExtract features finished for training set.")


features_train = np.array(features_train)

labels_train = np.array(labels_train)


scaler = StandardScaler()

features_train = scaler.fit_transform(features_train)


# print("\nFeature scaling finished for training set.")


# Initialize and train the classifier

model = SVC(kernel='linear', probability=True)

model.fit(features_train, labels_train)


print("\n Bad Classifier training finished.")


return model, scaler


def bad_classify(sample):
```

```
global running

running = True



# Define categories

bad_categories = ["uncategorized", "bad angle", "unclean", "undercut"]


# Extract features from the file

mfccs = extract_features(sample)

if mfccs is not None:

# Scale the features

mfccs_scaled = bad_scaler.transform([mfccs])

# Predict the label

pred_label = bad_model.predict(mfccs_scaled)[0]

bad_pred_label_name = bad_categories[pred_label]

print(f"Predicted Label: {bad_pred_label_name}")

return bad_pred_label_name




# Main function

if __name__ == "__main__":

global running

running = True

# ax.clear()


# Create a queue to communicate between threads

plot_queue = queue.Queue()


# Train the model and get the scaler
```

```
global model, scaler

model, scaler = train()


# Train the BAD TYPE model and get the scaler too

global bad_model, bad_scaler

bad_model, bad_scaler = badtype_train()




# Define categories

categories = ["acceptable", "bad"]


# Create an event handler for new samples

event_handler = NewSampleEventHandler(model, scaler, categories, plot_queue)


# Create an observer and schedule it

observer = Observer()

observer.schedule(event_handler, path="new samples", recursive=False)

observer.start()


print(f"Monitoring directory: 'new samples'")


# Keep checking the queue for new predictions

try:

while running:

if not plot_queue.empty():

pred_label_name, bad_classification = plot_queue.get()

update_plot(pred_label_name, bad_classification)

plt.pause(0.1) # Allow some time for other events to be handled

except KeyboardInterrupt:

# If a keyboard interrupt occurs (e.g., Ctrl+C), stop the observer and exit the loop
```

```
running = False

observer.stop()

print("Stopping observer due to keyboard interrupt.")


# Stop the observer if the program is interrupted

observer.join()
```

A-2: splice_audio.py

```
import os

import librosa

import soundfile as sf

import math



def splice_audio_directory(input_dir, output_dir, segment_length=10):

    # Create output directory if it doesn't exist

    if not os.path.exists(output_dir):

        os.makedirs(output_dir)


    # Iterate over all files in the input directory

    for filename in os.listdir(input_dir):

        if filename.endswith(".wav"):

            input_file = os.path.join(input_dir, filename)

            splice_audio(input_file, output_dir, segment_length)



def splice_audio(input_file, output_dir, segment_length=10):

    # Load the audio file
```

```
    audio_data, sr = librosa.load(input_file, sr=None)


    # Calculate total number of segments
    num_segments = math.ceil(len(audio_data) / (segment_length * sr))


    # Create output directory if it doesn't exist
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)


    # Get the original file name without extension
    file_name = os.path.splitext(os.path.basename(input_file))[0]


    # Iterate over segments and save them as separate files
    for i in range(num_segments):
        start_sample = i * segment_length * sr
        end_sample = min((i + 1) * segment_length * sr, len(audio_data))  # Make sure
end_sample doesn't exceed length
        segment = audio_data[start_sample:end_sample]
        output_file = os.path.join(output_dir, f"{file_name}_segment_{i + 1}.wav")
        sf.write(output_file, segment, sr)



splice_audio_directory("Path to Unspliced Audio Directory", "Path to Store Spliced
Audios")
```

A-3: audio_classifier.py

```
import numpy as np
import librosa
```

```
import soundfile as sf

import os

import joblib

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score

import matplotlib.pyplot as plt




###################################### display code below


def plot_color(input_text):

square_vertices = [(0, 0), (0, 1), (1, 1), (1, 0)] # Vertices of a square

square_vertices.append(square_vertices[0]) # Closing the square

x_values, y_values = zip(*square_vertices) # Unzipping the vertices into x and y
coordinates


# Determine the plot settings based on the input text

if "bad" in input_text.lower():

color = 'r' # Red color

# if "porosity" in input_text.lower():

# # Additional text for porosity

# plt.text(0.5, 0.5, "Error: Porosity", ha='center', va='center', size=12, color='white')

# plt.text(0.5, 0.3, "Accuracy: 73%", ha='center', va='center', size=12, color='white')

elif "acceptable" in input_text.lower():

color = 'g' # Green color

else:

print("Input text is not recognized.")

return
```

```
# Fill the square with the determined color
plt.fill(x_values, y_values, color)


# Hide axis lines and values
plt.axis('off')
plt.gca().set_aspect('equal', adjustable='box') # Set equal aspect ratio
plt.show()









######################################### model code below


# Function to extract features from an audio file
def extract_features(file_path):
target_sample_rate = 44100
try:
# Load audio data from the input file
audio_data, original_sample_rate = sf.read(file_path)
# Resample the audio data to the target sample rate
resampled_audio = librosa.resample(audio_data, orig_sr=original_sample_rate,
target_sr=target_sample_rate)


# Extract MFCC features from the resampled audio
mfccs = librosa.feature.mfcc(y=resampled_audio, sr=target_sample_rate, n_mfcc=40)
mfccs_processed = np.mean(mfccs.T, axis=0)
except Exception as e:
print("Error encountered while parsing file:", file_path)
```

```
print("Error:", e)

return None

return mfccs_processed


# Directory where your dataset is located

base_path = "samples"

categories = ["acceptable", "bad"] # Change these to your actual category folder names


# Load dataset and extract features

features = []

labels = []

for label, category in enumerate(categories):

for file_name in os.listdir(os.path.join(base_path, category)):


file_path = os.path.join(base_path, category, file_name)


if os.path.basename(file_path).startswith('.'):

print("Hidden file. Skipping: " + file_path)

else:

mfccs = extract_features(file_path)

if mfccs is not None:

features.append(mfccs)

labels.append(label)




#print("\n extract_features finished.")


# Convert into a numpy array

features = np.array(features)

labels = np.array(labels)
```

```
# Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2,
train_size=0.8, random_state=0)


# Feature scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)



#print("\n feature scaling finished.")


# Initialize and train the classifier
model = SVC(kernel='linear', probability=True)
model.fit(X_train, y_train)


#print("\n classifier training finished.")


# Define label names corresponding to numerical labels
label_names = ["acceptable", "bad"]


# Predicting the Test set results
y_pred = model.predict(X_test)


print("\nPredictions:")
for i, (pred_label, true_label) in enumerate(zip(y_pred, y_test)):
pred_label_name = label_names[pred_label]
true_label_name = label_names[true_label]
print(f"Sample {i+1}: Predicted Label: {pred_label_name}, True Label: {true_label_name}")
```

```
# plot_color(pred_label_name)




# Calculate the accuracy

accuracy = accuracy_score(y_test, y_pred)

print(f"\nAccuracy: {accuracy*100}%")
```

Appendix B

B-1: Savings calculations

Manual Rework Savings -

Lower: $75 labor rate (According to Caterpillar Welding Department) * 10% manual weld time spent reworking * 73% detection rate * 2000 hours * (6000 welders* 25% implementation) * 20% rework time prevention by ML algorithm = **$3,285,000 annually**

Upper: $75 labor rate * 15% manual weld time spent reworking * 73% detection rate * 2000 hours * (6000 welders* 25% implementation) * 40% rework time prevention by ML algorithm = **$9,855,000 annually**

Manual Rework Savings With 78% Detection Conservative –

Lower: $75 labor rate (According to Caterpillar Welding Department) * 10% manual weld time spent reworking * 78% detection rate * 2000 hours * (6000 welders* 25% implementation) * 20% rework time prevention by ML algorithm = **$3,510,000 annually**

$3,510,000 - $3,285,000 = $225,000 annual savings compared to 73% detection

Material Waste Savings -

According to Miller Electric, one of the largest global welding companies, “In a properly functioning welding operation, costs are typically broken down like this:

Labor: 85 percent

Filler metals: 6 percent

Raw materials: 4 percent

Shielding gas: 3 percent

Electric power: 2 percent

When a missed or defective weld goes undetected, these costs escalate at every stage in the welding operation and beyond”[6].

On average, raw materials expenses equate to 15%/85% of labor expenses. In a situation where rework is taking place, 1/2.5 of the expenses relate to the initial weld instead of rework time allowing us to calculate raw materials expense as 7.04% ((15% / 85%)*(1/2.5))

Lower bound: $3,285,000 * 7.04% = $231,264

Upper Bound: $9,855,000 * 7.04% = $693,792

Appendix C

C-1 Additional Tables and Figures

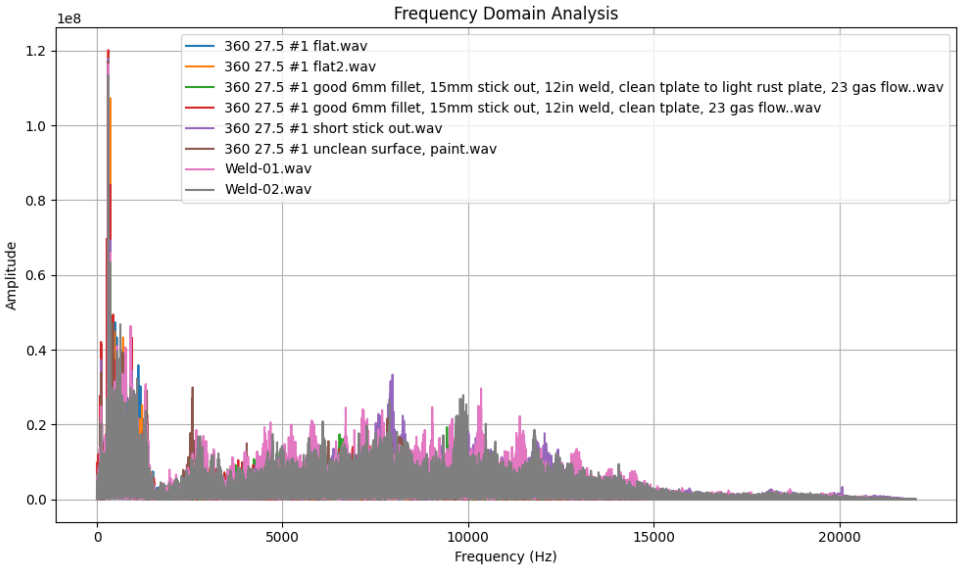

FIGURE 17: EXAMPLE OF "GOOD" PLOTS FOR FREQUENCY DOMAIN ANALYSIS

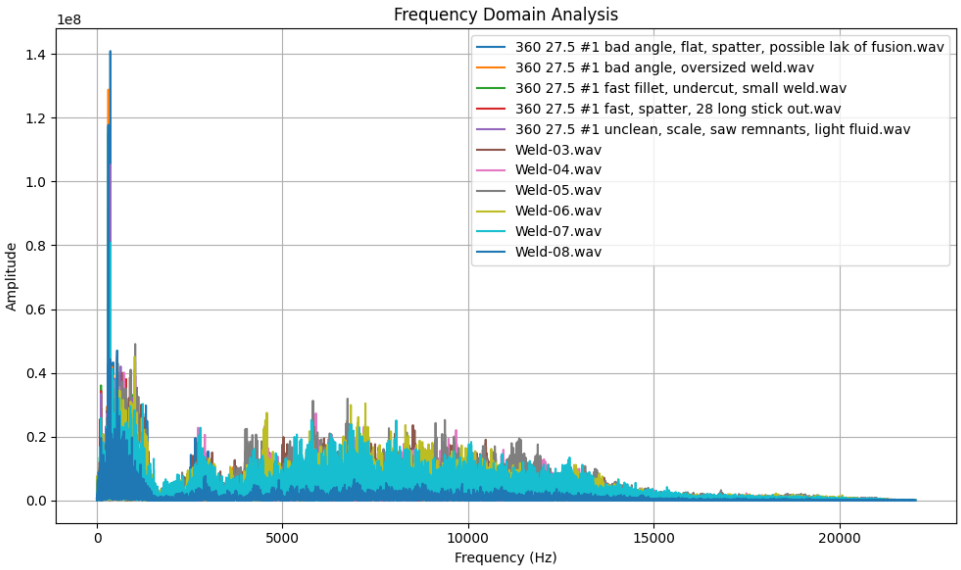

FIGURE 18: EXAMPLE OF "BAD" PLOTS FOR FREQUENCY DOMAIN ANALYSIS

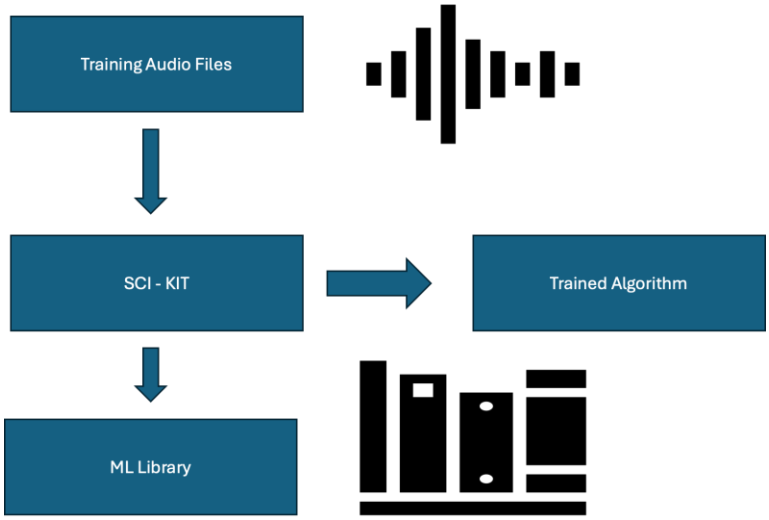

FIGURE 19: MACHINE LEARNING ALGORITHM BUILD

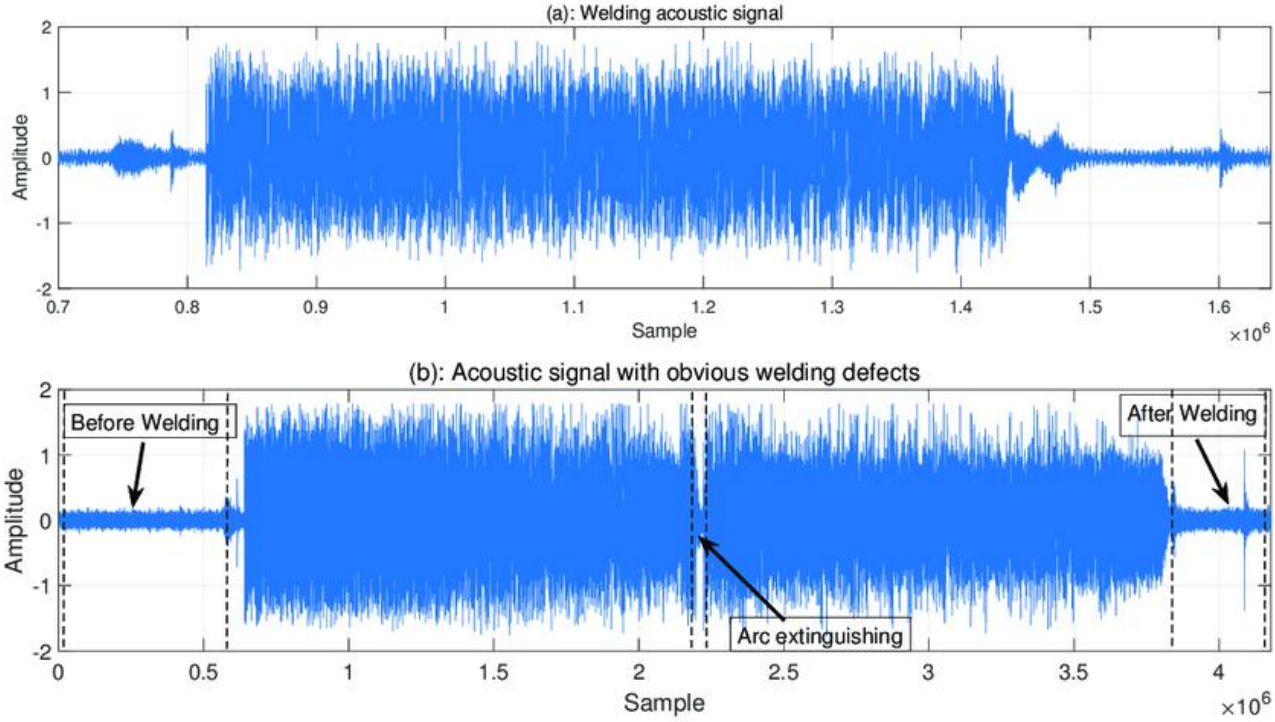

FIGURE 20: WELDING ACOUSTIC SIGNAL WITHOUT DEFECTS AND WITH OBVIOUS DEFECTS

TABLE 4: SOFTWARE PRICING

WeldStudio™ 3 Pro - Software Licensing required per system (PC)

Part #	Description	Price (US$)
897-WSP-3D60	WeldStudio™ 3 Pro, 60 Day Demo License Only. (*Demo fee can be applied towards the purchase of a license, perpetual or 1 year, if ordered before the end of the trial.)*	$250
897-WSP-3PER	WeldStudio™ 3 Pro, Permanent Access Digital License with 12 months updates and support. *Advanced Xiris Camera Software Utility that includes Machine Vision Tools, Measurement Tools and Real Time Streaming Protocol.*	$2,950
897-WSP-3MNT	1 year extension for ongoing maintenance updates and support for **WeldStudio™ 3 Pro**.	$350 per year
OR 897-WSP-3Y01	WeldStudio™ 3 Pro, 1 Year Access Digital License with updates and support. *(License will expire after 12 Months unless renewed)*	$1,500 per year