Classification Machine Learning Model for Faults in a Non-Inverting Amplifier with Altium's Monte Carlo Simulation

Santiago Vázquez Fernández
September 2025

Abstract

During my first year electronic labs, determining whether a circuit was functioning correctly from oscilloscope traces was a critical skill. This is because the reliability of larger prototypes depends on the correct behaviour of basic building blocks such as amplifiers. Faults such as short circuits and open circuits are among the most common and can significantly affect circuit performance.

This project developed a machine learning approach to classify the operation of a simple non-inverting amplifier circuit as healthy, open fault, or short fault using time-voltage waveforms obtained from circuit simulations with non-ideal components. The full project is available in the GitHub repository (circuit_fault_detection). A custom dataset was generated in Altium Designer using a realistic LM324N operational amplifier model and Monte Carlo transient analysis across multiple frequencies in the range 200-2000 Hz, yielding 900 simulated runs and oscilloscope-like voltage signals for training the model.

The pipeline, implemented in Python 3 via Google Colab (circuit_fault_pipeline.ipynb), included data loading, waveform standardisation, feature extraction (like harmonics, phase, noise), and classification with a Random Forest model. The results demonstrated clear class separation, and confusion matrices confirmed accurate classification on both training and test data, while cross-validation verified stability, and control experiments such as label shuffling and duplicate checks demonstrated that the model was not in fact overfitting.

An interactive demonstration (demo.ipynb) was also created, allowing users to upload sample time-voltage CSV files generated in Altium, and obtain a predicted class; from healthy, short, or open; with associated probabilities and waveform plots to confirm the model's prediction from the user's perspective. Overall, the workflow aims to demonstrate how circuit simulation data and machine learning can be combined to automate fault diagnosis in circuit building projects and in analogue hardware.

Contents

I. Introduction

Reliable detection of faults such as short and open circuits is essential in electronics, as the correct behaviour of larger prototypes depends on the integrity of individual building blocks. Traditional diagnosis relies on interpreting oscilloscope traces, a process that can be time-consuming and error prone. This project explores a machine learning approach to automate circuit fault detection using a Random Forest classification model. Using simulated oscilloscope waveforms from a non-inverting amplifier in Altium, a pipeline was implemented to classify circuits as healthy, open, or short.

II. Dataset Building

Unlike standard image or text machine learning classification problems, no open datasets exist for circuit fault classification in analogue amplifiers. For this reason, a custom dataset had to be created using Altium Designer's semi-realistic manufacturer part components, focusing on a simple but widely used configuration, the classic non-inverting amplifier. This allowed the pipeline to be tested on realistic but manageable data, while ensuring that the workflow could be replicated and extended for more complex circuits in future implementations.

i. Amplifier Features

The amplifier used was the LM324N operational amplifier. Importantly, this was not an idealized model but one including non-ideal behaviour such as slew rate and clipping, which ensured that the simulation output resembled real laboratory measurements more closely [1, 2].

Three configurations were built on Altium schematics to represent the three classes for the dataset to distinguish from [3]:

- Healthy: Standard non-inverting amplifier with a gain of 11, implemented using R_f=10 kΩR and R_1=1 kΩR.

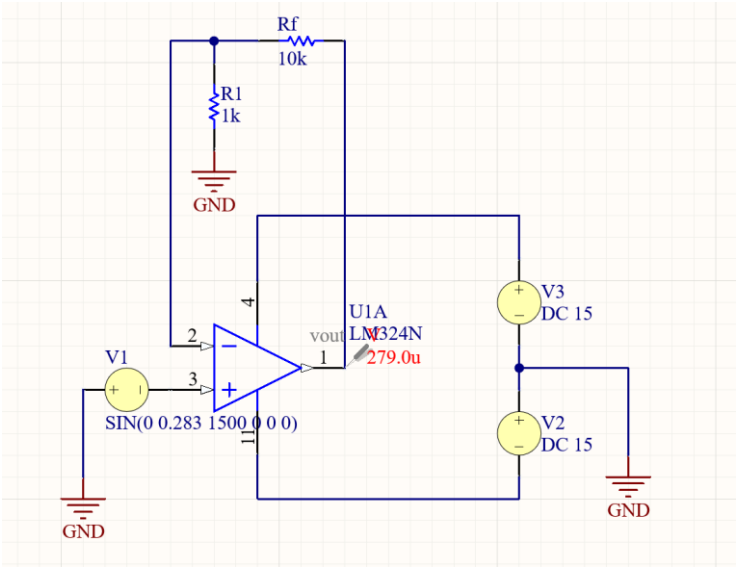

Figure 1: Healthy circuit schematic of a non-inverting amplifier with gain 11 and sinusoidal input source in Altium Designer.

- Short fault: Created by removing R_1 from the circuit while keeping R_f=10 kΩR. In this configuration, the feedback path collapses, effectively short-circuiting the amplifier input stage.

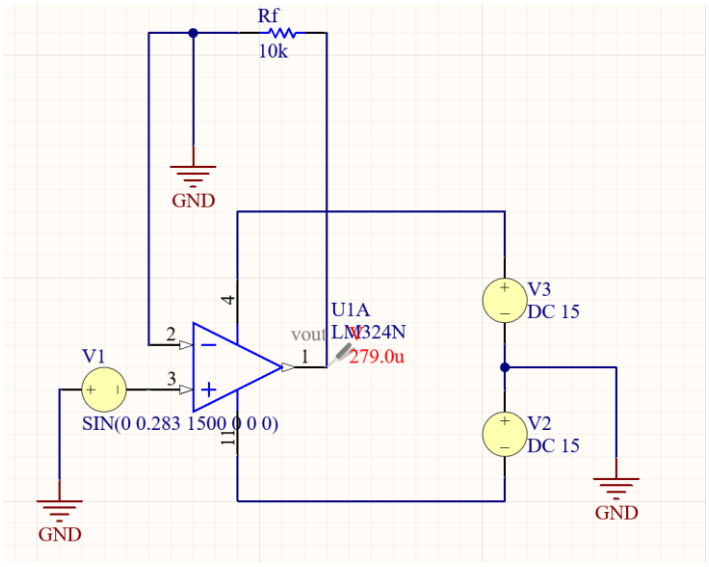

Figure 2: Short circuit schematic of a faulty non-inverting amplifier with a sinusoidal input source in Altium Designer.

- Open fault: Created by setting R_1=1×10^9 ΩR, simulating an open-circuit condition. With no feedback path, the op-amp should saturate or behaves in an unstable manner.

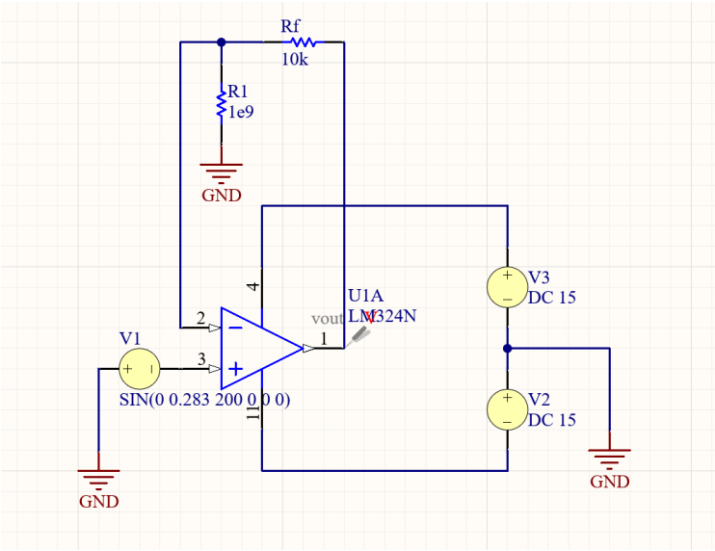

Figure 3: Open circuit schematic of a faulty non-inverting amplifier with a sinusoidal input source in Altium Designer.

This choice of faults (healthy, short, and open) was motivated by their practical relevance in real circuit debugging, where incorrectly soldered resistors, broken traces, or misplaced components often present as opens or shorts. Other fault types (like clipping, saturation, biasing errors) were not included due to dataset size constraints, but the workflow is easily extendable to these cases if this project were to be extended to further classify, or also extended to other specific circuits like inverting amplifiers, comparators, etc.

The relevance of these circuits lies in their transferability because the LM324N is a widely used op-amp, and non-inverting amplifier blocks are fundamental in analogue systems. Thus, while simple, the dataset represents a meaningful test case for this machine learning model.

ii. Transient Analysis with Monte Carlo Simulation

Each circuit was simulated under sinusoidal excitation at frequencies of 200, 500, 800, 1000, 1500, and 2000 Hz with a fixed amplitude of 0.283 V.

- Transient setup: A simulation time step of 2.5 µs and a total duration of 0.12 s was used. This ensured a sufficiently fine resolution for harmonic content analysis while maintaining manageable file sizes for machine learning preprocessing.

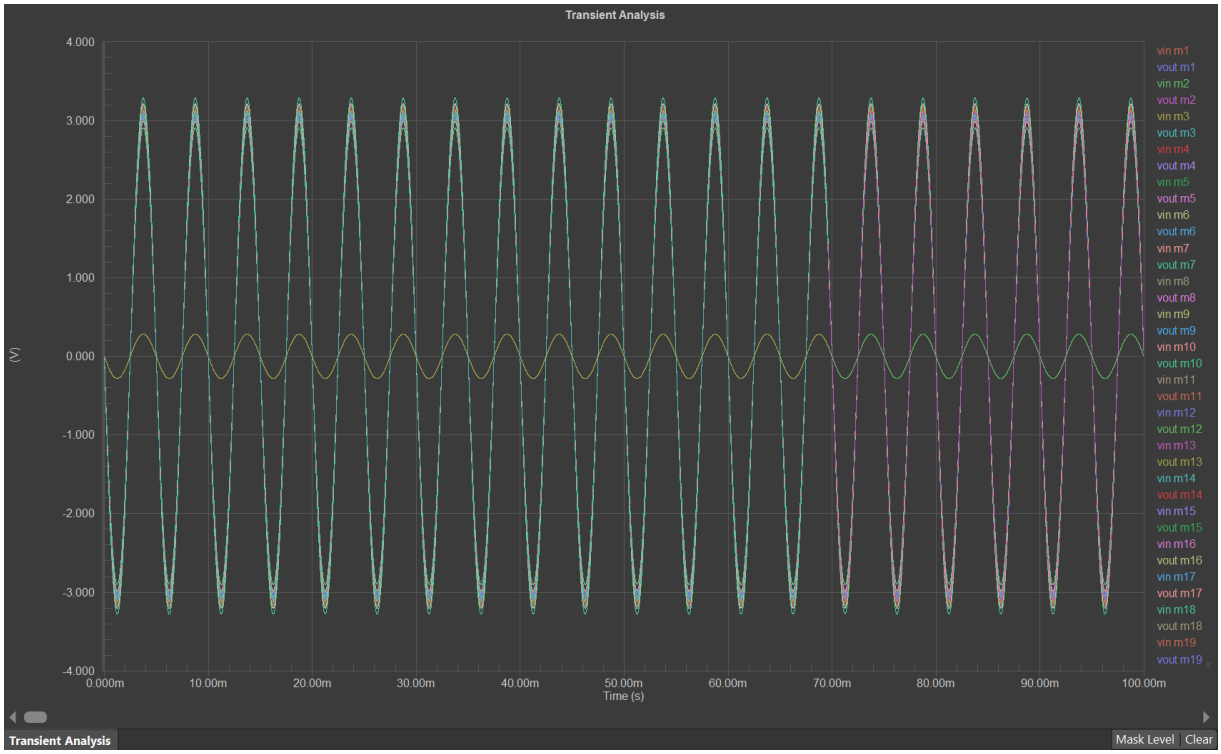

Figure 4: Transient analysis simulation with window 0.12s and time step 2.5 µs of a healthy non-inverting amplifier in Altium Designer with Monte Carlo parameters fitted across 50 runs.

- Monte Carlo simulation: For each frequency and fault condition, 50 Monte Carlo runs were performed. Gaussian-distributed resistor variations (tolerance = 5%) were applied, with random seed = –1 for reproducibility. These selected parameters made sure that there was variability across waveforms, mimicking component tolerances in real hardware while preventing the model from overfitting to perfectly identical signals 50 times [4].

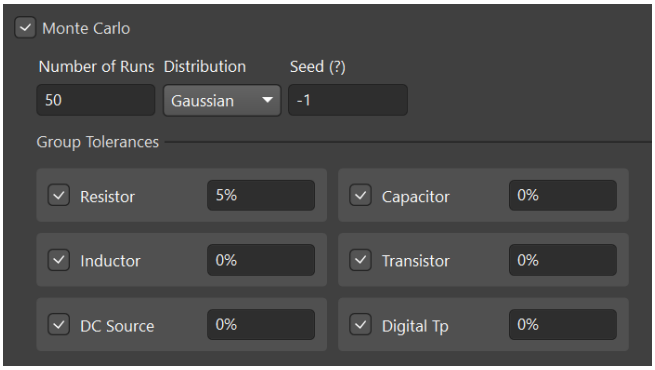

Figure 5: Monte Carlo parameters fitting in Transient Analysis menu in Altium Designer with runs, seed, and tolerances established.

This setup produced a total of 900 simulations (3 fault conditions × 6 frequencies × 50 Monte Carlo runs per frequency).

iii. Dataset Organisation

The simulated signals were exported as CSV files with the following format:

- 1 time column labelled ‘s’.
- 50 columns of output voltage runs, labelled ‘vout1 … vout50’.

The dataset was organised in a hierarchical folder structure:

```
data/
   healthy/
      200.csv
      500.csv
      ...
   open_r1/
      200.csv
      ...
   short_r1/
      200.csv
      ...
```

Cell 1: Hierarchical folder structure of the CSV files used to train and test the ML model.

This structure allowed the pipeline to easily iterate through classes and frequencies. Storing the files in CSV format ensured both human readability and compatibility with Python libraries such as Pandas. The entire dataset could also be compressed into a single ZIP archive for efficient loading into Google Colab to build the trained model.

III. Circuit Fault Detection Pipeline

i. Loading Data

The CSV files were stored such that the first column contains the simulation time, and the next 50 columns contain the results of the Monte Carlo simulation runs (`vout1, vout2, …, vout50`). So, when loaded, the time vector (`t`) is created and read as a one dimensional array. This defines the time base against which the output voltages are measured. The matrix of runs (`V`) is a two dimensional array, with shape (`N_points, N_runs`). Each column of (`V`) corresponds to one Monte Carlo run, preserving all 50 voltage waveforms for that circuit condition. These are not appended together, but rather stored side by side in matrix form, so that downstream functions can process each run independently while keeping the time base consistent. The loading data function (`read_multirun_vout_csv`) is presented below:

```
def read_multirun_vout_csv(path: str, n_keep=50):
    df = pd.read_csv(path)
    t = df["s"].to_numpy(float)
    V = df.iloc[:, 1:1+n_keep].to_numpy(float)
    return t, V
```

Cell 2: Read-in function for CSV files and assigning the `t`, and `V` variables to the correct columns.

The argument (`n_keep=50`) specifies how many runs are loaded. This ensures that the model always sees a fixed number of input columns. If fewer than 50 columns exist, slicing still works safely without breaking the pipeline, which makes the code robust to variations in dataset size.

The function returns raw NumPy arrays instead of DataFrames. This choice is deliberate because NumPy arrays are more efficient for vectorised signal processing and in removing any dependency on column labels once the data has been processed. Maintaining a consistent input and output format is critical for machine learning pipelines. If the CSV structure were inconsistent (for example, if columns were reordered, renamed, or missing), the model could train on mismatched data, leading to unreliable results, where schema drift arises. By enforcing a strict structure, time in the first column, voltage runs in the next 50 columns, the pipeline guarantees that the same features are extracted across all files.

ii. Key Pipeline Functions

a. Standardization

Before feature extraction, each waveform is standardised by extracting a fixed 0.1 s time window from the start of the trace and resampling it onto a uniform grid. The function is presented below:

```
def standardise(t, v, time_window=0.1, n_points=16384):
    t0 = t[0]
    mask = (t - t0) < time_window
    t2, v2 = t[mask], v[mask]
    t_fit = np.linspace(0.0, time_window, n_points, endpoint=False)
    v_fit = np.interp(t_fit, t2 - t0, v2)
    return t_fit, v_fit
```

Cell 3: Standardising function, establishing the time window, number of points, and returning the fitted variables.

The function is required because Altium's transient simulations do not produce evenly spaced time samples as some simulations place denser samples where the waveform changes quickly, especially at the peaks. Without resampling, the data would be irregular and unsuitable for direct use in machine learning. In early trials, using raw simulator data produced poor results, because the model was effectively comparing signals with inconsistent sampling densities.

Moreover, the argument (`time_window=0.1`) ensures that the time window is long enough to capture multiple waveform cycles even at the lowest simulated frequency (200 Hz having approximately 20 cycles), so that harmonic and phase information are stable and used to train the model. Longer windows would increase storage size and slow training unnecessarily, while shorter windows might capture too few cycles for reliable harmonic extraction.

Finally, the sampling number choice (`n_points=16384`) (2^{14}) provides a dense sampling grid with high frequency resolution while keeping arrays manageable. The power of two length leverages the binary nature of digital processing, ensuring numerically stable interpolation and compatibility with FFT-based operations if extended later. Overall, the standardising function was essential as waveform feature extraction would be inconsistent and model predictions would not be reliable enough.

b. Feature extraction

After standardising each waveform, the next step is to extract compact and interpretable features that capture the circuit's waveform behaviour. The function below projects the signal onto sinusoids at the fundamental corresponding frequency at which the circuit was run (`f0`) and its first two harmonics (`2*f0, 3*f0`) to derive the total mean square energy, harmonic power, and residual content of the waveform. The function then returns six values: the fundamental amplitude and phase (`A1`), phase (`ph`), total harmonic distortion (`thd`), residual noise (`noise_rms`), and the amplitudes of the second and third harmonics (`A2, A3`) as key features of each specific waveform.

```
def lockin_features(v_fit, t_fit, f0):
    x, tt = v_fit.astype(float), t_fit.astype(float)
    dc = float(np.mean(x)); xz = x - dc; N = x.size

    def ap(freq):
        w = 2*np.pi*freq
        c = np.cos(w*tt); s = np.sin(w*tt)
        a = (2.0/N)*np.dot(xz, c)
        b = (2.0/N)*np.dot(xz, s)
        A = float(np.hypot(a, b))
        ph = float(np.arctan2(-b, a))
        return A, ph

    A1, ph = ap(f0); A2,_ = ap(2*f0); A3,_ = ap(3*f0)
    thd = (np.sqrt(A2**2 + A3**2)/A1) if A1>0 else 0.0

    ms_total = float(np.mean(x**2)); ms_dc = dc**2
    ms_tones = (A1**2 + A2**2 + A3**2)/2
    noise_rms = float(np.sqrt(max(ms_total - ms_dc - ms_tones, 0.0)))

    return A1, ph, thd, noise_rms, A2, A3
```

Cell 4: Feature extraction function, with projection onto a sinusoid, and returning the key features of the waveform for analysis.

The features selected are strongly linked to circuit specific behaviour. The amplitude and phase at the fundamental frequency represent the linear response of the amplifier to the input signal. Opens or shorts alter the loop gain and bias, producing measurable shifts in both, which makes

them good indicators of faults. The second and third harmonic amplitudes, and the total harmonic distortion, capture nonlinear behaviour. Faults that unbalance the circuit increase even harmonics, while clipping and saturation strengthen odd harmonics [5]. Taking only the 2nd and 3rd harmonic keeps the feature set compact while still providing a clear signature of nonlinearity. Finally, the residual noise root mean square quantifies whatever energy remains in the waveform after accounting for DC and the first three tones. This residual reflects higher order harmonics, simulator artefacts, or unstable operating points, all of which can increase when the circuit is driven near its limits but it's not as prominent in simulated circuits.

Overall, this feature function therefore produces a small but informative set of descriptors that make training more effective on the model. Although restricting the analysis to the third harmonic may miss extreme clipping cases, this design keeps the model lightweight, and the feature set can be extended in the future if broader robustness is required or if other circuit types and faults need to be considered.

c. Dataset assembly

This final function iterates over every class and frequency, and using the previously constructed functions, it extracts features for each Monte Carlo run, normalizes amplitudes by the known input (`vin_pk`), and optionally appends the excitation frequency (`f0`) as an additional feature. The result is a standard (`X,y`) dataset array ready for training and testing. The function is presented below:

```
def build_dataset(data_root=DATA_ROOT, classes=CLASSES, freqs=FREQS,
                  vin_pk=0.283, include_f0=True):
    rows, labels = [], []
    for cls in classes:
        for f0 in freqs:
            t, V = read_multirun_vout_csv(join(data_root, cls, f"{f0}.csv"))
            for k in range(V.shape[1]):
                t_fit, v_fit = standardise(t, V[:,k])
                A1, ph, thd, noise_rms, A2, A3 = lockin_features(v_fit, t_fit,
f0)
                feats = [A1/vin_pk, ph, thd, noise_rms/vin_pk, A2/vin_pk,
A3/vin_pk]
                if include_f0:
                    feats.append(float(f0))
                rows.append(feats); labels.append(cls)
    X = np.array(rows, float)
    y = np.array(labels, object)
    feat_names = ["A1_gain","phase","THD","noise_rel","A2_gain","A3_gain"] +
(["f0"] if include_f0 else [])
    return X, y, feat_names
```

Cell 5: Dataset building function incorporating external loop for CSV files with respective frequency f_0 and internal loop for the columns corresponding to the Monte Carlo 50 runs, then incorporation of the previously built

functions, `read_multirun_vout_csv()`, `sandardise()`, and `lockin_features()` to return `X` with the feature names with appropriate values, and `y` with the respective label class (healthy/short/open).

While the outer loop iterates through the fault type and frequencies CSV files, the inner loop processes each column of `V`, which contains the 50 Monte Carlo runs. For each run across `V[:, k]`, the pipeline applies `standardise()` and `lockin_features()`, then stores the resulting feature vector in `rows` and the corresponding class label in `labels`. After all iterations, `rows` is converted into the two-dimensional array `X` such that each corresponds to one run, and `labels` is converted into the one dimensional array `y`, aligned one-to-one with the rows of `X`. Finally the list `feat_names` specifies the order of the columns in `X` which has the features saved. Each row of `X` therefore contains a single feature vector for one Monte Carlo run, not all 50 runs together.

To clarify, normalisation is applied to `A1, A2, A3`, and `noise_rms`, dividing each by the known input amplitude. This ensures that the features represent relative gain and noise rather than raw voltages, removing trivial scaling effects in the linear regime and making runs directly comparable. Also, adding `f0` as a feature makes the dataset frequency aware, improving accuracy across the trained frequencies, while excluding it allows testing whether the features generalise across frequencies using grouped cross validation.

Finally, the order of the functions: standardisation, feature extraction, scaling, and optional frequency appending ensures that each feature vector is consistent and complete before being added to the dataset for training and testing. This design keeps feature extraction local to each run and produces a compact and structured dataset that is easy to extend with additional features or harmonics if needed with other circuit types, faults, and more frequencies if extended.

iii. Model Building

The features and labels prepared in the dataset are used to train a supervised classifier. The chosen model is a Random Forest (RF), (`from sklearn.ensemble import RandomForestClassifier`). This model was selected because it performs well on relatively small datasets, handles nonlinear feature interactions effectively, and provides interpretable outputs such as specific class probabilities for `healthy`, `short`, and `open`. In addition, Random Forests are known to be robust against overfitting when trained on compact, physically meaningful features, a property made possible here by the `lockin_features()` function [6].

The code below shows the complete model-building snippet, which includes the dataset creation, data splitting, training, and predictions:

```
X, y, feat_names = build_dataset()
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2, stratify=y,
       random_state=42)

rf = RandomForestClassifier(n_estimators=300, random_state=42, n_jobs=-1,
       class_weight="balanced_subsample")

rf.fit(Xtr, ytr)
y_rf_train_pred = rf.predict(Xtr)
y_rf_test_pred = rf.predict(Xte)
```

Cell 6: Model building including calling `build_dataset()`, splitting the train and test data 80/20, using the Random Forest, fitting, and making predictions for `Xtr and Xte`, technique adapted from Data Professor Bing Videos.

The dataset was first split into training and test subsets using `train_test_split` with `test_size=0.2`, producing an 80 to 20 split. This ratio is a standard splitting technique because it provides enough data to train the model (80% of the samples) while keeping a sufficiently large, unseen set (20%) to evaluate generalisation. The `stratify=y` argument ensures that the class distribution remains balanced across both subsets, avoiding bias toward any one fault type.

The Random Forest was then trained with 300 estimators and `class_weight="balanced_subsample"`, ensuring that all classes are weighted fairly, even if imbalances appear in resampling. These parameters were chosen for stability rather than aggressive tuning like other models, which was consistent with the project's goal of demonstrating a reproducible methodology rather than chasing accuracy gains. After training, the model and metadata like the features order and the class names, were saved (rf_model.joblib and meta.json) in a repo folder for the model artifacts (model_artifacts). These saved files allow the trained classifier to be loaded directly in the interactive demo and other projects without requiring retraining saving a lot of time.

iv. Model Results

Initially, the classifier model appears flawless as both train and test sets achieve 100% accuracy, with identical confusion matrices showing zero misclassifications. This reflects how clean and separable the simulated dataset is, especially when frequency information is included. However, when frequency is withheld during validation, accuracy drops, and the label-shuffle control collapses to chance level, highlighting that while the model performs strongly within scope, its perfect scores should not be overinterpreted.

a. Train and Test Performance (with f0)

Including the excitation frequency as a feature yielded the complete separation of the three classes. Both training and test sets achieved perfect accuracy, with precision, recall, and F1-scores of 1.00 across all classes. These scores were generated using the scikit-learn functions `classification_report` and `ConfusionMatrixDisplay` [7]:

Dataset	Accuracy	Precision	Recall	F-1 Score
Train	1.00	1.00	1.00	1.00
Test	1.00	1.00	1.00	1.00

Table 1: Train and test performance with f_0 included. Scores calculated using scikit-learn's `classification_report` (precision = positive predictive value, recall = sensitivity, F1 = harmonic mean of precision and recall).

The confusion matrices below confirm that every sample was classified correctly:

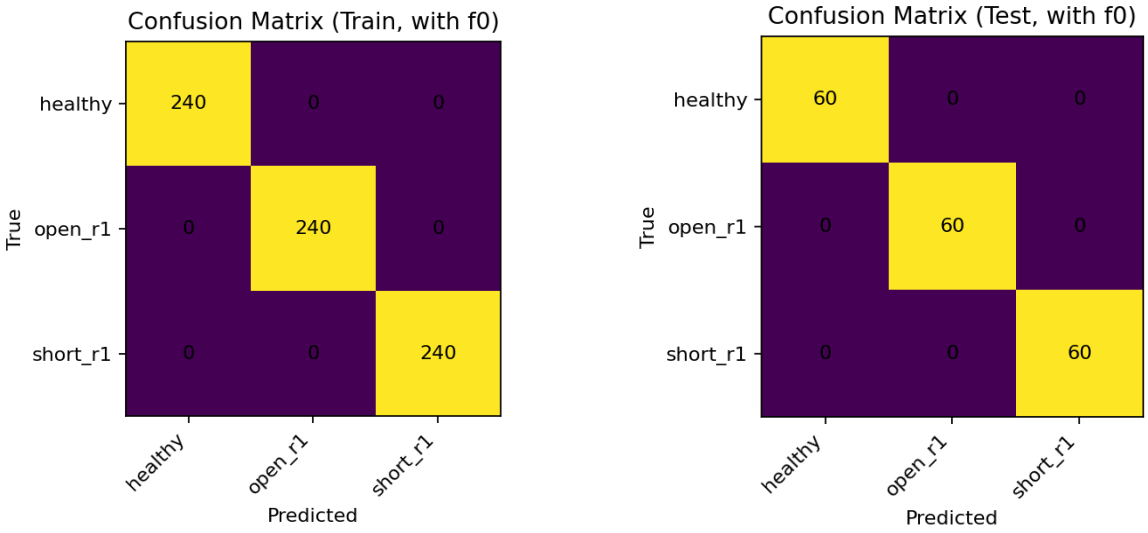

Figure 6: Confusion matrices for train and test with f_0 included showing the 'predicted' and 'true' axes with their correct classes.

The numbers 240 (train) and 60 (test) arise directly from the 80/20 split of the dataset: each class (`healthy`, `open`, `short`) contained 300 runs in total (6 frequencies × 50 Monte Carlo runs), giving 240 runs per each class in training and 60 per class in testing. The perfectly diagonal matrices show that all runs were assigned to their correct class, with zero misclassifications.

5-fold cross-validation further confirmed the consistency of this result. A 5-fold split was chosen as it is the standard compromise between computational cost and variance reduction as

the model is trained and validated on five different partitions, ensuring robustness across multiple splits rather than relying on a single train and test division [8].

Validation Method	Accuracy (mean ± std)
5-Fold Cross Validation Accuracy	1.0000 ± 0.0000

Table 2: 5-fold cross-validation accuracy with f_0 included. Reported using scikit-learn's `cross_val_score` with `cv=5`.

b. Frequency held out validation (without f0)

To test whether the model generalises beyond memorised frequencies, validation was repeated with f_0 excluded as a feature. Group K Fold was used, where all runs at one frequency are held out per fold:

Validation Method	Accuracy (mean ± std)
Group K Fold by f_0 Accuracy	0.9444 ± 0.1242

Table 3: Frequency-held-out validation without f_0. Accuracy reported using scikit-learn's `cross_val_score` with `GroupKFold` grouped by frequency.

This test is stricter than a simple random split: the model cannot "see" any runs from the held-out frequency during training. The high but imperfect score (0.9444) shows that the features generalise well across frequencies, but small overlaps in feature space exist, particularly at frequencies where amplifier responses converge. This indicates the model works as intended, but also that performance is not uniformly perfect when extrapolating across all frequencies so even more care should be taken in predictions made outside the 200-2000Hz trained range.

c. Label shuffle control

As a control experiment, the labels were randomly permuted before training. Under these conditions, classification accuracy dropped to chance level:

Control	Accuracy (mean ± std)
Label-shuffle Test Accuracy	0.2555

Table 4: Label shuffle control test. Accuracy collapses to chance level, confirming the true model exploits real structure in the data.

This confirms that the high performance observed in the real experiments reflects genuine relationships between extracted features and fault classes, rather than pipeline artefacts or accidental data leakage [9].

d. Train and Test Duplicate Check

Finally, a duplicate check confirmed that no identical feature vectors appeared across training and test sets.

Testing condition	Overlaps
Exact duplicates across test/train	0

Table 5: Duplicate check across train/test split. Confirmed using array comparison between `Xtr` and `Xte`.

This rules out the possibility that identical samples leaked between training and testing, which could otherwise have artificially inflated performance.

v. Conclusion and Model Limitations

At first, the results look almost too perfect, especially with all 1.00 scores when f_0 is included. This outcome is explained by the controlled, synthetic nature of the dataset in Altium, because the three circuit conditions (`healthy`, `open`, `short`) produce distinct harmonic and phase signatures, particularly once frequency information is explicitly provided. Under stricter conditions, such as frequency-held-out validation, accuracy drops. This demonstrates that the model is powerful but not perfect, and the boundaries between fault classes are not perfectly sharp.

Such imperfections are significant. They reflect the reality that real circuits will exhibit overlapping behaviours, random noise, component tolerances, and nonlinearities that are only partially captured in Altium simulations. Therefore, the near-perfect scores should be understood as an feature of the dataset's cleanliness rather than proof that the model will flawlessly diagnose faults.

a. Main Model Limitations

Several limitations arise from the design choices made in this project:

- Synthetic dataset scope: Only three fault types were simulated on one amplifier topology. Other realistic failure modes such as clipping, saturation, bias drift, or thermal effects were excluded due to time and storage constraints.

- Monte Carlo assumptions: Component tolerances were sampled with Gaussian distributions at 5%. While this introduces variability, it may not capture the full range of manufacturing or ageing effects in physical components.
- Feature set truncation: Features were extracted only at the fundamental and first two harmonics. Severe nonlinearities (like strong clipping) would generate higher-order harmonics that the model was not trained on.
- Perfect scores artefact: The 1.00 train and test metrics, while attractive, risk overstating performance. They primarily reflect the separability of the chosen fault types in simulation, not necessarily robustness to noisy real world data.

b. General Improvements and Future Work

To strengthen the approach, several improvements could be made. For instance, using broader datasets to expand beyond one amplifier topology and three fault modes to include more circuits, fault classes, and operating conditions would make the classifier more representative. Also, adding higher-order harmonics, crest factor, or time-domain statistics could help capture nonlinear faults missed by the current features.

Moreover, testing the model on measured lab data would provide essential evidence of real world performance and uncover differences between simulation and practice. Alternative ML models could be pursued and tested against each other. For example, comparing Random Forests against lightweight neural networks could highlight trade-offs in accuracy and scalability.

Despite these limitations, the project remains valuable. It demonstrates a complete, reproducible pipeline. From Altium based dataset generation to feature engineering, model training, and interactive demo. While the synthetic setting simplifies reality, the workflow is generalisable because engineers could readily extend it to larger datasets, different circuits, or new fault types. In that sense, this work provides a proof of concept for how machine learning can be applied to circuit fault detection.

IV. Appendix: User Guide for Running Demo

As an extra but useful feature, the project includes an interactive demonstration located in circuit_fault_detection/demo. This allows users to upload oscilloscope-like CSV files and receive a fault classification with probability outputs and waveform plots. The step by step user guide for running the demo is as follows:

i. Download a demo file

Navigate to the folder circuit_fault_detection/demo and click on either demo_file_1.csv, demo_file_2.csv, or demo_file_3.csv:

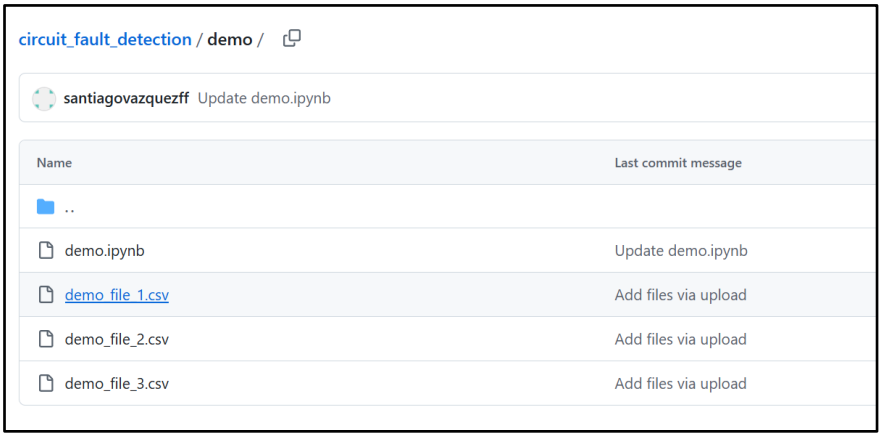

Figure 7: Demo instruction #1, location of CSV files in the demo section of the repository.

After opening the file tab on GitHub, click on the "Download raw file" icon:

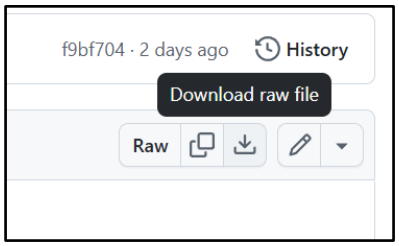

Figure 8: "Download raw file" button to access the sample CSV, to the upper right corner of the page.

The file will be saved into your browser's current "Downloads" folder. This file will be uploaded later on in the demo:

Figure 9: Downloaded sample CSV, this is what should appear in your "Downloads" browser window.

ii. Open the demo notebook

Return to circuit_fault_detection/demo and click on demo.ipynb:

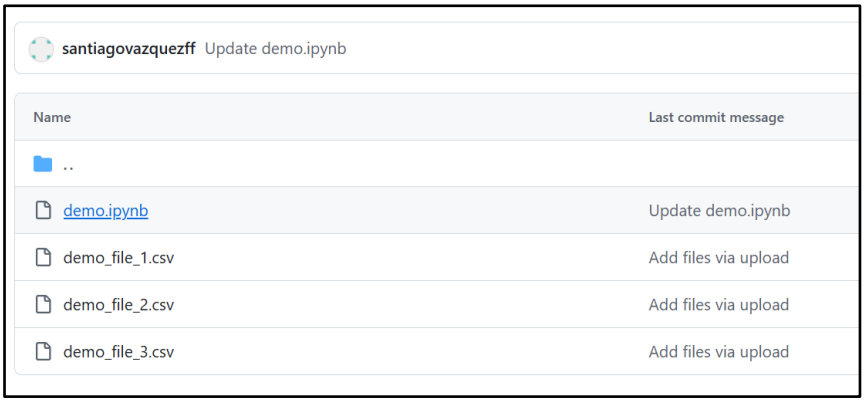

Figure 10: Location of the main demo program in the repository.

In the notebook preview window, select the blue "Open in Colab" button. This opens a temporary, runnable copy of the demo notebook in Google Colab:

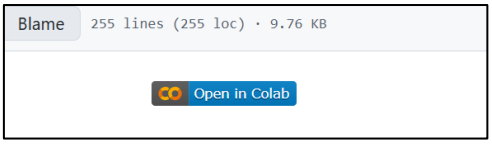

Figure 11: Button for opening the demo program in Colab, in the upper left corner of the preview window.

iii. Run the notebook

In Colab, click "Run" on the first code cell and authorise any connections when prompted. Successful execution is indicated by a green checkmark to the left of the cell:

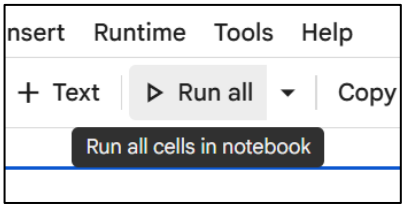

Figure 12: Button for running the demo program in Colab, in the upper central section of the notebook window.

iv. Upload the demo file

Scroll down to the User Inputs section. Use the "Choose Files" button to upload the CSV file you downloaded earlier:

```
Instructions for uploading demo files:

1. In the GitHub repo santiagovazquezff/circuit_fault_detection/demo/sample_files, select one of the CSV files included:

File dictionary:
{'demo_file_1': {'amplitude': 0.28, 'frequency': 200}, 'demo_file_2': {'amplitude': 0.38, 'frequency': 600}, 'demo_file_3': {'amplitude': 0.05, 'frequency': 2500}}

2. Click on the (`download raw`) file button in the GitHub window to download it

3. Upload the file in the box below

Choose Files  No file chosen        Cancel upload
```

Figure 13: Terminal window where dictionary, instructions, and uploads should take place after running the code.

Once the file upload is complete, enter the required input amplitude and frequency. These values are found in the dictionary provided in the command window under "File dictionary" (for example, demo_file_1 corresponds to amplitude = 0.28 and frequency = 200).

```
Enter input amplitude of waveform uploaded [consult dictionary below step 1 terminal window]: 0.280

Enter excitation frequency of waveform uploaded [consult dictionary below step 1 terminal window]: 200
```

Figure 14: Terminal window where user inputs the corresponding amplitude and frequency from the dictionary after uploading the file.

v. View prediction results

Continue scrolling to the "Results for data upload" section. The model will display the predicted class (healthy, open, or short), associated class probabilities, and a waveform plot of the uploaded run. This allows the user to visually confirm the prediction:

```
FINAL PREDICTION: healthy  (1/1 runs)

Class probabilities:
  healthy: 1.000
  open_r1: 0.000
  short_r1: 0.000

Plot of raw data for user to confirm prediction as either healthy, open, or short:
```

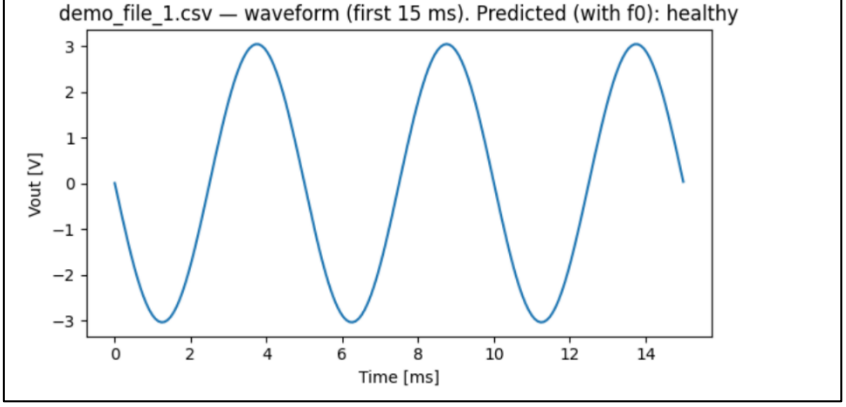

Figure 15: Terminal window displaying final results and predictions after successful running of the demo, these results effectively correspond the sample file 1 (`healthy`).

vi. Resetting and testing other files

If you wish to test another demo file, simply click “Run” again to reset the notebook. Repeat the same steps with the new file. Since Colab generates a temporary copy of the notebook each time, you do not need to worry about saving changes.

V. Reference List

[1] Texas Instruments. *LM324 Low Power Quad Operational Amplifiers Datasheet*. Dallas: Texas Instruments; 2015. Available from: https://www.ti.com/lit/ds/symlink/lm324.pdf

[2] Sedra AS, Smith KC. *Microelectronic Circuits*. 8th ed. New York: Oxford University Press; 2020.

[3] Altium. *Altium Designer User Manual*. San Diego: Altium LLC; 2023. Available from: https://www.altium.com/documentation

[4] Kundert K, Sangiovanni-Vincentelli A. *Simulation of Analog and Mixed-Signal Circuits*. Boston: Springer; 1990.

[5] Oppenheim AV, Schafer RW. *Discrete-Time Signal Processing*. 3rd ed. Upper Saddle River (NJ): Pearson; 2009.

[6] Breiman L. Random Forests. *Mach Learn*. 2001; 45(1):5–32.

[7] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine learning in Python. *J Mach Learn Res*. 2011; 12:2825–30.

[8] Hastie T, Tibshirani R, Friedman J. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed. New York: Springer; 2009.

[9] Ojala M, Garriga GC. Permutation tests for studying classifier performance. *J Mach Learn Res*. 2010;11:1833–63.