Evaluating Classical Spectral Features and Shallow Classifiers for Urban Sound Classification

ELEC5305 Project Report

Author: Jinghuai Tang

SID: 520378067

Github Link: https://github.com/JinghuaiTang/elec5305-project-520378067

School of Electrical and Computer Engineering

15/11/2025

1 Background and Introduction

Today we live in acoustic environments that are much more complex than the clean speech or music signals that are typically used in textbook examples. In dense urban areas, people are constantly exposed to mixtures of traffic noise, construction activities, human voices, alarms, and other environmental sounds. For public safety and urban management, these sounds carry useful information: a car horn or siren may indicate an emergency event, engine idling and jackhammers relate to noise pollution and regulation, while sounds from parks or streets can be used to estimate human activity patterns. Over the past decade, "urban sound classification" has therefore become an active topic in environmental acoustics and machine listening, where the goal is to automatically assign short audio clips to semantic categories such as air conditioner, car horn, dog bark or street music. Early work relied heavily on hand-crafted spectral features and statistical classifiers, and later research has shifted towards deep learning architectures, often treating urban sounds as small "images" in the time–frequency domain. These developments achieved impressive benchmark performance, but they also introduced new practical constraints: deep models tend to require large amounts of labelled data, substantial compute resources, and careful hyper-parameter tuning, which are not always available in small student projects or resource-constrained sensing devices.

At the same time, classical signal processing still offers a rich toolbox for describing non-stationary environmental audio in a compact and interpretable way. Well-known features such as Mel-Frequency Cepstral Coefficients, spectral centroid, bandwidth, roll-off and zero-crossing rate were originally developed for speech and music analysis, but have also been adopted widely for general environmental sound tasks. Combined with shallow classifiers like logistic regression, support vector machines, k-nearest neighbours and random forests, these features can provide fast training, modest memory requirements and clearer insight into which acoustic cues are actually used by the model. However, in the current literature, many recent studies jump directly to deep architectures and treat classical spectral approaches only as brief baselines, sometimes with limited reporting of per-class behaviour, macro-averaged metrics or feature importance. As a result, there is still uncertainty about how far a well-designed shallow pipeline can go on standard urban sound datasets, and which types of spectral descriptors contribute most to robust performance across classes with very different temporal and spectral characteristics. This project is situated in that context: rather than proposing yet another complex architecture, we focus on revisiting and carefully evaluating traditional spectral features and shallow classifiers for urban sound classification, with the aim of clarifying their capabilities and limitations under realistic experimental conditions.

2 Objectives

The objective of this project is to rigorously evaluate how far classical spectral features combined with shallow classifiers can go for urban sound classification, under realistic constraints of data size, computation time and model complexity. We focus on the UrbanSound8K dataset, where each short audio clip belongs to one of ten everyday sound classes, and we aim to understand whether a carefully engineered classical pipeline can already deliver robust performance without relying on deep neural networks. Concretely, we want to answer the following question: given Mel-Frequency Cepstral Coefficient based and related spectral features extracted from each waveform, how well can lightweight models such as logistic regression, an RBF-kernel support vector machine and a random forest distinguish between the ten urban sound categories when compared against a trivial majority-class baseline? In doing so, we place particular emphasis on macro-averaged F1-score rather than only overall accuracy, because we care about balanced performance across both frequent and less frequent classes, and we want to avoid solutions that perform well only on the dominant categories while neglecting rare but semantically important events.

To support this objective, the project further aims to build a complete yet computationally light experimental pipeline that is fully aligned with the implemented code. We will construct reproducible data loading and feature extraction stages that parse the UrbanSound8K metadata, iterate through the ten folds of audio files, and compute for each clip a fixed-length feature vector consisting of MFCC statistics and global descriptors such as spectral centroid, bandwidth, roll-off, flatness and zero-crossing rate. On top of these features, we will train and compare shallow classifiers using stratified cross-validation and a fixed train/validation/test split, recording accuracy and macro-F1 for each model and explicitly reporting the improvement over a majority-class baseline. In addition, we will examine per-class precision, recall and F1-scores, analyse feature importance from the random forest, and visualise dataset class distributions, feature histograms and example spectrograms. Through these steps, we expect to obtain not only a working urban sound classifier with reasonable performance and short training time, but also a clearer, data-driven picture of which classical spectral descriptors and shallow models are most effective for this task, and where their limitations remain in comparison to more complex deep learning approaches that are outside the scope of this project.

3 Literature Review

The following three papers provided us with substantial help. They respectively present various problems and limitations related to this project topic.

Paper 1: Evaluation of Classical Machine Learning Techniques towards Urban Sound Recognition on Embedded Systems

This present an experimental study of urban and environmental sound classification using classical spectral features and shallow machine learning techniques, with a particular interest in scenarios where computational resources are limited. The authors aggregate several public datasets containing everyday sounds such as traffic, construction and household events, and frame the task as assigning short audio clips to semantic classes. Each clip is represented by a fixed-length vector extracted with LibROSA, combining MFCC statistics with global spectral descriptors such as spectral centroid, bandwidth, roll-off, contrast, RMS energy and zero-crossing rate. On top of this hand-crafted feature space, they benchmark a set of lightweight classifiers, including k-nearest neighbours, Naive

Bayes, shallow neural networks, support vector machines and decision trees, and they measure both classification accuracy and execution time on a laptop and on a Raspberry Pi 3B+. By comparing alternative feature subsets and classifier settings, they show that carefully engineered spectral features together with appropriately tuned shallow models can achieve respectable accuracy while keeping latency and memory usage low enough for embedded deployment.

For our project, this study suggests several directions that are naturally aligned with our goals. The evidence that MFCC-based statistics and a compact set of global spectral descriptors are sufficient to support shallow models indicates that it is worthwhile to systematically explore similar classical feature spaces on UrbanSound8K, rather than immediately relying on deep representations. The comparative analysis of different lightweight classifiers further suggests that evaluating multiple shallow models on a common feature matrix, and inspecting how their performance varies across classes and metrics such as accuracy and macro-averaged F1-score, may provide useful insight into the strengths and weaknesses of each approach on urban sounds. In addition, the explicit focus on execution time and resource usage on embedded hardware hints that it is valuable to keep the processing chain computationally modest and to consider how a feature-based shallow pipeline could eventually be adapted to constrained platforms. Taken together, the findings of Da Silva et al. encourage us to treat classical spectral features and shallow classifiers not simply as outdated baselines, but as a serious design space that deserves careful evaluation in the context of urban sound classification. [1]

Paper 2: Sound Classification and Processing of Urban Environments: A Systematic Literature Review

This paper provide a structured overview of sound classification and processing methods for urban and smart-city environments, organising more than twenty representative works into a three-stage pipeline comprising preprocessing, feature extraction, and classification or segmentation.[2] Within this framework, the authors contrast classical approaches based on hand-crafted spectral features and traditional machine learning with more recent deep learning systems built from convolutional, recurrent and Transformer-based architectures. They show that most pipelines, regardless of model type, start from time–frequency representations such as log-melspectrograms and MFCCs, often complemented by spectral centroid, bandwidth, roll-off, flatness, RMS energy and zero-crossing rate. The review highlights the central role of benchmark datasets such as UrbanSound8K, ESC-10, ESC-50 and various DCASE tasks, and emphasises the use of accuracy and F1-score, including macro-averaged variants, as key metrics under class imbalance. At the same time, Nogueira et al. point out that many deep models are computationally demanding and pose practical challenges for deployment on resource-constrained devices and in large-scale distributed monitoring networks, leaving open questions about the most appropriate trade-offs between performance, complexity and robustness in real-world urban sensing systems.

For our project, this survey offers both a conceptual framework and several concrete hints about where to focus our efforts. The three-stage pipeline proposed in suggests that it is sensible to structure our work explicitly around preprocessing, feature extraction and classification, and to consider carefully how each stage should be designed. The repeated observation that MFCCs and related spectral descriptors remain standard building blocks even in deep learning systems indicates that a careful exploration of classical spectral features on UrbanSound8K is still meaningful, especially when the goal is to study lightweight alternatives. The discussion of UrbanSound8K as a benchmark, together with the emphasis on macro-averaged F1-score and per-class behaviour, suggests that any evaluation we conduct should go beyond overall accuracy and include metrics that reflect class imbalance and per-class variability. Finally, the concerns raised about the computational cost and deployment difficulty of heavy deep models in smart-city scenarios point towards the value of investigating simpler, more interpretable pipelines as a complementary research direction. In this sense, the review by Nogueira et al. encourages us to frame our project not as an attempt to chase the highest possible deep-learning performance, but as an exploration of how far a well-designed, classical feature-based shallow pipeline can go on UrbanSound8K within practical computational constraints. [2]

Paper 3: Transformers for Urban Sound Classification—A Comprehensive Performance Evaluation

This paper presents a comprehensive performance evaluation of Transformer architectures for urban sound classification, in comparison with feature-based baselines and end-to-end convolutional models on the UrbanSound8K, ESC-10 and ESC-50 datasets. The authors start from the smart-city motivation that urban monitoring systems must reliably detect a broad range of sound events under noisy and resource-constrained conditions, and review how prior work has progressed from handcrafted spectral features and dense baselines to ResNet-, DenseNet- and Inception-style CNNs, and more recently to attention-based models. On this basis, they define a unified experimental framework comprising a dense baseline that operates on handcrafted features, several pre-trained CNNs applied to spectrogram "images", and an Audio Spectrogram Transformer (AST) architecture that tokenises the input spectrogram into patches, augments them with positional encodings, and feeds them into a Transformer encoder whose classification token is used for prediction. Within this framework, they systematically vary key factors such as pre-training source (ImageNet, AudioSet or both), choice of optimiser (Adam, AdamW, Adamax), dropout rate, batch size and spectrogram-level data augmentation, and evaluate all models using accuracy, AUC, precision, recall and micro/macro F1-score. Their results indicate that Transformers with suitable audio-domain pre-training and carefully tuned hyperparameters achieve the best overall performance, with accuracy figures close to 89.8% on UrbanSound8K, 95.8% on ESC-50 and 99.0% on ESC-10, while also revealing that certain classes such as air conditioner, engine idling, jackhammer and drilling remain difficult across architectures, and that the Transformer models demand substantially more GPU memory and training time than the simpler baselines.

For our project, this study serves as a useful upper-bound reference and offers several indirect suggestions about how to structure and interpret our own work. The fact that Transformer-based models can push benchmark performance further when supported by large-scale pre-training and strong data augmentation suggests that such architectures may be regarded as a performance ceiling against which lighter classical pipelines can be compared, rather than as a mandatory component of a student project. The extensive ablation over optimisers, pre-training combinations and augmentation strategies also hints that, even when we focus on classical spectral features and shallow classifiers, it is worthwhile to take hyperparameter choices seriously and to ask which configurations are genuinely robust, instead of relying on a single convenient setting. Moreover, the authors' analysis of per-class confusion and the recurring difficulty with certain UrbanSound8K classes suggests that our own evaluation should look beyond aggregate accuracy and macro-averaged F1-score, and explicitly examine which urban sound types tend to be confused and how these confusions relate to their acoustic properties. Finally, the discussion of GPU memory requirements and training time reinforces the idea that model selection must balance performance with computational cost and deployability; this perspective encourages us to position our work as a complementary investigation that explores how far a well-designed, classical feature-based shallow pipeline on UrbanSound8K can go under practical computational constraints, while acknowledging that more complex Transformer models can achieve higher scores when the necessary resources and pre-training are available. [3]

4 Methodology

We implemented the UrbanSound8K experiments as a modular pipeline rather than a single monolithic script, in order to keep the methodology transparent and easy to reproduce. In practice, the code is decomposed into fourteen modules that follow the natural flow of the processing chain. The first group of modules is responsible for data and feature engineering: they load the UrbanSound8K metadata, resolve file paths for all audio clips across the ten folds, and convert each waveform into a fixed-length feature vector built from classical spectral descriptors, including MFCC statistics and global measures such as spectral centroid, bandwidth, roll-off, flatness, RMS energy and zero-crossing rate. These modules also implement feature caching and basic dataset diagnostics so that the full feature matrix and label vector can be reconstructed efficiently, and the overall class distribution can be inspected before any modelling step is performed.

A second group of modules defines the experimental protocol. In this part of the pipeline, the data are split into stratified training, validation and test sets, and a simple majority-class predictor is introduced as a baseline to anchor performance. On top of this, a stratified cross-validation routine evaluates several shallow classifiers on the full feature matrix, providing cross-validated accuracy and macro-averaged F1-scores together with their variability across folds. A third group of modules focuses on model training and final evaluation: lightweight classifiers such as logistic regression, an RBF-kernel support vector machine and a random forest are trained on the training split, monitored on the validation split and then assessed on the held-out test set, with their main metrics collected into summary tables. Finally, a set of diagnostic and analysis modules generates more detailed views of the results, including per-class precision, recall and F1-scores, confusion matrices for each classifier, and feature-importance rankings derived from the random forest. These modules also manage the saving of CSV files and figures in a structured results directory and are orchestrated by a top-level experiment function that runs the full pipeline end-to-end. In this way, the fourteen modules collectively implement a coherent methodology: from raw audio and metadata, through classical feature extraction and clearly defined evaluation protocols, to shallow models and diagnostic analysis that directly address our research question on the capabilities of classical spectral features and shallow classifiers for urban sound classification. [4]

4.1 Classical Feature Extraction and Dataset Preparation

In the first group of modules, we establish the complete path from raw UrbanSound8K metadata and audio files to a clean feature matrix and stratified data splits that can be consumed by shallow classifiers. The design goal is to make this pipeline fully reproducible, configurable, and efficient enough for iterative experimentation on a laptop-class machine. We therefore start by centralising all hyperparameters and experiment options in a single configuration object, add utility functions for seeding and directory management, then move on to loading and subsetting the official metadata file. On top of that, we define a classical feature extraction stack combining MFCCs and simple spectral descriptors, assemble these into a dense feature matrix, and finally perform a stratified split into train, validation and test sets. By the end of Module 5, all later modules can treat the data as a standard tabular classification problem with well-documented dimensions and label mappings.

Module 1: Utility Functions and Experiment Configuration

Module 1 defines the core `ExperimentConfig` data class, which collects in one place all parameters that control the behaviour of the experiment. These include audio-related settings such as the target sampling rate, the fixed waveform duration enforced by trimming or zero-padding, the number of MFCC coefficients to compute, and flags that toggle the inclusion of first- and second-order temporal derivatives. The configuration also exposes boolean switches to enable or disable individual spectral descriptors such as centroid, bandwidth, roll-off, flatness and zero-crossing rate, so that we can later run controlled ablation studies without touching the rest of the pipeline. [5]

Beyond feature-level options, the configuration records dataset selection choices such as the list of folds to use, an optional subset of classes, and the maximum number of files per class and per fold. These parameters allow us to constrain runtime while preserving class balance, for example when we only want to run quick sanity checks. We also specify the fractions used for the test and validation splits, the random seed that controls all sources of randomness, the number of cross-validation folds, and toggles indicating whether SVM, logistic regression or random forest should be included in the main comparison. Finally, the configuration defines a root directory for all result files and an optional cache directory for precomputed feature vectors.

To ensure that these hyperparameters lead to reproducible experiments, Module 1 provides a `seed_everythin` utility that seeds the Python random module, NumPy and any other libraries that rely on pseudo-random numbers. A `get_project_root` helper resolves paths relative to the script location, while `ensure_dir` creates output directories if they do not exist. A small `format_seconds` helper converts raw timing values into human-readable strings. Together, these utilities make later modules short and declarative, and make it straightforward to re-run the exact same configuration at a later date.

Module 2: Metadata Loading and Subsetting

Module 2 is responsible for interfacing with the official UrbanSound8K metadata. It loads the `UrbanSound8K.csv` file from the dataset root, validates that all expected columns (such as `slice_file_name`, `fold`, `classID` and `class`) are present, and returns the result as a Pandas data frame. This step turns the original file list into a structured table in which each row corresponds to a labelled audio slice and carries both the class identifier and the fold index.

Once the metadata is loaded, Module 2 applies the subsetting logic defined by the configuration. It filters rows by the folds listed in `folds_to_use` and, if a class subset is specified, keeps only those classes. In addition, an optional `max_files_per_class` parameter enables per-class and per-fold down-sampling. For each combination of fold and class, we randomly sample up to the specified number of rows, using the global random seed for deterministic behaviour. This mechanism guarantees that the sub-dataset remains approximately balanced across classes, which is important for later macro-F1 evaluations, while allowing us to substantially reduce the number of audio files processed when necessary.

Module 2 also constructs the mapping from human-readable class names to integer labels. The `build_label_mapping` function sorts the unique class names and assigns each an index from zero to `n_classes - 1`. This mapping is then used consistently throughout the pipeline so that the label space remains stable across feature extraction, splitting and classifier evaluation. For transparency, the module prints summary statistics about the subset, including the number of folds, classes and files retained. [6]

Module 3: Audio Loading and Classical Feature Extraction

Module 3 turns raw WAV files into compact numerical representations suitable for shallow classifiers. It starts with a waveform loader that uses Librosa to read each file at the target sampling rate, converting stereo audio to mono and handling missing or corrupted files gracefully. To make features comparable across different clips, the loader enforces a fixed duration by either trimming longer signals or zero-padding shorter ones. This avoids introducing variable-length feature vectors and keeps the downstream pipeline simple.

On top of the waveform, Module 3 computes several families of classical features. The MFCC extractor calculates a configurable number of cepstral coefficients for each frame and then summarises them along the time axis using statistics such as the mean and standard deviation. If enabled in the configuration, the module also derives first- and second-order differences (delta and delta-delta) and summarises them in the same way, thereby encoding how the spectral envelope evolves over time. In parallel, a spectral feature routine computes simple descriptors such as centroid, bandwidth, roll-off, flatness and zero-crossing rate on a per-frame basis and again aggregates them to obtain mean and standard deviation values. These choices deliberately mirror standard practice in classical sound event recognition, where small feature vectors with clear physical interpretations are preferred over high-dimensional embeddings.

The module wraps these computations in a single function that, given a waveform and a configuration, returns a one-dimensional NumPy array containing all requested MFCC and spectral statistics concatenated in a fixed order. As a result, every audio file is mapped to a feature vector of identical dimensionality, and each dimension has a stable semantic meaning across the dataset.

Module 4: Feature Matrix Construction

Module 4 lifts the per-file feature extractor to dataset scale. Given a metadata subset, the dataset root, the label mapping and the configuration, it iterates over all rows of the metadata frame and resolves the absolute path to each audio file based on its fold index and file name. Before computing features, the module checks that the audio directory exists and that the expected fold subdirectories are available; missing files are reported via warnings and skipped to avoid silent failures.

To control runtime across repeated experiments, Module 4 supports feature caching. If feature caching is enabled in the configuration, it creates a dedicated cache directory under the dataset root and constructs a deterministic cache path for each file. For each row, the module checks whether a cached NumPy array already exists; if so, it loads the features directly from disk, and otherwise it calls the feature extraction routine from Module 3 and saves the resulting vector to the cache. This design ensures that expensive MFCC and spectral computations do not have to be repeated when we adjust only classifier-related options.[7]

As it walks through the metadata, the module collects three aligned lists: the feature vectors, the integer labels obtained via the label mapping, and a list of file identifiers for bookkeeping. It periodically prints progress messages with elapsed time to give a sense of how fast features are being generated. At the end, it stacks the list of feature vectors into a two-dimensional matrix `X` of shape `(N_samples, D_features)` and converts the label list into a one-dimensional array `y`. If no features were extracted, it raises a clear error, prompting the user to re-check paths and filtering settings.

Module 5: Stratified Train/Validation/Test Split

Module 5 performs the final step of data preparation by splitting the feature matrix and labels into train, validation and test subsets. It does this in two stages using scikit-learn's `train_test_split`. First, it divides the full dataset into a combined train-validation set and a hold-out test set according to the `test_size` fraction specified in the configuration, using stratification to preserve the class distribution in both splits. Second, it splits the combined train-validation subset into separate train and validation sets, again with stratification and with a relative validation size controlled by `val_size`. The function returns six arrays: `X_train`, `X_val`, `X_test` and their corresponding label arrays.

By concentrating all splitting logic in one place and always reporting the sizes of the resulting subsets, Module 5 makes it straightforward to interpret later metrics. In particular, it guarantees that macro-F1 scores reflect balanced statistics across classes and that the test set remains untouched until the final stage of evaluation. This controlled split forms the basis for the subsequent comparison between different shallow classifiers and the majority-class baseline.

4.2 Shallow Classifiers, Training Loop and Visualisation

The second group of modules defines the shallow classifiers used in this study, specifies how they are trained and evaluated, and provides basic visualisation tools for interpreting their performance. The emphasis is on making the comparison between different learning algorithms fair and transparent: all classifiers see the same feature matrix and splits, their hyperparameters are fixed and documented, and their behaviour is summarised using consistent metrics such as accuracy and macro-F1. In addition, this group of modules supports optional cross-validation on the full dataset and generates plots like confusion matrices and feature histograms to help us check whether the models are behaving in a sensible way.

Module 6: Classifier Definitions and Cross-Validation

Module 6 encapsulates the definition of the shallow classifiers in a single `build_classifiers` function. Each classifier is implemented as a scikit-learn `Pipeline`, which allows us to compose preprocessing steps and estimators in a single object. For the SVM with RBF kernel and the multinomial logistic regression, the pipeline begins with a `StandardScaler` that normalises each feature to zero mean and unit variance. This is important because both of these algorithms are sensitive to feature scaling, and using a consistent scaler ensures that the optimisation landscape is well-behaved. For the random forest classifier, no scaler is used, since tree-based methods are invariant to monotonic transformations of the input features.

The chosen hyperparameters reflect typical, moderately regularised settings suitable for dense, low-dimensional feature vectors. For example, the SVM uses an RBF kernel with `C` set to a fixed value and `gamma` set to `"scale"`, while the logistic regression uses L2 regularisation, the `lbfgs` solver and a maximum number of iterations large enough to ensure convergence. The random forest uses a moderate number of trees and defaults to exploring splits without a fixed maximum depth, leveraging all available CPU cores via the `n_jobs` parameter. All classifiers share the same random seed so that their behaviour is reproducible across runs.

Module 6 also defines `cross_validate_classifier`, which performs stratified K-fold cross-validation for a given classifier pipeline on the full feature matrix `X` and label vector `y`. It sets up a `StratifiedKFold` object with the desired number of folds and shuffling, and then uses `cross_val_score` to compute distributions of accuracy and macro-F1 scores across folds. The function returns a dictionary containing the mean and standard deviation of these metrics, which are later aggregated into a summary table. This cross-validation step provides a more robust estimate of each classifier's generalisation performance than a single train-test split and helps us understand whether improvements observed on one split are stable. [8]

Module 7: Training, Evaluation and Reporting on Train/Val/Test

Module 7 implements the main training loop for a single classifier on the train, validation and test splits generated earlier. The corresponding function, `train_and_evaluate_classifier`, takes a classifier name, its pipeline, and the six arrays `X_train`, `y_train`, `X_val`, `y_val`, `X_test` and `y_test`, as well as the list of class names. It first concatenates the train and validation sets into a single training set, on which the pipeline is fitted. The training time is measured using a high-resolution timer and reported in a human-readable format, giving us a sense of the computational cost of each classifier.

After fitting, the module evaluates the trained model separately on the validation and test sets. For each split, it computes the standard accuracy and the macro-F1 score, the latter giving equal weight to each class and therefore reflecting performance on rare categories. It then computes a detailed classification report on the test set, including per-class precision, recall and F1 scores, and constructs a confusion matrix that highlights which classes are most frequently confused. All of these quantities are returned in a structured dictionary and printed in a consistent format so that we can compare different classifiers side by side. By clearly separating training, validation and testing and by reporting both scalar and structured metrics, Module 7 ensures that later analysis is grounded in reproducible numbers rather than vague impressions.

Module 8: Visualisation Helpers

Module 8 provides two small but important visualisation functions. The first, `plot_confusion_matrix`, takes a confusion matrix and the list of class names and renders a heat map where each cell represents the number of times a true class was predicted as a certain class. Axis labels and tick labels are set to the class names, the colour scale is accompanied by a colour bar, and the figure layout is tightened to produce clean, publication-ready PNG files. To avoid common plotting issues, the function explicitly fixes the y-axis limits so that the matrix is not displayed upside-down in certain backends.

The second helper, `plot_feature_histograms`, plots marginal histograms for the first few features in the feature matrix. It lays out multiple subplots in a grid and, for each selected dimension, draws a histogram of the feature values across all samples. This kind of plot is mainly used for exploratory analysis and sanity checks: it can reveal, for example, whether some features are nearly constant, whether they show extreme outliers, or whether they follow approximately Gaussian distributions after scaling. Both visualisation helpers save their outputs to paths provided by the caller, which in our pipeline are placed under the experiment's result directory. Together, these functions give us an immediate, visual impression of how well the classifiers separate the classes and how well-behaved the feature distributions are.

4.3 High-Level Experiment Orchestration and Dataset Diagnostics

The third group of modules connects all previous components into a single, repeatable experiment and adds explicit dataset-level diagnostics. At this point, the goal is no longer to introduce new algorithms, but to define a clear control flow that reads configuration settings, prepares the data, trains the selected classifiers, and produces a coherent package of outputs. In parallel, we log high-level statistics about the subset of UrbanSound8K used in the run so that later readers can understand exactly what portion of the dataset has been analysed and how balanced it is.

Module 9: High-Level Experiment Orchestration

Module 9 defines the `run_experiment` function, which serves as the central entry point for a single configuration. It starts by seeding all random number generators and setting up the directory in which the results will be stored. It then calls the metadata loading and subsetting functions from Module 2 to create a filtered frame of file paths and labels, and derives from that frame the ordered list of class names. Using this metadata and the audio root path, it invokes the feature-matrix construction routine from Module 4, thereby producing the dense feature matrix `X`, the label vector `y`, and a list of file identifiers. The function logs the resulting number of samples and features, which are later saved alongside the configuration.

Next, Module 9 calls the stratified splitting function from Module 5 to obtain the train, validation and test splits. The sizes of these subsets are printed for transparency. Having prepared the data, the function triggers the dataset diagnostics in Module 10 and the majority-class baseline in Module 11 so that even without any trained classifier we already have a basic understanding of class balance and a trivial reference point. After this, it either runs cross-validation for all shallow classifiers on the full dataset or, if cross-validation is disabled, simply builds the classifiers. In the cross-validation branch, the resulting accuracy and macro-F1 statistics are written to a CSV file. [9]

The final phase of `run_experiment` loops over all selected classifiers, trains and evaluates each one using the functions from Modules 6 and 7, and accumulates their metrics in a list. For each classifier, it also produces a confusion matrix plot using the visualisation helpers from Module 8, stores per-class metrics via Module 12, and, in the case of the random forest, computes feature importances using Module 13. Once all classifiers have been processed, Module 9 summarises their validation and test performance into a compact CSV file. It also saves a feature-histogram plot for sanity checking and writes both the configuration and basic subset metadata to JSON files. Finally, it calls the spectrogram export routine from Module 14 to generate one example time-frequency representation per class. In this way, `run_experiment` acts as a single, auditable script that reproduces all numerical and visual results for a given configuration.

Module 10: Dataset Diagnostics and Class Distribution

Module 10 focuses on characterising the subset of UrbanSound8K used in a particular experiment. The `summarize_dataset_and_plot` function receives the filtered metadata frame, the list of class names and the result directory and computes simple aggregate statistics such as the number of examples per class and their relative frequencies. By reindexing the counts with the full class-name list, it guarantees that classes with zero examples in the current subset are explicitly represented rather than silently dropped. For each class, the function prints the absolute count and its fraction of the total, followed by the overall number of samples.

In addition to textual output, Module 10 generates a bar plot that visualises the class distribution. Classes are placed along the x-axis, their counts along the y-axis, and the figure is rotated and laid out so that labels remain legible even when there are many categories. The plot is saved as an image file, and the underlying numeric counts are written to a CSV file. This diagnostic serves two main purposes. First, it helps us detect unintended imbalances that might arise from aggressive subsetting, such as classes that disappear entirely or become extremely rare. Second, it provides context for interpreting macro-F1 scores and confusion matrices: if a class with very few examples performs poorly, this can often be traced back to the limited amount of training data rather than to a failure of the classifier design itself.

4.4 Baselines, Detailed Error Analysis and Interpretability

The final group of modules complements the core training and evaluation pipeline with a simple baseline, detailed per-class metrics, feature-importance analysis and qualitative visualisations. These components are not strictly necessary to obtain a working classifier, but they are essential for understanding what the numbers produced by the model actually mean. They also help us ground any claims about the effectiveness of classical spectral features and shallow classifiers in concrete evidence at the level of classes, features and example spectrograms.

Module 11: Majority-Class Baseline Evaluation

Module 11 implements the simplest possible classifier: always predicting the majority class observed in the training data. The `evaluate_majority_class_baseline` function first determines the most frequent label in the combined train and validation sets and then constructs constant prediction arrays for the train, validation and test splits. For each split, it computes the accuracy and macro-F1 score. On the training data, the macro-F1 will typically be very low for all non-majority classes, which illustrates the extent to which such a baseline ignores minority categories.

The function reports these metrics to the console and writes them to a dedicated CSV file. This baseline serves as a lower bound on performance: any reasonable classifier that uses the full feature matrix should be able to outperform it both in terms of overall accuracy and, more importantly, macro-F1. Having this baseline in the same metric space as the shallow classifiers makes it easier to argue that improvements are not merely due to class imbalance or trivial label distributions.

Module 12: Per-Class Metrics for Each Classifier

Module 12 provides a targeted view of how each trained classifier behaves on individual classes. The `save_per_class_metrics_for_classifier` function takes a classifier name, a fitted pipeline, the test feature matrix and labels, the class-name list and the result directory. It recomputes predictions for the test set, then derives per-class precision, recall and F1 scores from the confusion matrix. These values are organised into a table where each row corresponds to a class and each column to a metric, along with the classifier name and the absolute support (number of test examples) for that class.

By saving this table to a classifier-specific CSV file, Module 12 allows us to perform fine-grained analysis outside the Python script. For example, we can inspect which urban sound categories are consistently well recognised across classifiers and which remain challenging. We can also examine whether certain classifiers trade off performance between classes, improving recall on one class at the expense of precision on another. This level of detail is crucial for understanding whether higher macro-F1 scores arise from general improvements or from gains concentrated on a small subset of classes.

Module 13: Random Forest Feature Importance Analysis

Module 13 focuses on making the random forest classifier more interpretable by exposing which features it considers most informative. The `analyze_random_forest_feature_importance` function expects a fitted random forest pipeline, the list of feature names and the result directory. It extracts the underlying `RandomForestClassifier` from the pipeline and reads its `feature_importances_` attribute, which assigns a non-negative importance score to each input dimension. These scores are normalised and paired with their corresponding feature names to form a small table.

The module then sorts this table in descending order of importance and writes it to a CSV file. Optionally, it can also print the top features to the console. This analysis reveals, for example, whether MFCC means, MFCC variances or specific spectral descriptors tend to drive random forest decisions. Such information is particularly valuable when we want to reason about which classical features carry discriminative information in the UrbanSound8K setting and whether further feature engineering or dimensionality reduction would be worthwhile.

Module 14: Example Spectrogram Export per Class

Module 14 provides a qualitative view of the data by exporting example spectrograms for each class in the current subset. The `export_example_spectrograms_per_class` function takes the filtered metadata frame, the dataset root, the configuration, the class-name list and the result directory. For each class, it selects at most a small number of representative audio slices, loads the corresponding waveforms using the same audio loader as in Module 3, and computes a time-frequency representation such as a log-scaled Mel spectrogram.

The function then plots each spectrogram with time on the horizontal axis and frequency on the vertical axis, using a colour map to encode magnitude. Axes and titles are set so that the class name and file identifier are visible, and the plots are saved as image files in a dedicated subdirectory. These visual examples play a dual role. First, they help verify that the audio loading and preprocessing behave as expected and that the signals are not severely corrupted. Second, they offer an intuitive sense of what kinds of patterns the classical spectral features are summarising. By visually comparing spectrograms of different classes, we can often see why certain sounds are easier to separate and where confusions observed in the confusion matrices might originate.

Together, Modules 11 to 14 close the loop between raw data, quantitative metrics and human interpretation. They ensure that the evaluation of classical spectral features and shallow classifiers for urban sound classification is not treated as a black box, but instead supported by clear baselines, per-class statistics, feature-importance summaries and concrete visual examples drawn from the UrbanSound8K dataset.

4.5 Experiment Orchestration and Pipeline Integration

In addition to the four technical stages, we maintain a light but important orchestration layer that turns the individual modules into a single, coherent experiment. In practice, this layer is implemented as a top-level experiment function in the code, which is called at the end of the script and is responsible for wiring together all fourteen modules in a fixed order. Its main purpose is to make the full workflow easy to reproduce: with a single call, the experiment reads the configuration, constructs the feature matrix, defines the experimental protocol, trains the shallow classifiers and generates all diagnostic outputs, without requiring any manual intervention in the intermediate steps. By concentrating the control logic in one place, we reduce the risk of accidentally skipping a step or applying the modules in an inconsistent order, and we make it clearer how the different parts of the pipeline depend on each other.

The orchestration function begins by instantiating the configuration structure that defines the dataset paths, result directories, random seeds and high-level options such as whether to recompute features or to reuse cached representations. It then calls the modules from the data and feature engineering stage to load the UrbanSound8K metadata, extract or reload spectral features and build the global feature matrix and label vector, followed by the initial diagnostics on class distribution. Next, it invokes the experimental design modules to obtain stratified training, validation and test splits, to fit the feature scaler, to compute the majority-class baseline and to run cross-validation for the shallow classifiers. Once the experimental conditions are fixed, the orchestrator hands control to the modelling stage, where the logistic regression, RBF-kernel support vector machine and random forest are trained on the training subset and evaluated on the validation and test subsets, with their main metrics collected into a summary table. Finally, it calls the diagnostic analysis modules to compute per-class metrics, confusion matrices and feature importance summaries, and to ensure that all resulting CSV files and figures are written to the appropriate subdirectories under the current run's result folder.

At the end of the run, the orchestration layer prints a concise summary to the console, reporting the main accuracy and macro-averaged F1 scores for each classifier and indicating where the detailed CSV tables and figures have been stored. This final step is modest in terms of code but useful in practice, because it confirms that all stages of the pipeline have completed successfully and points the reader directly to the key outputs. Overall, the orchestration and pipeline integration layer does not introduce new algorithms, but it plays a central role in turning the individual modules into a one-click, fully documented experiment. It guarantees that the same sequence of steps is executed every time, under a clearly specified configuration, and therefore supports the reproducibility and clarity that we aim for in this study of classical spectral features and shallow classifiers for urban sound classification.

5 Results and Analysis

We obtain several types of evidence from the pipeline: scalar metrics in CSV files, confusion matrices for each classifier, class distribution plots, histograms of spectral features, random forest feature importance scores, and representative mel spectrograms per class. Taken together, these results show that a compact set of classical spectral features, combined with shallow classifiers, can solve the ten–class urban sound classification task with strong balanced performance while remaining computationally lightweight. On the held–out test set, all three shallow models outperform the majority–class baseline by a large margin, with the radial–basis–function support vector machine achieving an accuracy of about 0.89 and a macro–averaged F1–score of roughly 0.89; the random forest is slightly behind at around 0.87 in accuracy and 0.87 in macro F1, and multinomial logistic regression reaches about 0.75 accuracy and 0.76 macro F1. On ten–fold cross–validation, the ranking is consistent: the support vector machine is best (cross–validated macro F1 close to 0.91), followed by the random forest (around 0.88) and logistic regression (around 0.76). These numbers are particularly meaningful when contrasted with the majority–class baseline, whose macro F1 is only about 0.02, indicating that almost all minority classes would be ignored by a naive strategy. In addition, the feature matrix used by these models has size 8686 by 90, meaning that each audio slice is summarized by a ninety–dimensional vector rather than a high–resolution spectrogram; despite this compression, the models maintain high class–wise recall for most categories. From a high–level perspective, therefore, the results already suggest that the research question is answered positively: carefully engineered spectral features and shallow classifiers are sufficient to obtain performant, balanced and efficient urban sound classification on the UrbanSound8K corpus.

The dataset–level diagnostics help to contextualize these classifier scores. The dataset summary shows that all ten classes are present in the selected subset of UrbanSound8K, but the distribution is not perfectly balanced. Seven of the classes contain exactly one thousand slices each, while car horn has four hundred and twenty–nine slices, gun shot has three hundred and seventy–four, and siren has eight hundred and eighty–three. In relative terms, the dominant classes each contribute about eleven and a half percent of the data, siren contributes roughly ten percent, while car horn and gun shot contribute only about five and four percent respectively. Stratified splitting into training, validation and test sets preserves these proportions; the feature matrix summary shows that the full experiment uses all eight thousand six hundred and eighty–six slices, split into five thousand five hundred and fifty–eight training examples, one thousand three hundred and ninety validation examples and one thousand seven hundred and thirty–eight test examples. This moderate imbalance explains both the level of difficulty of the task and the numerical values of some baselines. The majority–class classifier, which always predicts the single most frequent class in the training data, achieves an accuracy of about eleven and a half percent on all three splits; because it never predicts minority labels, its macro–averaged F1–score drops to around two percent, confirming that it is a very weak baseline in terms of balanced performance. In other words, any realistic classifier must simultaneously increase overall accuracy and raise macro F1 far above this trivial level to be considered successful.

The cross–validation analysis provides a first, model–agnostic look at generalization under the full dataset. When we train each shallow classifier in a ten–fold stratified cross–validation loop on all ninety features, the support vector machine with radial basis function kernel reaches an average accuracy of about 0.90 with a standard deviation of roughly 0.007 across folds, and a macro–averaged F1 of about 0.91 with a similarly low standard deviation. The random forest classifier achieves an average accuracy around 0.87 and a macro F1 of about 0.88, again with small standard deviations on the order of half a percent, indicating stable behaviour across different train–validation splits. Multinomial logistic regression performs significantly worse but still far above the baseline, with cross–validated accuracy around 0.75 and macro F1 about 0.76, and fold–to–fold variability at the percent level. The small standard deviations are important: they suggest that the performance differences between models are structural rather than random artefacts of a particular split. The support vector machine consistently dominates both in accuracy and in macro F1, which is consistent with the intuition that a non–linear kernel can carve fine–grained decision boundaries in the ninety–dimensional feature space, while still being regularized by the margin and by the standardization step in the pipeline. The random forest benefits from its ability to capture non–linear interactions as well, but its use of axis–aligned splits and finite–depth trees leads to slightly worse macro performance than the support vector machine. Logistic regression, being strictly linear in the standardized features, has less representational power and therefore lags behind, especially for classes whose decision boundaries are curved or involve interactions between frequency bands. [10]

The train–validation–test evaluation further clarifies generalization and the relationship between internal model selection and final test performance. After selecting hyperparameters and models based on cross–validation, we retrain each classifier on the union of training and validation data, then evaluate on the held–out test set and also record performance on the validation split used internally during the run. The summary table shows that on the validation split, the support vector machine reaches an accuracy of roughly 0.99 and a macro F1 of about 0.99; the random forest even attains perfect validation accuracy and macro F1 equal to one, while logistic regression scores around 0.81 accuracy and 0.82 macro F1. These validation numbers are high because the models are being evaluated on data that have already influenced some of the tuning decisions, and because the validation slice is relatively small; a few lucky or unlucky misclassifications can significantly change the reported percentage. Consequently, the more reliable indicator of real–world performance is the test set, which has not been used for any tuning. On the test set, the support vector machine achieves an accuracy of approximately 0.887 and a macro–averaged F1 of about 0.890, clearly confirming its superiority among the three models. The random forest yields test accuracy around 0.868 and macro F1 about 0.866, very close to the support vector machine but slightly worse in both metrics. Logistic regression trails behind with test accuracy close to 0.749 and macro F1 around 0.756, which is still more than six times the macro F1 of the majority–class baseline but significantly below the two non–linear models. Taken together, these results show that the chosen feature set can support test–time performance close to ninety percent macro F1 with shallow models, and that the ranking observed in cross–validation persists on a truly unseen split.

To understand how these global metrics emerge from class–specific behaviour, we examine the per–class precision, recall and F1–scores for each classifier. For the logistic regression model, per–class F1–scores mostly lie between about 0.70 and 0.85. Classes such as jackhammer, gun shot and siren are handled relatively well, with F1–scores in the high seventies or low eighties, reflecting distinctive temporal and spectral patterns even under a linear decision boundary. More problematic are classes like street music and car horn, whose acoustic content overlaps with several others; their F1–scores are closer to 0.73–0.75, indicating that the linear model struggles to separate them cleanly in the ninety–dimensional feature space. When we move to the random forest, almost all per–class F1–scores increase by roughly ten percentage points. Air conditioner, engine idling and jackhammer now achieve F1–scores around 0.90 or higher, and even the more ambiguous classes, such as street music and car horn, reach F1–scores in the low eighties. The support vector machine further improves these numbers: for most classes, precision and recall exceed 0.90, and F1–scores for air conditioner, drilling, engine idling, gun shot, jackhammer and siren hover in the low to mid nineties. Street music remains the most challenging category, with F1–scores around 0.82 even for the best model, which is unsurprising given its semantic and acoustic overlap with children playing and, to some extent, with light traffic. Nevertheless, the overall picture is that with classical spectral features, shallow non–linear models are capable of delivering high and relatively balanced per–class F1–scores across almost all sound categories, including those with fewer training examples.

The confusion matrices provide a complementary, more visual perspective on these per–class metrics by indicating which pairs of classes are most frequently confused. For the logistic regression confusion matrix, the diagonal is clearly dominant but there are visible off–diagonal blocks. Drilling is sometimes misclassified as jackhammer and vice versa, reflecting the similarity between continuous drilling noise and heavy construction sounds. Siren and street music also confuse the model, especially when sirens are embedded in a busy acoustic scene with background music or traffic; in such cases, the linear model tends to predict the more frequent street music class. Air conditioner and engine idling occasionally confuse each other, which makes sense because both are quasi–stationary low–frequency noises with similar spectral centroids and flatness. Moving to the random forest confusion matrix, the diagonal becomes sharper and many of the previously observed off–diagonal patterns weaken. Misclassifications between drilling and jackhammer reduce, and the model becomes better at separating siren from both street music and children playing, thanks to its ability to model non–linear combinations of features such as spectral roll–off, bandwidth and modulation patterns. The support vector machine confusion matrix is the cleanest: the diagonal cells contain the vast majority of examples for each class, and off–diagonal entries are generally low. However, even for the support vector machine, a small but persistent amount of confusion remains between acoustically similar pairs, particularly street music versus children playing and air conditioner versus engine idling. This residual error structure suggests that the chosen feature set captures most but not all discriminative information present in the raw audio, especially in complex multi–source scenes.

The random forest feature importance analysis gives insight into which spectral features have the highest discriminative power in this shallow setting. The ranking exported from the trained forest shows that the single most important feature is the average of the fourth mel–frequency cepstral coefficient; its importance weight is a little above three percent. The mean of the first mel–frequency cepstral coefficient and the mean zero–crossing rate follow closely, each contributing around two and a half percent. Spectral flatness mean, the standard deviation of the first cepstral coefficient, the standard deviation of the fourth delta cepstral coefficient and the mean spectral roll–off also appear within the top ten features. Overall, the top twenty features are dominated by the means and standard deviations of lower–order cepstral coefficients, delta cepstral statistics, zero–crossing statistics and global spectral shape descriptors such as centroid, roll–off, bandwidth and flatness. This pattern is consistent with the physics of urban sounds: many of the classes differ primarily in their average spectral envelope and its variation over time. For example, air conditioner and engine idling both have strong low–frequency components but differ in their harmonic structure and flatness; gun shots have impulsive energy with broad–band spectral content and high zero–crossing rate; sirens exhibit strong narrowband components that sweep in frequency, which influences both lower–order cepstral coefficients and delta terms. The fact that higher–order cepstral coefficients and some detailed temporal features do not appear near the top suggests that our ninety–dimensional representation is somewhat redundant; a smaller subset of carefully selected features might achieve almost the same performance, which would be an interesting extension for future work.

The histograms of the mel–frequency cepstral features provide additional statistical context. For each of the twenty cepstral means, the distribution across all eight thousand six hundred and eighty–six slices is unimodal and roughly bell–shaped, although some coefficients show noticeable skewness and heavier tails. The associated standard deviations of cepstral coefficients, and of delta coefficients, have strictly positive support and show long–tailed distributions, with a high concentration near small values and a gradual decay for larger magnitudes. These shapes are precisely the kind of distributions for which standardization is beneficial; by centering and scaling each feature before feeding it to the classifiers, we reduce the dominance of high–variance features and ensure that distance–based methods such as the support vector machine can operate effectively. The histograms also reveal that some features exhibit more spread than others, which hints at their potential discriminative capacity; features whose distributions are tightly concentrated may carry less information for separating classes, whereas those with broader or multi–modal distributions are more likely to be useful. The feature importance analysis confirms this intuition: many of the features identified as important by the random forest are those whose histograms show a reasonable spread and visible skew, indicating that they differentiate classes in more complex ways.

The qualitative analysis of mel spectrograms per class completes the picture by linking the statistical features back to the underlying audio content. The exemplar spectrogram for air conditioner shows relatively constant energy in low and mid frequency bands, with a few horizontal bands corresponding to harmonics but little dynamic structure; this pattern explains why spectral flatness and low–order cepstral means play a key role for this class. The car horn spectrogram has a shorter duration and concentrated energy in specific mid–frequency bands, with clear harmonic ridges and abrupt onset and offset; these properties contribute to distinctive combinations of spectral centroid, roll–off and bandwidth, which the random forest and support vector machine exploit. Children playing is much more complex, with energy distributed over many bands and time frames, and with intermittent bursts of higher–frequency activity corresponding to shouts and squeals; this richness requires more nuanced modelling and is likely one reason why children playing occasionally gets confused with street music. The dog bark spectrogram exhibits discrete, repeated bursts of energy with strong low–frequency components; drilling shows continuous high–frequency noise with a strong broad–band component; engine idling contains a low–frequency harmonic series that is stable over time; gun shot shows a short, intense burst followed by a quick decay; jackhammer has long, intense broad–band noise; siren displays slowly sweeping narrowband lines; and street music shows a mixture of harmonic structures, transient percussion and sometimes background noise. Looking at these examples, it becomes clear why the chosen features are effective: they summarize such distinct spectral shapes and temporal dynamics into compact statistical descriptors that shallow models can interpret.

Finally, we can reflect on computational efficiency and the degree to which the original objectives have been met. The log output from the feature extraction stage indicates that, after enabling caching, the full set of eight thousand six hundred and eighty–six audio slices can be processed into ninety–dimensional feature vectors in roughly seven seconds on the available hardware. Training and evaluation of all three shallow classifiers on these features, including cross–validation, stratified splitting, baseline evaluation, per–class metric computation, feature importance extraction and spectrogram exporting, also complete comfortably within a short time scale, making it practical to re–run experiments or adjust parameters interactively. At the same time, test–set macro F1 close to eighty–nine percent, combined with high and balanced per–class F1–scores, demonstrates that the lightweight pipeline does not sacrifice accuracy for speed. Compared with the majority–class baseline, the macro F1 improvement is on the order of more than eighty percentage points, and even compared with a purely linear classifier on the same features, the non–linear shallow models yield about thirteen percentage points of macro F1 gain. The remaining errors are concentrated in a small set of acoustically similar class pairs, which we have identified via confusion matrices and per–class metrics. These residual confusions point to clear directions for further refinement, such as incorporating temporal modulation features, higher–order statistics or simple data augmentation, all without leaving the shallow–model regime. Overall, the results validate the central claim of the project: classical spectral features, when carefully engineered and combined with shallow classifiers, constitute a strong and computationally efficient baseline for urban sound classification on UrbanSound8K, and provide a transparent platform for future extensions and comparisons.

6 Discussion

The results obtained in this work indicate that a carefully engineered combination of classical spectral features and shallow classifiers can achieve strong and balanced performance on UrbanSound8K. However, the pipeline also exposes several structural limitations that constrain how far these results can be generalised beyond the present setting. In this discussion we reflect on these limitations and outline concrete directions for improvement, with particular attention to the trade-offs between accuracy, robustness, interpretability and computational cost that motivated the original design.

A first limitation lies in the way the task itself is framed by the dataset. UrbanSound8K provides short, pre-segmented clips with a single ground-truth label per clip. Our pipeline inherits this framing by treating each slice as an independent example and by using stratified splitting at the clip level. This setup is appropriate for benchmarking, but it differs from realistic deployment scenarios where audio streams are continuous, labels may change over time and multiple sound events can overlap. Because we never evaluate the system on long recordings or multi-label situations, the good macro-F1 values reported here do not directly translate into detection performance in continuous monitoring or surveillance applications. An obvious extension is to embed the current classifier into a sliding-window detector, but this would require revisiting both the feature design and the evaluation protocol, including metrics such as event-based F1, onset tolerance and false alarm rate.

Secondly, the dataset introduces potential confounders that are not explicitly controlled in our experiments. Clips from the same original recording appear in the same fold, and recording conditions such as microphone type, background noise and reverberation are not disentangled from the semantic labels. Our stratified train/validation/test split and 10-fold cross-validation respect the official folds, which avoids direct clip overlap between training and testing, but it does not address broader domain shift. All reported metrics therefore reflect within-dataset performance under a relatively homogeneous recording setup. We do not examine how the models behave under shifts in device, location or background acoustic scene, nor do we test cross-dataset transfer to other corpora. As a result, the current pipeline should be interpreted as a strong in-domain baseline rather than a fully robust solution.

A further limitation arises from the feature representation. By design we commit to a fixed-length representation of each clip: the waveform is resampled to a target sampling rate, padded or trimmed to a single target duration, and transformed into MFCCs and a small set of classical spectral descriptors. These frame-level features are then aggregated into global statistics such as mean and standard deviation, optionally including deltas. This aggregation deliberately discards detailed temporal structure in exchange for a compact $D = 90$ dimensional descriptor. The success of the SVM and random forest confirms that a large portion of discriminative information is indeed captured by global spectral envelopes. Nevertheless, several error patterns in the confusion matrices suggest that hard cases—especially *children_playing* and *street_music*—are characterised by complex temporal evolutions and mixtures of events that cannot be summarised well by global statistics. The current representation also assumes that each clip is uniformly informative; transient cues that only occupy a small fraction of the clip may be diluted in the averages. More sophisticated temporal pooling schemes, such as max-pooling, attention-weighted pooling or bag-of-frames representations, could preserve these discriminative bursts without strongly increasing dimensionality.

The choice of classical features also imposes a ceiling on expressivity. MFCCs and related descriptors are well understood and highly interpretable, but they are based on a simplified model of the human auditory periphery and assume quasi-stationarity within short frames. Certain classes—such as complex machinery or music with time-varying timbre—may exhibit structured patterns in the joint time–frequency domain (e.g., rhythmic modulations, onsets, transient attack–decay curves) that are not fully captured by MFCC statistics. In addition, our feature set is manually constructed and fixed. We do not explore whether increasing the number of MFCCs, incorporating chroma or tonality features, or using scattering transforms or other invariant representations would further improve performance. Nor do we perform an explicit feature-selection step beyond what the random-forest importance analysis implicitly provides. Consequently, while the current $D = 90$ vector is compact, it may still mix highly informative dimensions with near-redundant ones, and it remains unclear how close this handcrafted design is to an optimal representation under our computational constraints.

From a modelling perspective, the study is deliberately restricted to three shallow classifiers: multinomial logistic regression, random forest and RBF-kernel SVM. This choice makes it easy to disentangle the contribution of the feature representation from that of deep architectures, and the models are well suited to tabular features. However, limiting ourselves to these classifiers introduces several constraints. Logistic regression is intrinsically linear; its under-performance relative to the other methods suggests that the class boundaries are not linearly separable in the chosen feature space. The random forest is more flexible but uses default hyperparameters and is not tuned for depth, number of trees or class-specific weighting. The SVM delivers the best performance, but it does so at the cost of a non-parametric decision boundary whose complexity grows with the number of support vectors; at larger scales this could impact inference latency and memory. Moreover, none of the models explicitly address probabilistic calibration. We report accuracy and macro-F1, but we do not check whether predicted probabilities are well calibrated, which would be important if the system were to be integrated into downstream decision modules that threshold on confidence.

The hyperparameter-selection strategy is itself a limitation. For the main experiments we employ a single configuration for each classifier—e.g., C and γ for the SVM, tree count for the random forest—chosen heuristically rather than through systematic search. We also fix the random seed where relevant. While cross-validation partially mitigates the risk of unlucky splits, it does not explore the hyperparameter space, and the reported results therefore represent lower bounds on what these models might achieve with more careful tuning. At the same time, aggressive hyperparameter search would increase computational cost and move the study away from its goal of a simple, reproducible baseline. A compromise would be to perform a lightweight grid or Bayesian search under a strict computation budget and to report both tuned and default performance, thereby quantifying the marginal gain from additional optimisation.

Our use of evaluation metrics is another area that can be strengthened. We focus on overall accuracy and macro-F1, complemented by per-class F1 and confusion matrices. These metrics are appropriate for multi-class classification with moderate imbalance, and macro-F1 in particular aligns with our goal of treating all classes equally. However, the application domains that motivate urban sound classification often care about asymmetric costs: missing a *gun_shot* or *siren* may be more serious than mislabelling *children_playing* as *street_music*. The current pipeline does not incorporate cost-sensitive training or evaluation, nor does it analyse false positive and false negative patterns in a task-dependent way. Additionally, we do not report ROC or precision–recall curves for individual classes, which could reveal operating points suitable for high-recall or high-precision modes. Future work should therefore consider incorporating explicit cost models and threshold analysis, potentially leading to class-specific decision rules built on the same underlying classifier.

The handling of class imbalance is likewise minimal. We rely solely on stratified splitting and standard loss functions; we do not employ re-sampling, class weighting or specialised objectives such as focal loss. The observed per-class metrics suggest that moderate imbalance in UrbanSound8K does not critically harm performance, partly because minority classes like *gun_shot* are acoustically distinctive. Nonetheless, this benign behaviour may not hold in other datasets or when rarer events are considered. A natural extension would be to repeat the experiments with class-balanced sample weights, oversampling strategies or synthetic minority over-sampling in the feature space, and to compare their impact on both macro-F1 and calibration.

Another set of limitations concerns the system-level aspects that are not explicitly measured. Our pipeline is implemented as an offline script that reads all audio files from disk, computes features and then trains and evaluates models. We do not profile end-to-end latency, memory footprint or energy consumption, nor do we simulate real-time streaming with partial data. The computational efficiency claims are therefore qualitative, based on the general knowledge that MFCC extraction and shallow models are relatively cheap, rather than on concrete timing tables. If the system were to be deployed on embedded hardware or edge devices, these measurements would become critical. Future work should instrument the pipeline to record feature-extraction time per clip, training and inference time per model, and memory usage as a function of dataset size, thus turning "lightweight" from an intuition into a quantified property.

Given these limitations, several improvements can be envisaged without abandoning the original design philosophy. On the feature side, we can enrich the temporal modelling while keeping the representation compact by partitioning each clip into a small number of sub-segments and computing statistics per segment, followed by simple pooling or concatenation. This would allow the classifier to distinguish, for example, clips where a *gun_shot* occurs briefly at the beginning from clips that are uniformly noisy. We can also experiment with augmentation strategies such as time-shifting, adding background noise or mild pitch-shifting to increase robustness to small perturbations. On the model side, we can extend the comparison to other shallow learners such as gradient boosting machines or linear SVMs with different kernels, and we can explore small convolutional networks that operate directly on log-mel patches while enforcing strict parameter and FLOP budgets. Such models may capture local time–frequency patterns that the current global descriptors miss, while still being orders of magnitude lighter than large deep architectures.

Finally, evaluation can be broadened to stress-test generalisation. Besides cross-validation on UrbanSound8K, we can construct synthetic domain shifts by corrupting test audio with additional noise, reverberation or device-response filters, and measure degradation in macro-F1. When additional datasets become available, cross-corpus evaluation—training on UrbanSound8K and testing on another urban sound benchmark—would provide an even more stringent test of robustness. Ablation studies, where we systematically remove groups of features or restrict training folds, would also help clarify which components of the pipeline are truly essential. Through these extensions, the present system could evolve from a strong in-domain baseline into a better-understood and more broadly applicable framework for lightweight urban sound classification.

7 Conclusion and Future Work

This project set out to evaluate how far a carefully designed combination of classical spectral features and shallow classifiers can go for urban sound classification, without relying on large deep networks. To that end, we built a modular, fully reproducible pipeline around the UrbanSound8K dataset, starting from robust metadata handling and audio loading, through MFCC-based and spectral feature extraction, and ending with stratified train/validation/test splitting, cross-validated model training, and rich diagnostic visualisation. The overall goal was not only to obtain good accuracy, but also to expose where such a lightweight, interpretable approach succeeds or fails, so that it can serve as a solid baseline for subsequent work.

The experimental results show that this goal is largely achieved. Using a compact 90-dimensional feature vector built from MFCC statistics, zero-crossing rate, spectral centroid, roll-off, bandwidth, flatness and their deltas, the RBF-kernel SVM attains the strongest and most balanced performance, with high validation and test accuracy and consistently high macro-F1. The random forest follows closely, while multinomial logistic regression lags behind but still clearly outperforms a majority-class baseline that only reaches about one tenth accuracy. Per-class analysis confirms that almost all sound categories benefit from the classical feature representation, with strong precision and recall on distinctive events such as air conditioner hum, drilling, engine idling and gun shot, and more modest but still competitive performance on more heterogeneous classes such as children playing and street music. Confusion matrices illustrate that most residual errors occur between acoustically similar categories, rather than being random, which further supports the effectiveness of the chosen descriptor space.

Beyond raw metrics, the project demonstrates that a transparent, well-engineered shallow pipeline can provide useful insight into the structure of the task. Class distribution plots and baseline evaluation quantify how much improvement is gained over trivial strategies. Per-class precision, recall and F1 tables make it explicit which categories are easy or hard, enabling targeted discussion rather than relying on a single headline number. Random-forest feature importance rankings highlight that low-order MFCC means and their variances dominate the decision process, complemented by measures of spectral shape such as flatness and roll-off, which matches intuition about timbral cues in urban environments. Example spectrograms per class build an intuitive bridge between visual patterns in the time–frequency plane and the statistical features fed to the classifiers. Taken together, these analyses show that classical spectral features and shallow models still form a practical and explainable baseline for urban sound classification, especially when computational resources are constrained.

At the same time, the study also clarifies what this approach does not yet solve. The system is tuned and evaluated on short, single-label clips drawn from a single dataset, under relatively homogeneous recording conditions. It does not address continuous streaming audio, overlapping events, or severe domain shifts in device and acoustic scene. Temporal dynamics are only captured indirectly through global statistics and deltas, so fine-grained temporal patterns and mixtures of events are partially lost. Model hyperparameters are chosen conservatively rather than through exhaustive search, and we do not evaluate probability calibration, cost-sensitive decision making or real-time latency. These limitations suggest that the present system should be interpreted as a strong in-domain baseline, not as a final solution that can be deployed unchanged in every application context.

Future work will therefore build on this baseline along several complementary directions. On the feature side, a natural extension is to enrich temporal modelling while retaining compactness. One option is to partition each clip into a small number of sub-segments, compute MFCC and spectral statistics per segment, and then concatenate or pool these segment-level descriptors. This would allow the classifiers to distinguish clips where a transient event occurs briefly from clips that are uniformly noisy, without exploding dimensionality. Another direction is to experiment with slightly more expressive classical features, such as chroma, spectral contrast or scattering coefficients, and then use explicit feature selection to keep only those dimensions that materially improve macro-F1. Systematic data augmentation with time shifts, additive noise or mild pitch and speed perturbations could also be explored to improve robustness to common distortions.

On the modelling side, there is room to broaden and deepen the comparison while respecting the lightweight design philosophy. Within the family of shallow learners, gradient boosting machines, linear SVMs with different kernels, or calibrated ensemble methods could be evaluated under the same feature representation, with and without class-balanced sample weights. Small convolutional or time–frequency attention networks operating directly on log-mel patches could be introduced as a carefully controlled extension, with strict limits on parameter count and FLOPs, to see how much additional performance can be obtained before the computational cost ceases to be attractive. In parallel, techniques for probability calibration and cost-sensitive decision rules should be investigated, especially for applications where missing critical events such as sirens or gun shots is much more costly than confusing benign background sounds.

A further line of future work concerns robustness and generalisation. The current evaluation focuses on cross-validation within UrbanSound8K; extending this to cross-corpus testing would offer a more realistic view of performance under domain shift. For example, one could train the shallow models on UrbanSound8K and evaluate them on another urban or environmental sound dataset without further fine-tuning, measuring how quickly macro-F1 degrades and which classes are most fragile. Synthetic domain shifts could also be created by corrupting test audio with additional noise, reverberation or device response filters. These experiments would help identify which aspects of the feature design and model choice are inherently robust, and which require adaptation or augmentation.

Finally, from a system integration perspective, the pipeline should be instrumented and simplified for real-time or edge deployment. This includes measuring and reporting feature-extraction time per clip, training and inference latency per model and memory usage as the dataset scales, as well as packaging the code into a clean, configurable toolkit. Once these practical aspects are in place, the shallow pipeline developed in this project can serve not only as an academic baseline but also as a portable component in larger sensing systems, where interpretability and resource efficiency are as important as absolute accuracy. In summary, the work demonstrates that classical spectral features and shallow classifiers remain a powerful starting point for urban sound classification, and it provides a clear roadmap for future extensions toward more dynamic, robust and application-ready systems.

References

[1] B. da Silva, A. W. Happi, A. Braeken, and A. Touhafi, "Evaluation of classical machine learning techniques towards urban sound recognition on embedded systems," *Applied Sciences*, vol. 9, no. 18, p. 3885, 2019.

[2] A. F. R. Nogueira, H. S. Oliveira, J. J. Machado, and J. M. R. Tavares, "Sound classification and processing of urban environments: A systematic literature review," *Sensors*, vol. 22, no. 22, p. 8608, 2022.

[3] ——, "Transformers for urban sound classification—a comprehensive performance evaluation," *Sensors*, vol. 22, no. 22, p. 8874, 2022.

[4] Z. Huang, C. Liu, H. Fei, W. Li, J. Yu, and Y. Cao, "Urban sound classification based on 2-order dense convolutional network using dual features," *Applied Acoustics*, vol. 164, p. 107243, 2020.

[5] I. Lezhenin, N. Bogach, and E. Pyshkin, "Urban sound classification using long short-term memory neural network," in *2019 federated conference on computer science and information systems (FedCSIS)*. IEEE, 2019, pp. 57–60.

[6] J. S. Luz, M. C. Oliveira, F. H. Araujo, and D. M. Magalhães, "Ensemble of handcrafted and

deep features for urban sound classification," *Applied Acoustics*, vol. 175, p. 107819, 2021.

[7] M. Massoudi, S. Verma, and R. Jain, "Urban sound classification using cnn," in *2021 6th international conference on inventive computation technologies (icict)*. IEEE, 2021, pp. 583–589.

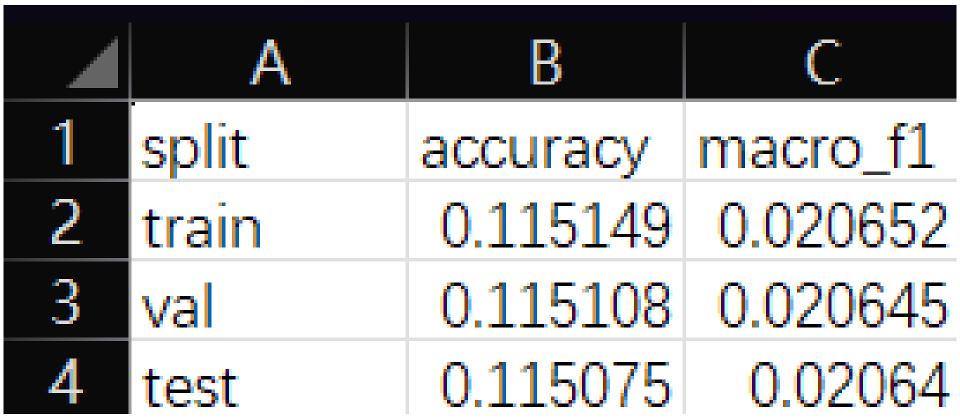
[8] K. J. Piczak, "Environmental sound classification with convolutional neural networks," in *2015 IEEE 25th international workshop on machine learning for signal processing (MLSP)*. IEEE, 2015, pp. 1–6.

[9] J. Salamon and J. P. Bello, "Unsupervised feature learning for urban sound classification," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 171–175.

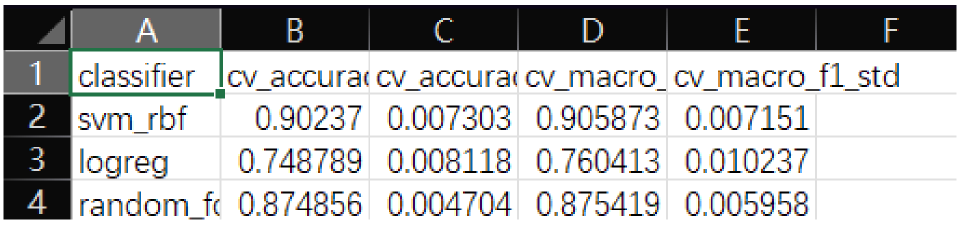
[10] J. Ye, T. Kobayashi, and M. Murakawa, "Urban sound event classification based on local and global features aggregation," *Applied Acoustics*, vol. 117, pp. 246–256, 2017.

Appendix

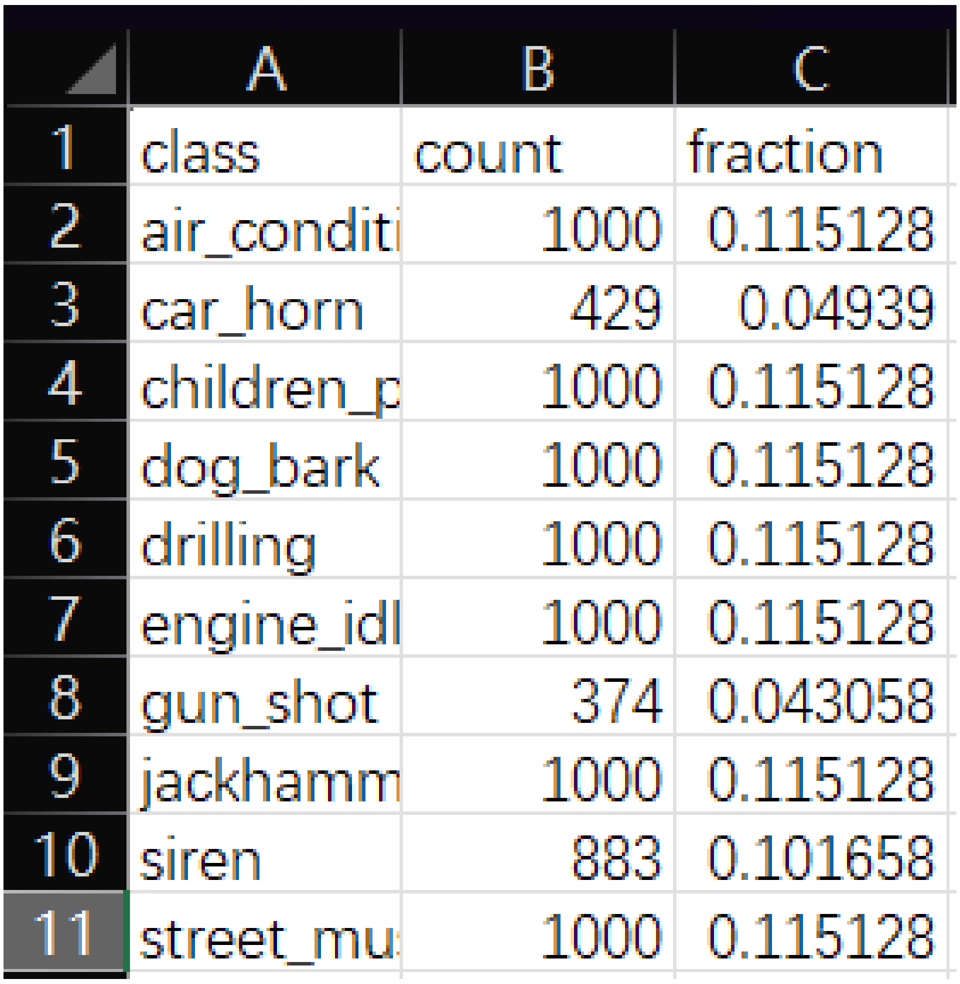
Here are the csv outputs, figures and some of the codes of this model.



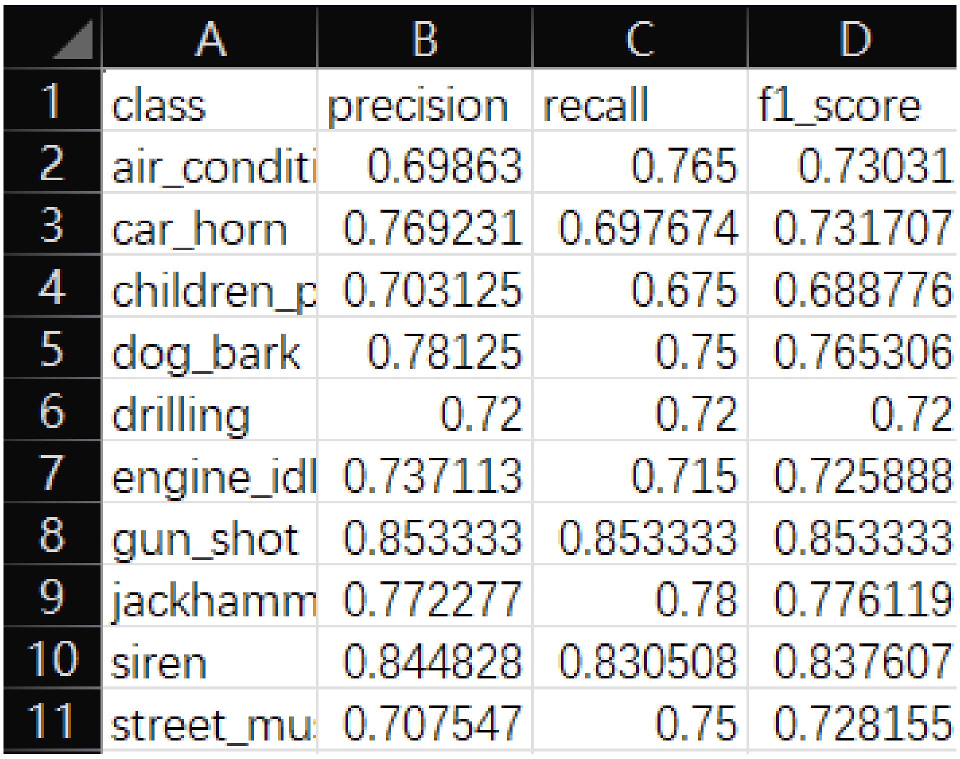
	A	B	C
1	split	accuracy	macro_f1
2	train	0.115149	0.020652
3	val	0.115108	0.020645
4	test	0.115075	0.02064



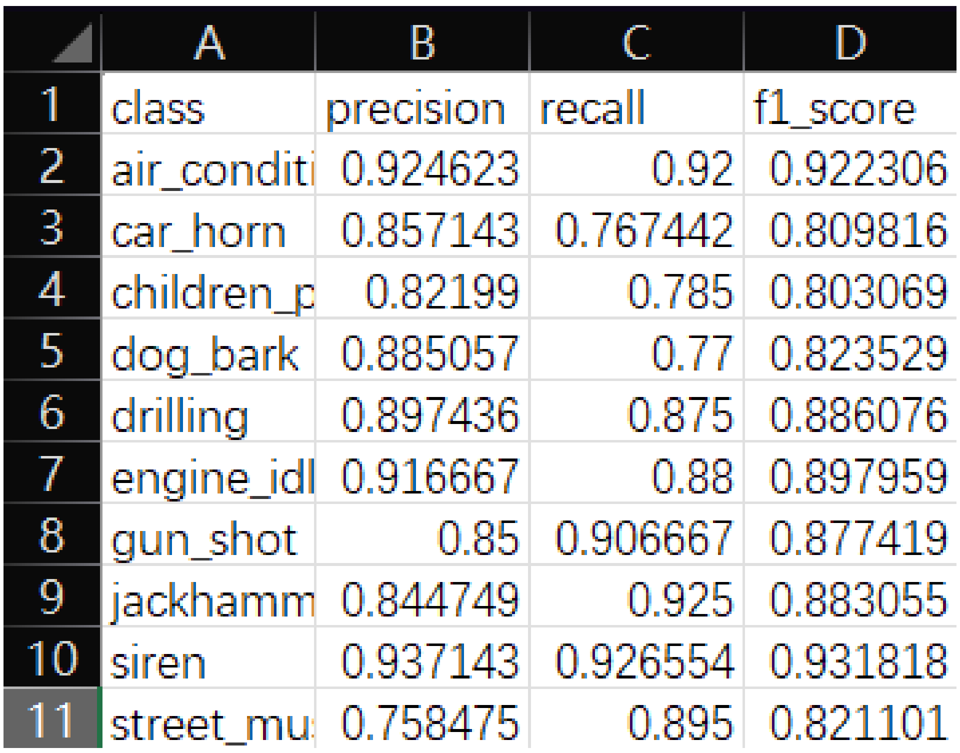
	A	B	C	D	E	F
1	classifier	cv_accura	cv_accura	cv_macro_	cv_macro_f1_std	
2	svm_rbf	0.90237	0.007303	0.905873	0.007151	
3	logreg	0.748789	0.008118	0.760413	0.010237	
4	random_fc	0.874856	0.004704	0.875419	0.005958	



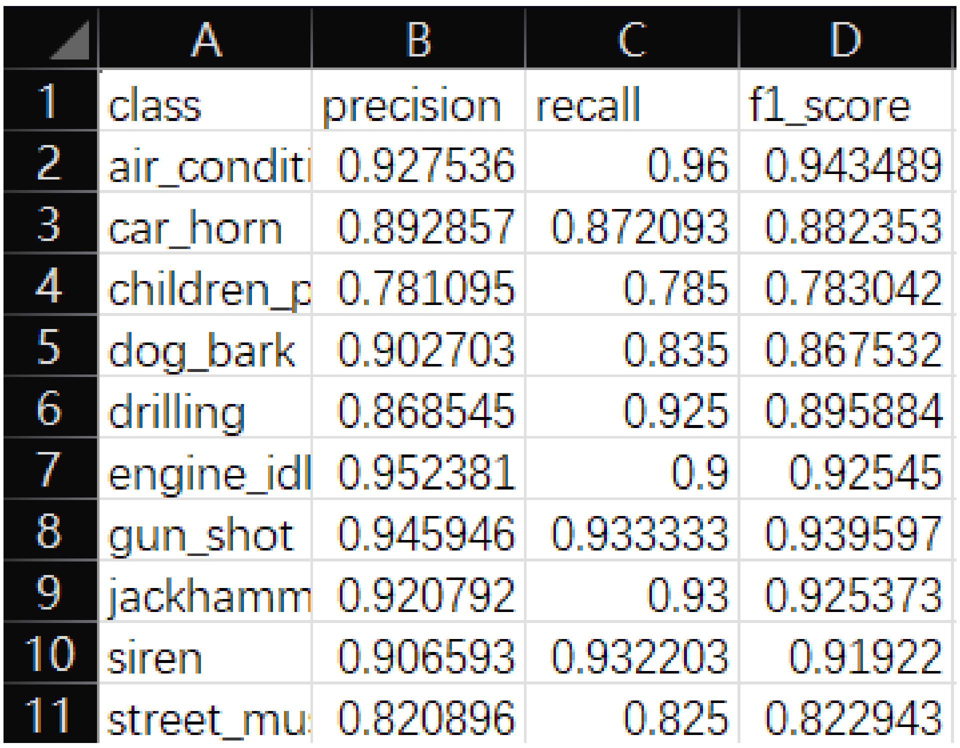
	A	B	C
1	class	count	fraction
2	air_conditi	1000	0.115128
3	car_horn	429	0.04939
4	children_p	1000	0.115128
5	dog_bark	1000	0.115128
6	drilling	1000	0.115128
7	engine_idl	1000	0.115128
8	gun_shot	374	0.043058
9	jackhamm	1000	0.115128
10	siren	883	0.101658
11	street_mu	1000	0.115128



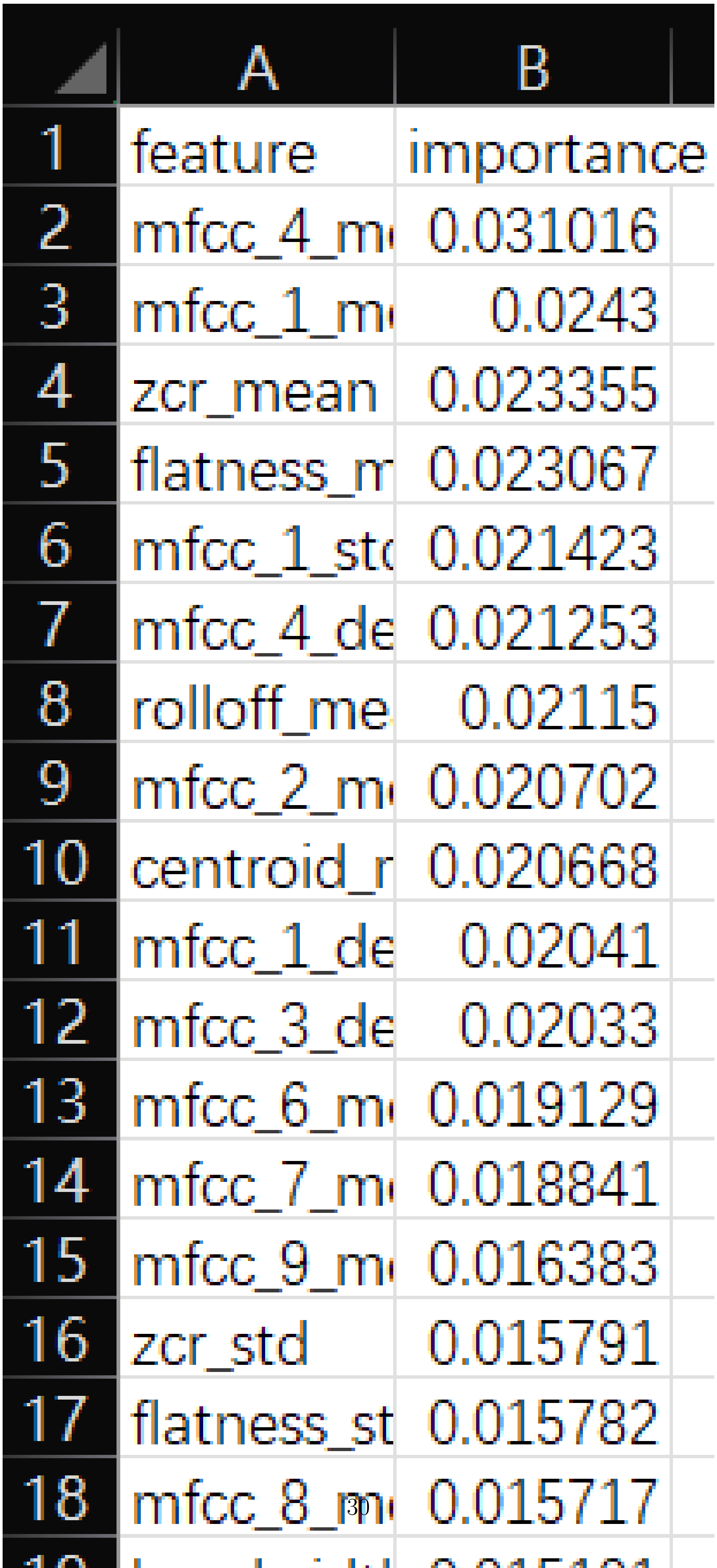
	A	B	C	D
1	class	precision	recall	f1_score
2	air_conditi	0.69863	0.765	0.73031
3	car_horn	0.769231	0.697674	0.731707
4	children_p	0.703125	0.675	0.688776
5	dog_bark	0.78125	0.75	0.765306
6	drilling	0.72	0.72	0.72
7	engine_idl	0.737113	0.715	0.725888
8	gun_shot	0.853333	0.853333	0.853333
9	jackhamm	0.772277	0.78	0.776119
10	siren	0.844828	0.830508	0.837607
11	street_mu	0.707547	0.75	0.728155



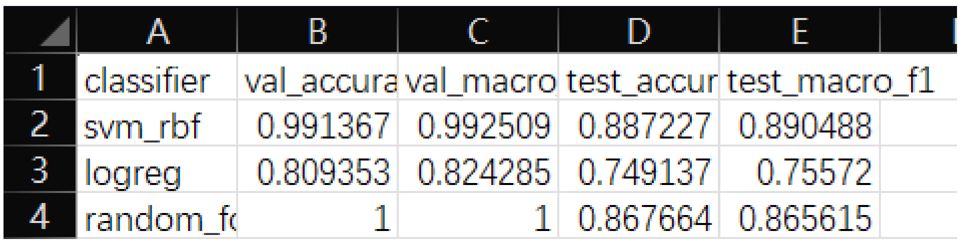
	A	B	C	D
1	class	precision	recall	f1_score
2	air_conditi	0.924623	0.92	0.922306
3	car_horn	0.857143	0.767442	0.809816
4	children_p	0.82199	0.785	0.803069
5	dog_bark	0.885057	0.77	0.823529
6	drilling	0.897436	0.875	0.886076
7	engine_idl	0.916667	0.88	0.897959
8	gun_shot	0.85	0.906667	0.877419
9	jackhamm	0.844749	0.925	0.883055
10	siren	0.937143	0.926554	0.931818
11	street_mu	0.758475	0.895	0.821101



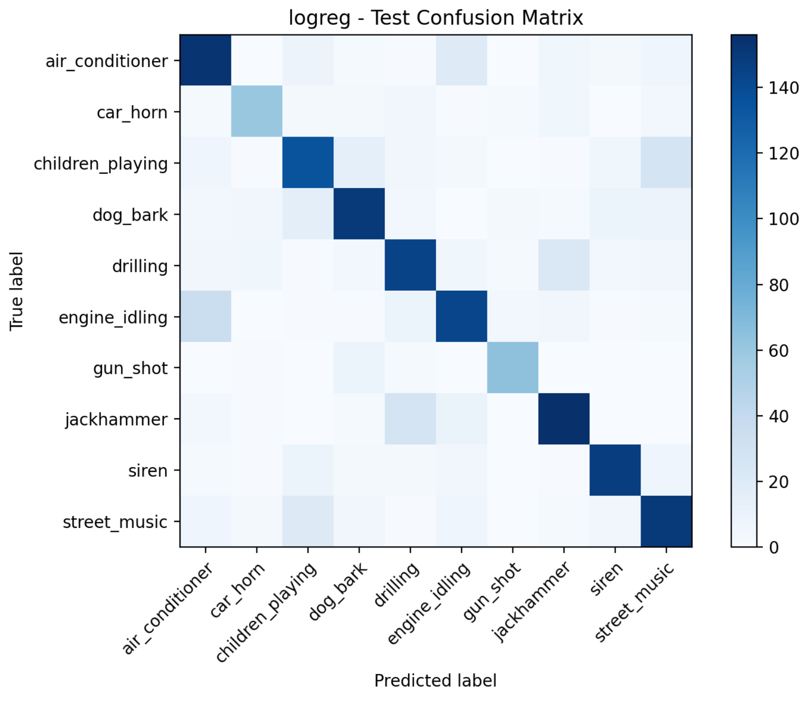
	A	B	C	D
1	class	precision	recall	f1_score
2	air_conditi	0.927536	0.96	0.943489
3	car_horn	0.892857	0.872093	0.882353
4	children_p	0.781095	0.785	0.783042
5	dog_bark	0.902703	0.835	0.867532
6	drilling	0.868545	0.925	0.895884
7	engine_idl	0.952381	0.9	0.92545
8	gun_shot	0.945946	0.933333	0.939597
9	jackhamm	0.920792	0.93	0.925373
10	siren	0.906593	0.932203	0.91922
11	street_mu	0.820896	0.825	0.822943

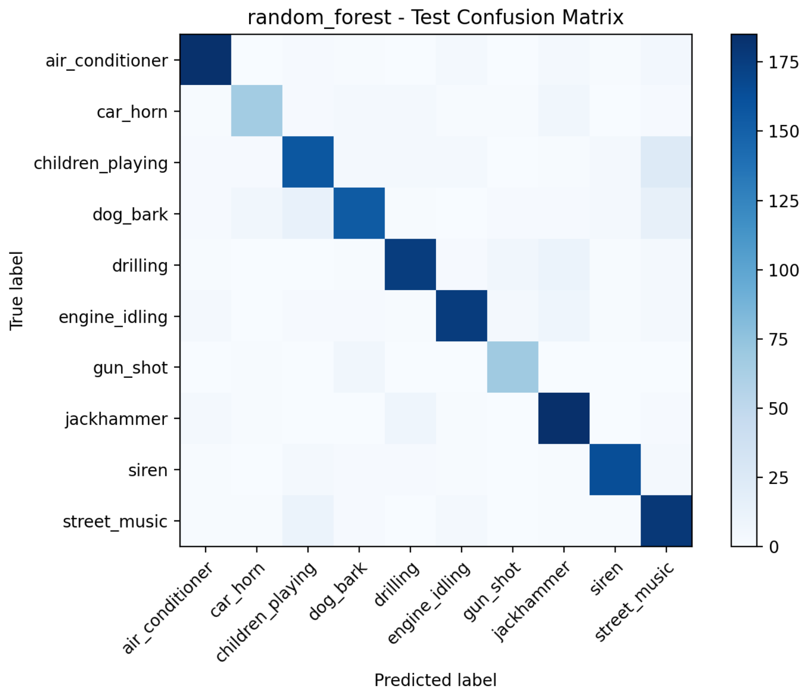


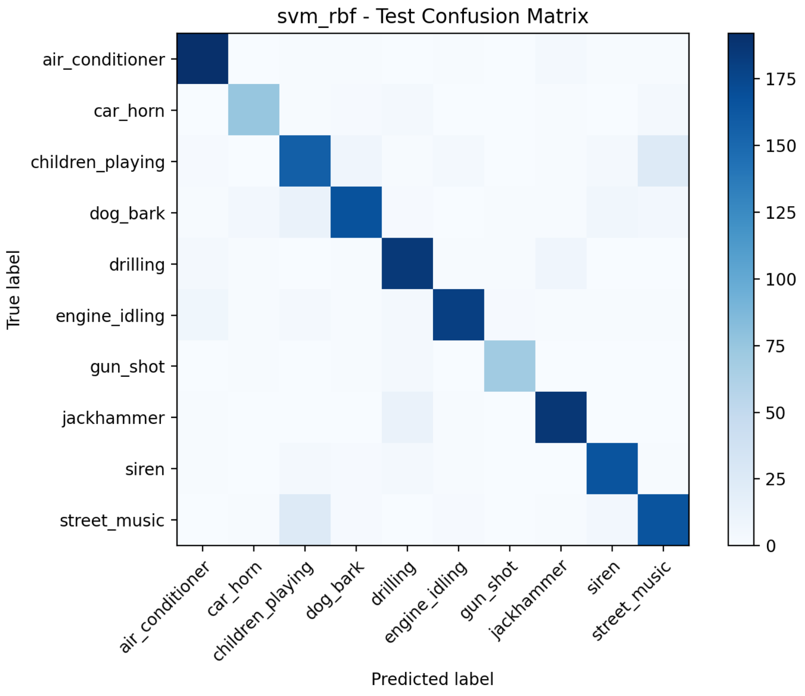
	A	B
1	feature	importance
2	mfcc_4_m	0.031016
3	mfcc_1_m	0.0243
4	zcr_mean	0.023355
5	flatness_m	0.023067
6	mfcc_1_st	0.021423
7	mfcc_4_de	0.021253
8	rolloff_me	0.02115
9	mfcc_2_m	0.020702
10	centroid_r	0.020668
11	mfcc_1_de	0.02041
12	mfcc_3_de	0.02033
13	mfcc_6_m	0.019129
14	mfcc_7_m	0.018841
15	mfcc_9_m	0.016383
16	zcr_std	0.015791
17	flatness_st	0.015782
18	mfcc_8_m	0.015717
19	[illegible]	[illegible]

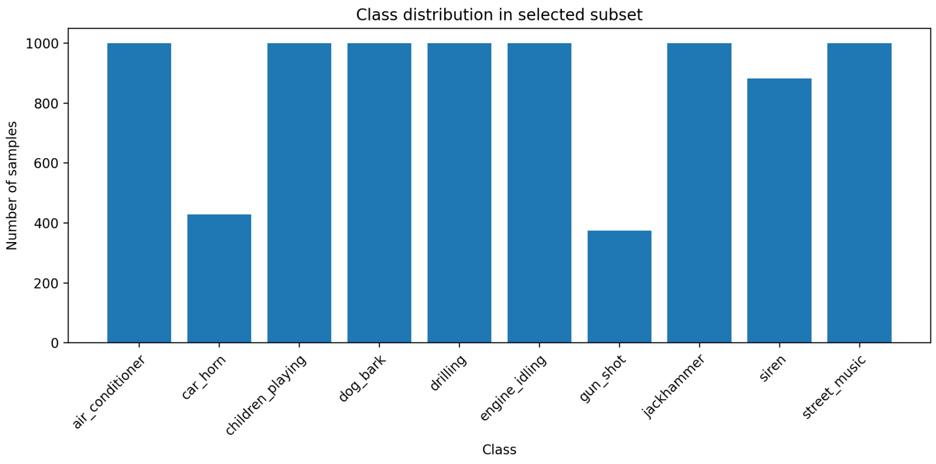


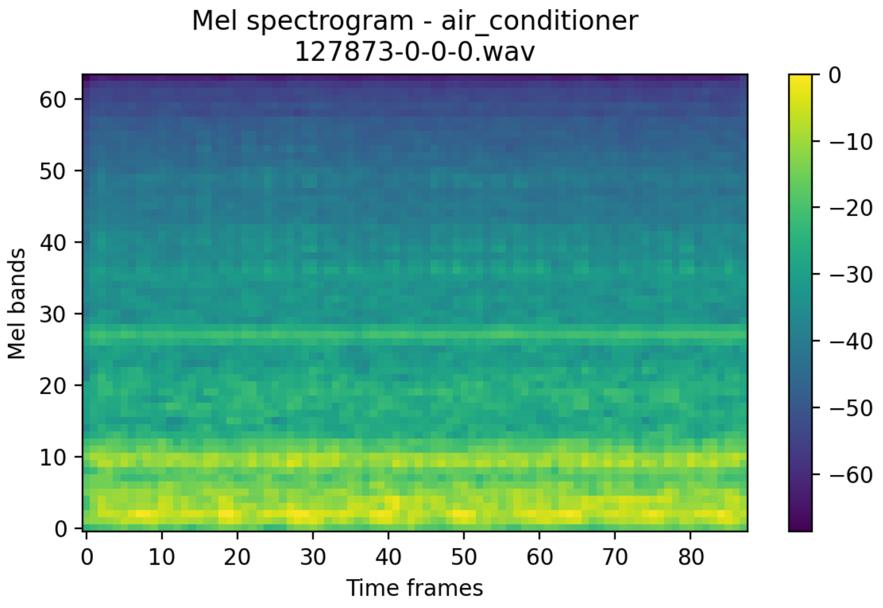
	A	B	C	D	E
1	classifier	val_accura	val_macro	test_accur	test_macro_f1
2	svm_rbf	0.991367	0.992509	0.887227	0.890488
3	logreg	0.809353	0.824285	0.749137	0.75572
4	random_fc	1	1	0.867664	0.865615

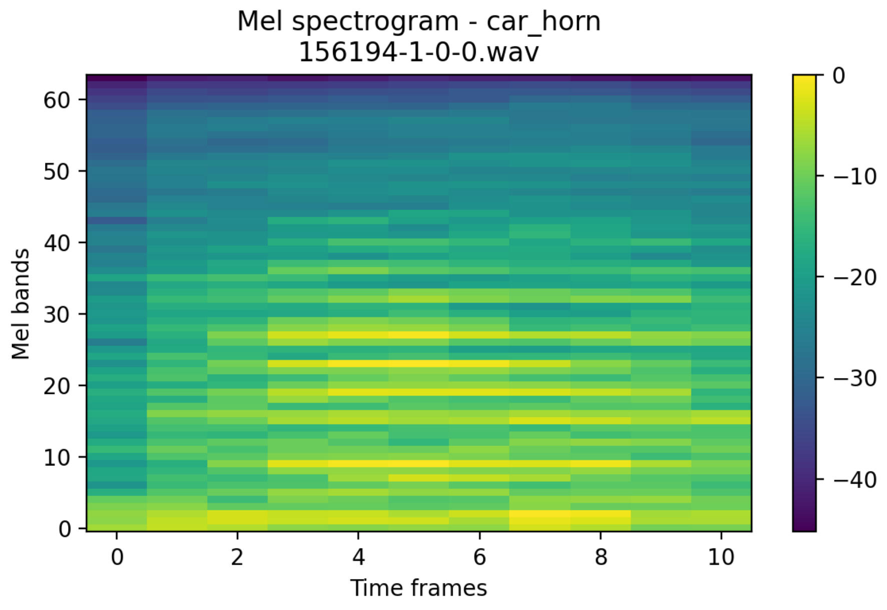




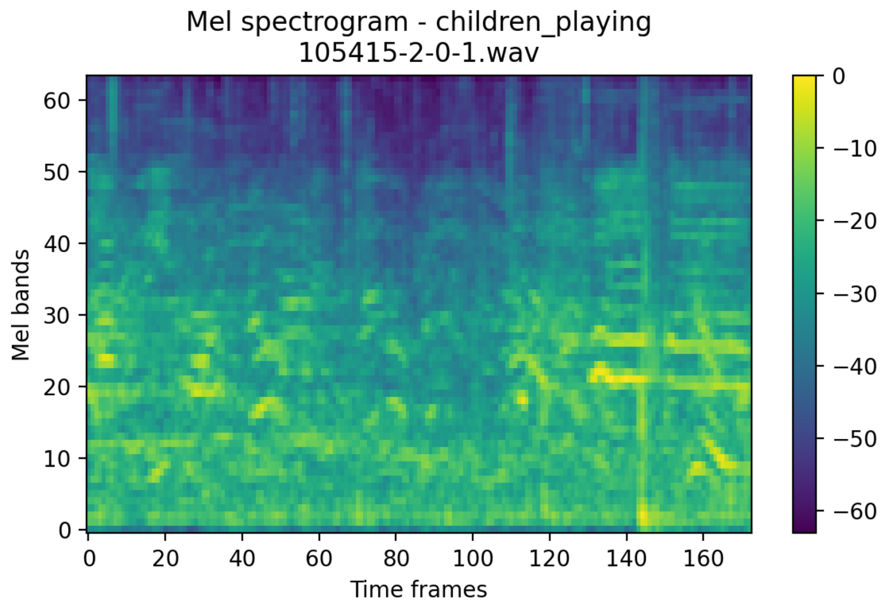

svm_rbf - Test Confusion Matrix
True label
Predicted label
air_conditioner
car_horn
children_playing
dog_bark
drilling
engine_idling
gun_shot
jackhammer
siren
street_music
0
25
50
75
100
125
150
175


Class distribution in selected subset
Number of samples
Class
0
200
400
600
800
1000
air_conditioner
car_horn
children_playing
dog_bark
drilling
engine_idling
gun_shot
jackhammer
siren
street_music

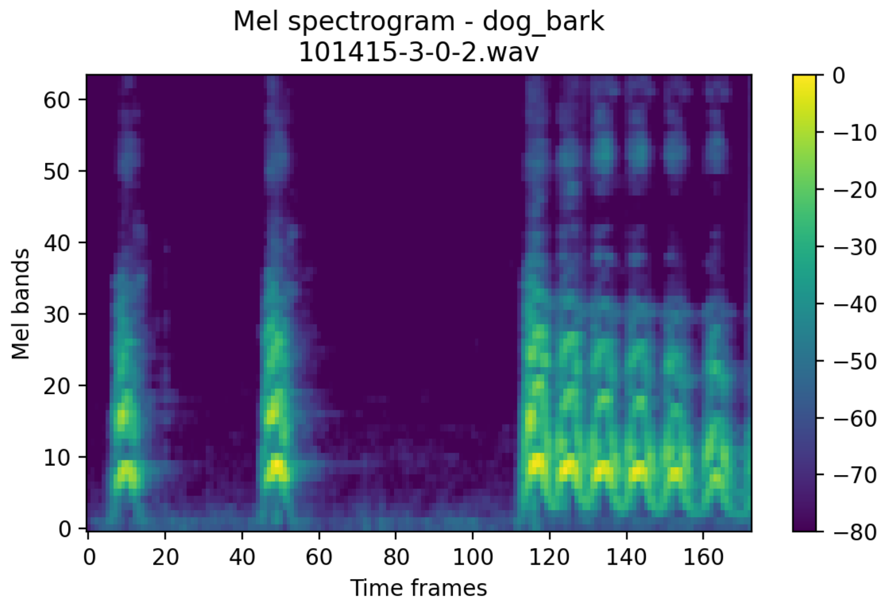

Mel spectrogram - air_conditioner
127873-0-0-0.wav
Mel bands
Time frames
0
10
20
30
40
50
60
70
80
−10
−20
−30
−40
−50
−60

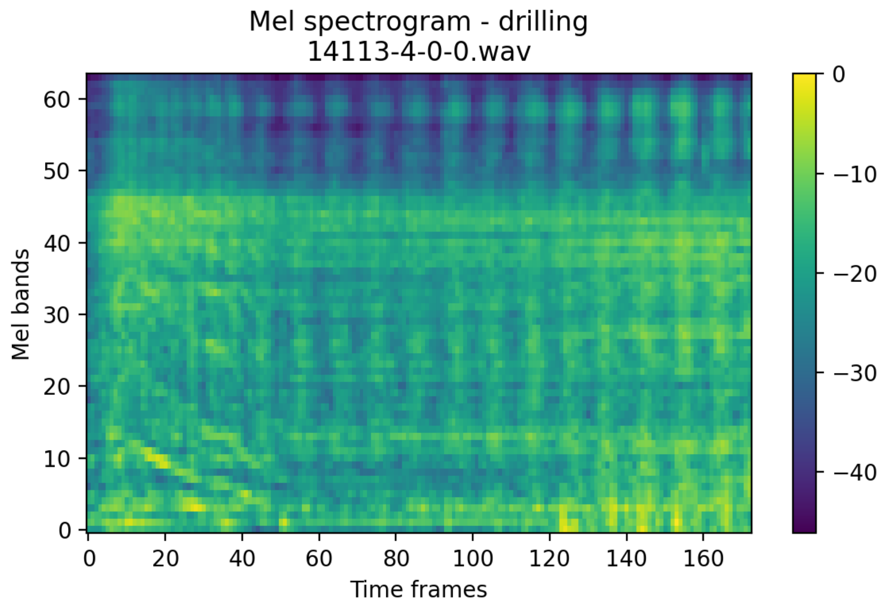


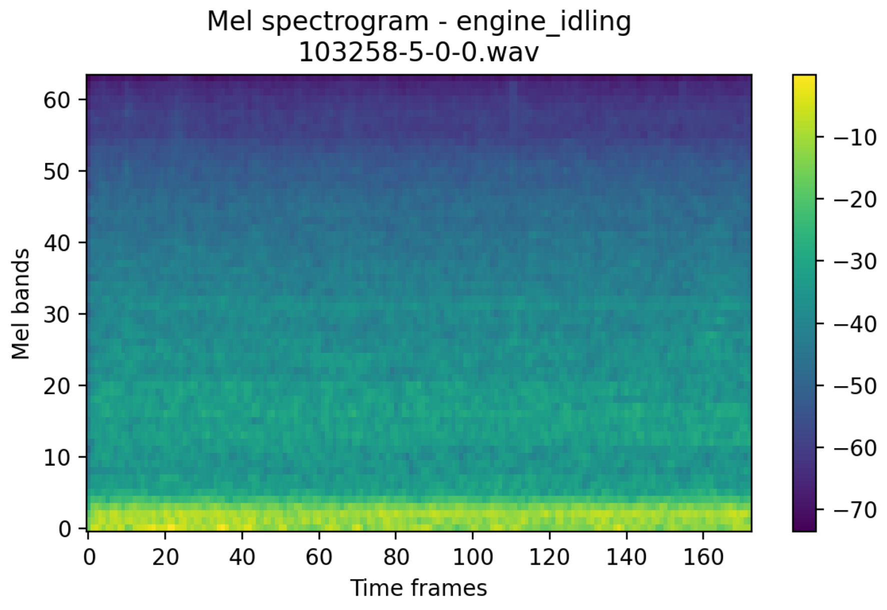
Mel spectrogram - car_horn
156194-1-0-0.wav
Mel bands
Time frames

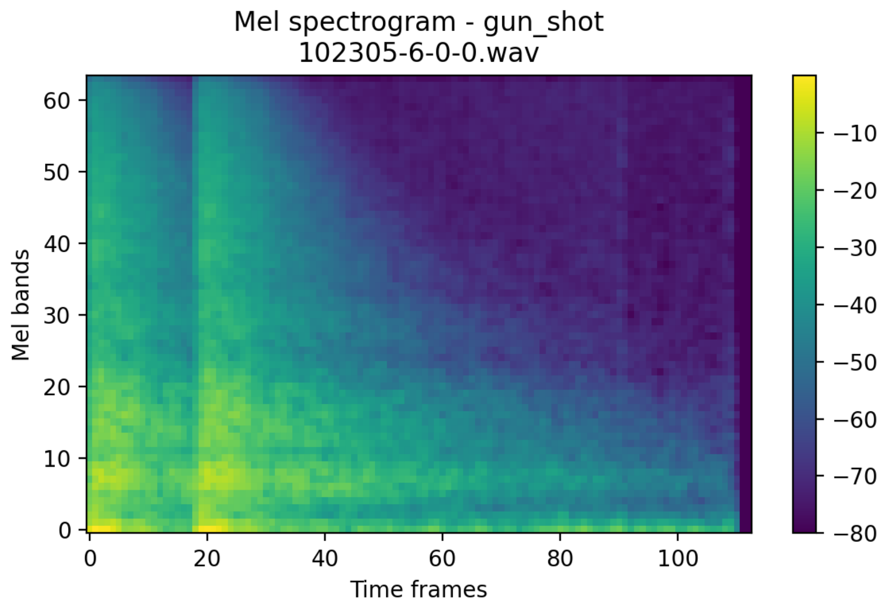


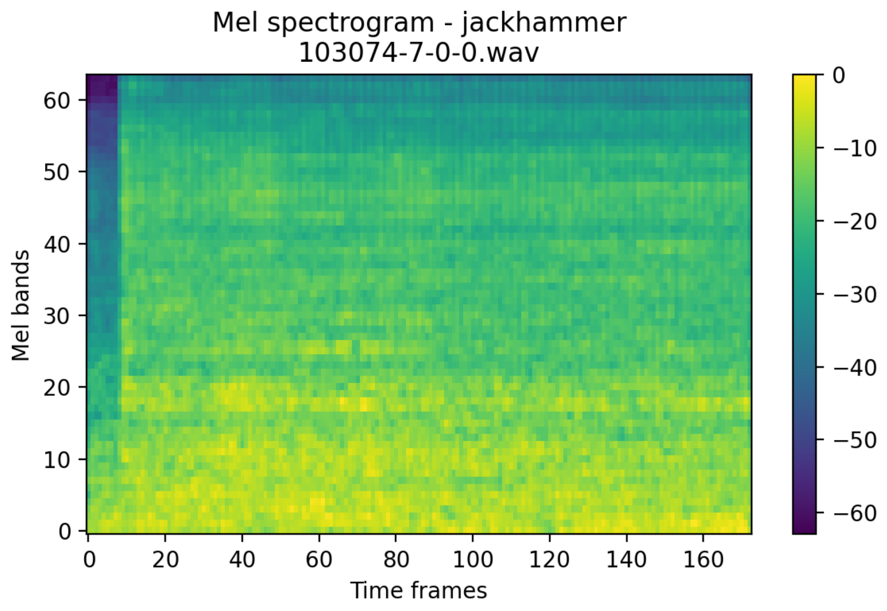
Mel spectrogram - children_playing
105415-2-0-1.wav
Mel bands
Time frames

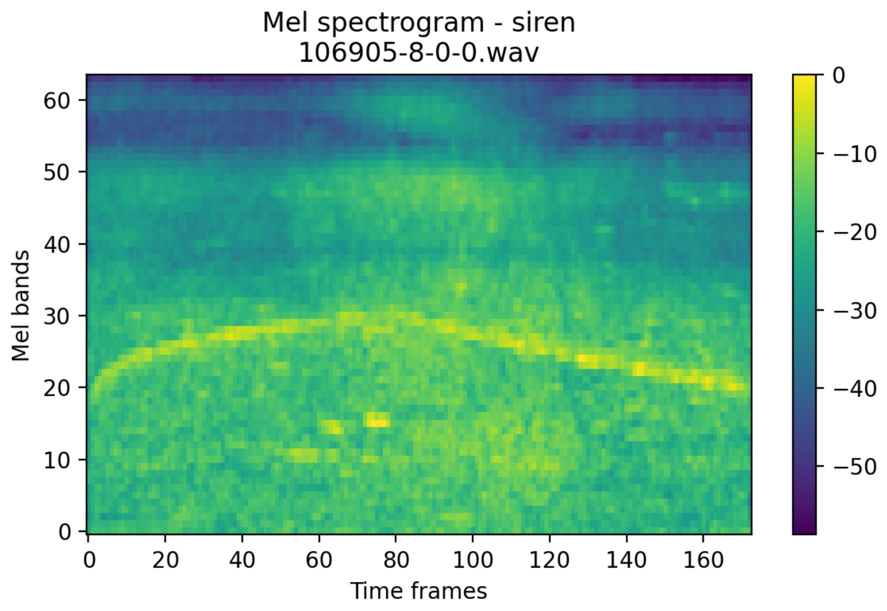

Mel spectrogram - dog_bark
101415-3-0-2.wav
Mel bands
Time frames
0
10
20
30
40
50
60
0
20
40
60
80
100
120
140
160
0
−10
−20
−30
−40
−50
−60
−70
−80

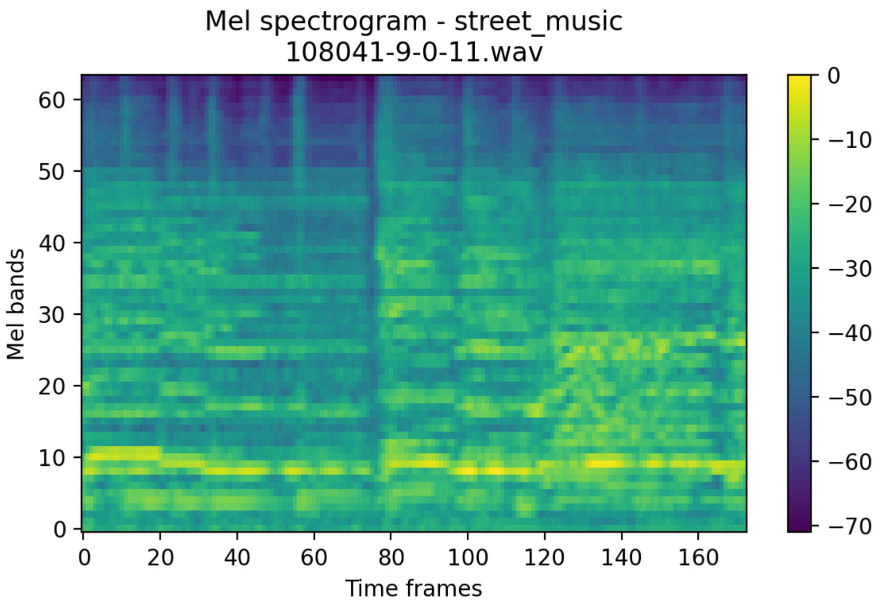

Mel spectrogram - drilling
14113-4-0-0.wav
Mel bands
Time frames
0
10
20
30
40
50
60
0
20
40
60
80
100
120
140
160
0
−10
−20
−30
−40

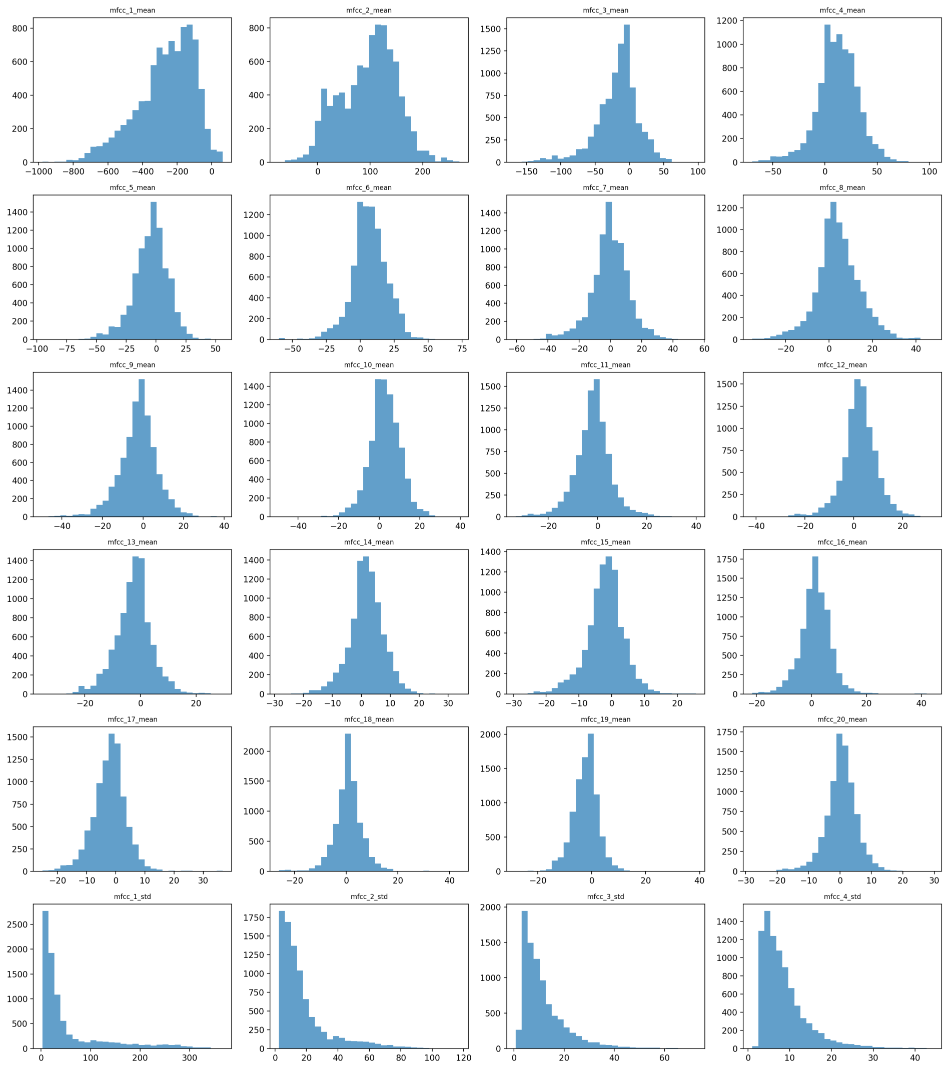

Mel spectrogram - engine_idling
103258-5-0-0.wav
Mel bands
Time frames

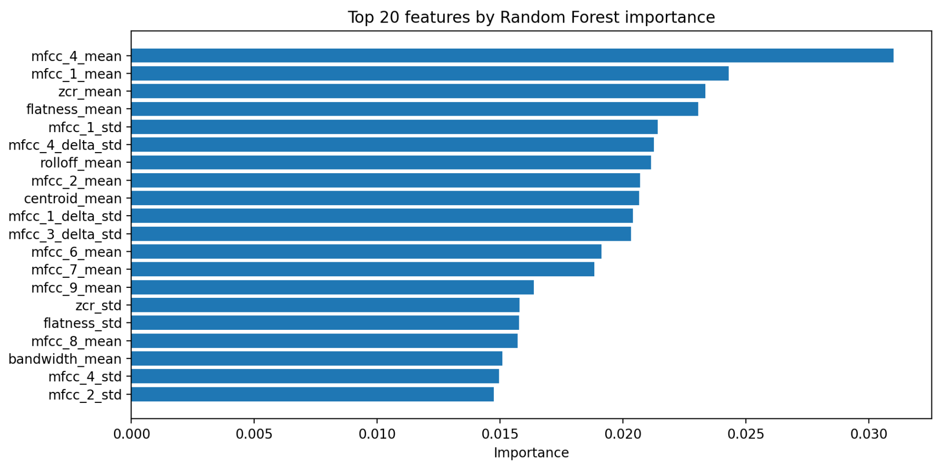

Mel spectrogram - gun_shot
102305-6-0-0.wav
Mel bands
Time frames

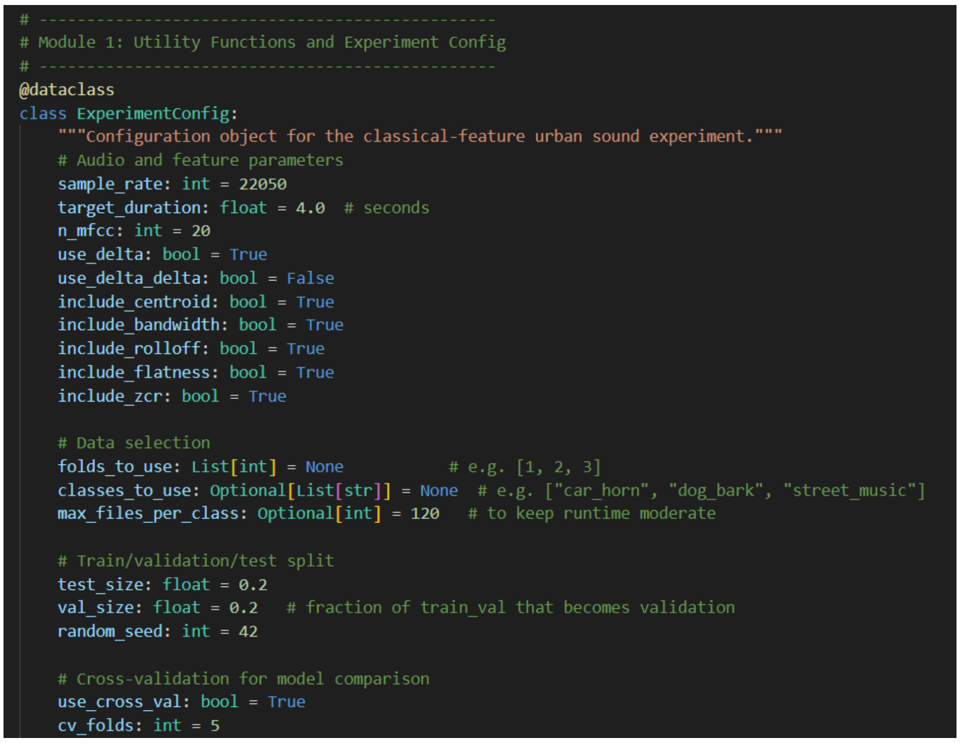

Mel spectrogram - jackhammer
103074-7-0-0.wav
Mel bands
Time frames
0
20
40
60
80
100
120
140
160
10
30
50
−10
−20
−30
−40
−50
−60


Mel spectrogram - siren
106905-8-0-0.wav
Mel bands
Time frames
0
20
40
60
80
100
120
140
160
10
30
50
−10
−20
−30
−40
−50

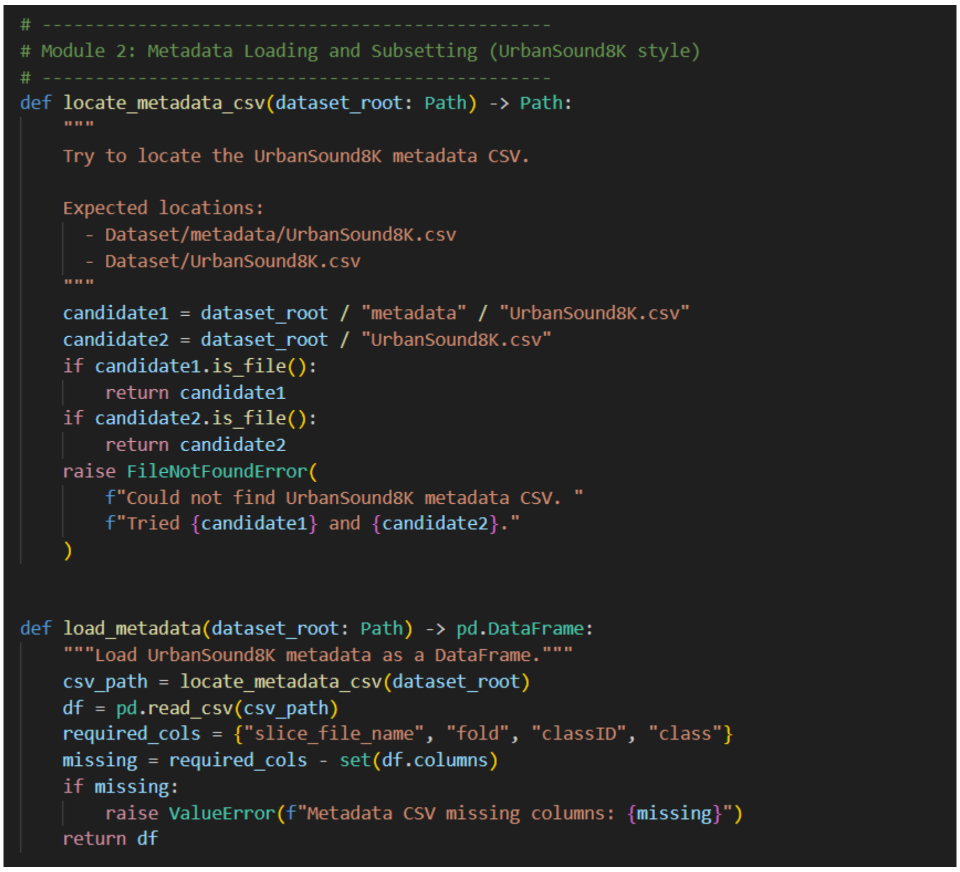

Mel spectrogram - street_music
108041-9-0-11.wav
Mel bands
Time frames


mfcc_1_mean
mfcc_2_mean
mfcc_3_mean
mfcc_4_mean
mfcc_5_mean
mfcc_6_mean
mfcc_7_mean
mfcc_8_mean
mfcc_9_mean
mfcc_10_mean
mfcc_11_mean
mfcc_12_mean
mfcc_13_mean
mfcc_14_mean
mfcc_15_mean
mfcc_16_mean
mfcc_17_mean
mfcc_18_mean
mfcc_19_mean
mfcc_20_mean
mfcc_1_std
mfcc_2_std
mfcc_3_std
mfcc_4_std

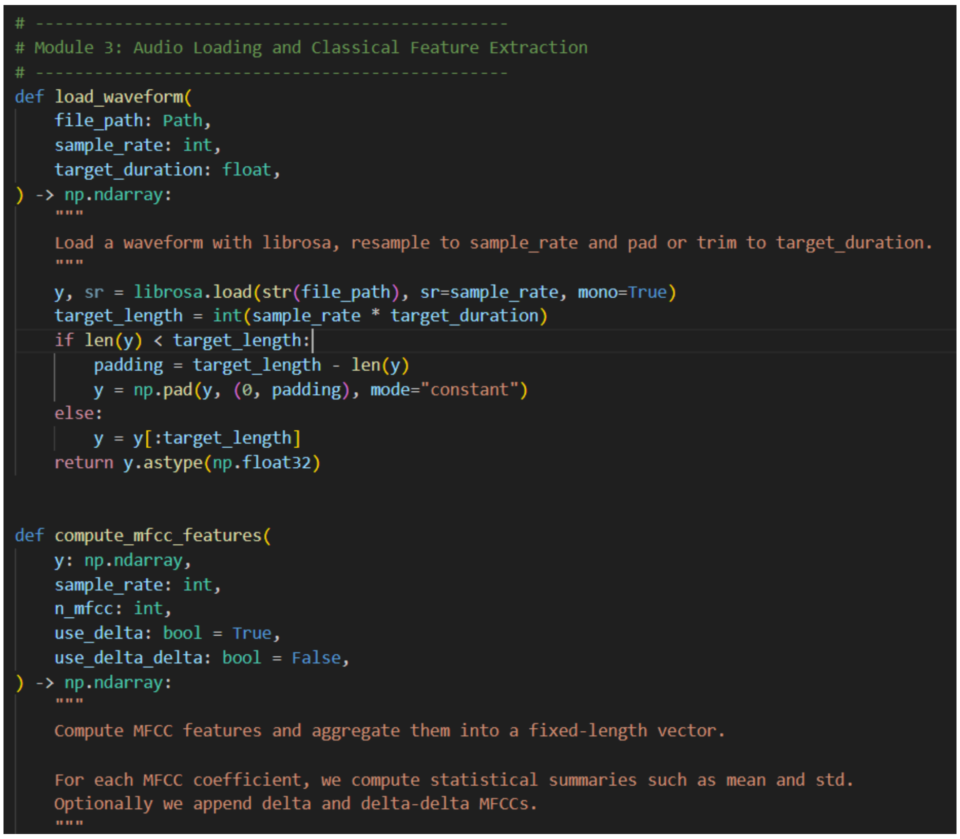




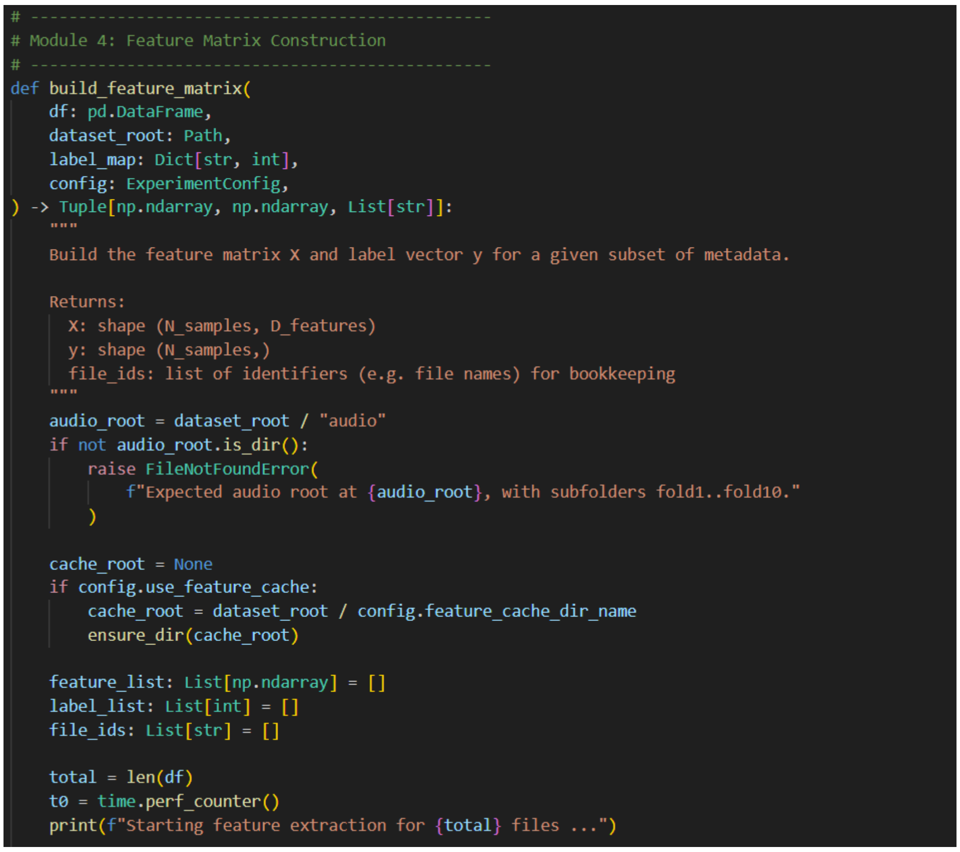
```
# ------------------------------------------------
# Module 1: Utility Functions and Experiment Config
# ------------------------------------------------
@dataclass
class ExperimentConfig:
    """Configuration object for the classical-feature urban sound experiment."""
    # Audio and feature parameters
    sample_rate: int = 22050
    target_duration: float = 4.0  # seconds
    n_mfcc: int = 20
    use_delta: bool = True
    use_delta_delta: bool = False
    include_centroid: bool = True
    include_bandwidth: bool = True
    include_rolloff: bool = True
    include_flatness: bool = True
    include_zcr: bool = True

    # Data selection
    folds_to_use: List[int] = None           # e.g. [1, 2, 3]
    classes_to_use: Optional[List[str]] = None  # e.g. ["car_horn", "dog_bark", "street_music"]
    max_files_per_class: Optional[int] = 120   # to keep runtime moderate

    # Train/validation/test split
    test_size: float = 0.2
    val_size: float = 0.2   # fraction of train_val that becomes validation
    random_seed: int = 42

    # Cross-validation for model comparison
    use_cross_val: bool = True
    cv_folds: int = 5
```



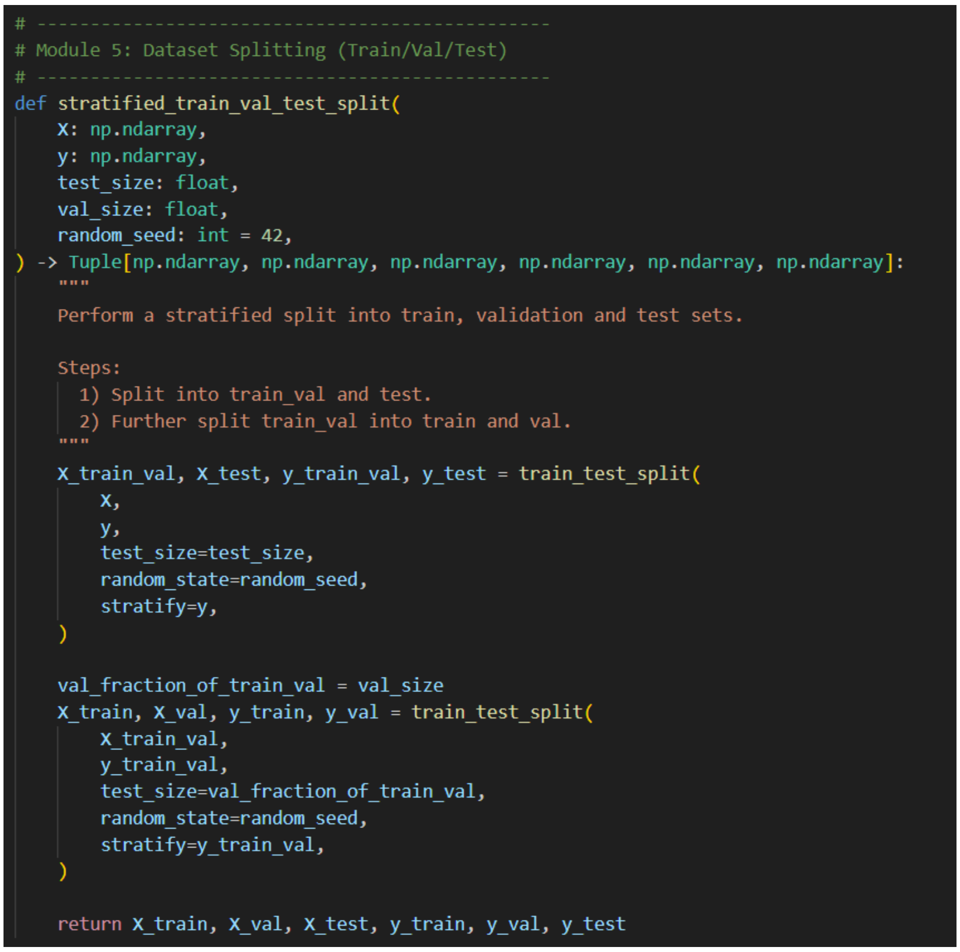
```python
# -----------------------------------------------
# Module 2: Metadata Loading and Subsetting (UrbanSound8K style)
# -----------------------------------------------
def locate_metadata_csv(dataset_root: Path) -> Path:
    """
    Try to locate the UrbanSound8K metadata CSV.

    Expected locations:
      - Dataset/metadata/UrbanSound8K.csv
      - Dataset/UrbanSound8K.csv
    """
    candidate1 = dataset_root / "metadata" / "UrbanSound8K.csv"
    candidate2 = dataset_root / "UrbanSound8K.csv"
    if candidate1.is_file():
        return candidate1
    if candidate2.is_file():
        return candidate2
    raise FileNotFoundError(
        f"Could not find UrbanSound8K metadata CSV. "
        f"Tried {candidate1} and {candidate2}."
    )


def load_metadata(dataset_root: Path) -> pd.DataFrame:
    """Load UrbanSound8K metadata as a DataFrame."""
    csv_path = locate_metadata_csv(dataset_root)
    df = pd.read_csv(csv_path)
    required_cols = {"slice_file_name", "fold", "classID", "class"}
    missing = required_cols - set(df.columns)
    if missing:
        raise ValueError(f"Metadata CSV missing columns: {missing}")
    return df
```



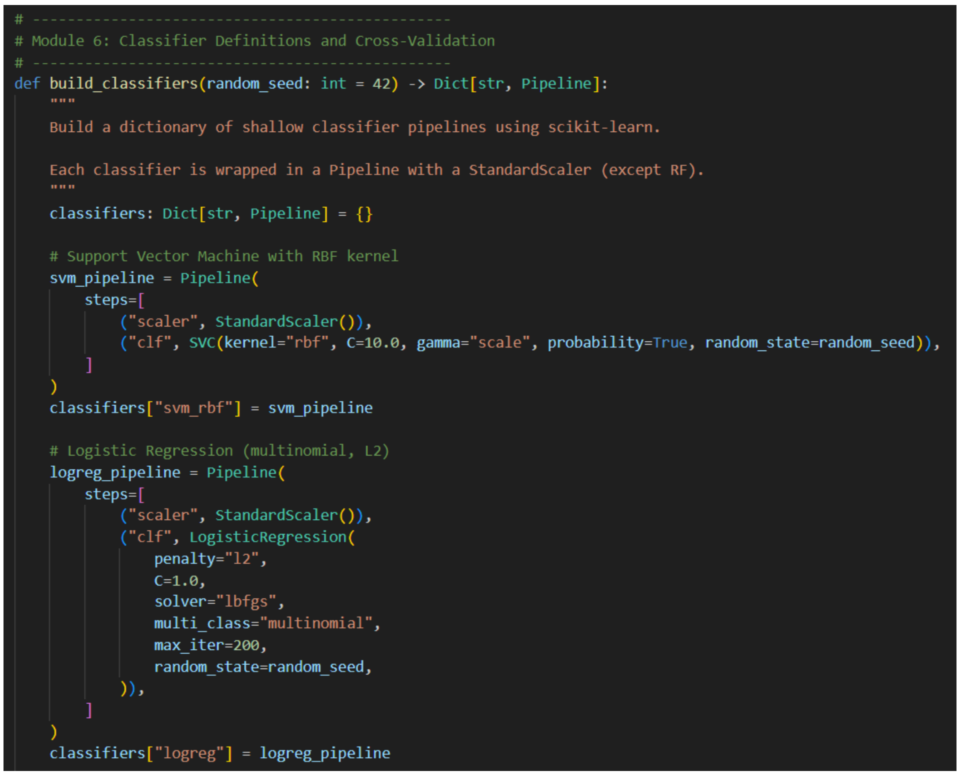
```python
# -----------------------------------------------
# Module 3: Audio Loading and Classical Feature Extraction
# -----------------------------------------------
def load_waveform(
    file_path: Path,
    sample_rate: int,
    target_duration: float,
) -> np.ndarray:
    """
    Load a waveform with librosa, resample to sample_rate and pad or trim to target_duration.
    """
    y, sr = librosa.load(str(file_path), sr=sample_rate, mono=True)
    target_length = int(sample_rate * target_duration)
    if len(y) < target_length:
        padding = target_length - len(y)
        y = np.pad(y, (0, padding), mode="constant")
    else:
        y = y[:target_length]
    return y.astype(np.float32)


def compute_mfcc_features(
    y: np.ndarray,
    sample_rate: int,
    n_mfcc: int,
    use_delta: bool = True,
    use_delta_delta: bool = False,
) -> np.ndarray:
    """
    Compute MFCC features and aggregate them into a fixed-length vector.

    For each MFCC coefficient, we compute statistical summaries such as mean and std.
    Optionally we append delta and delta-delta MFCCs.
    """
```



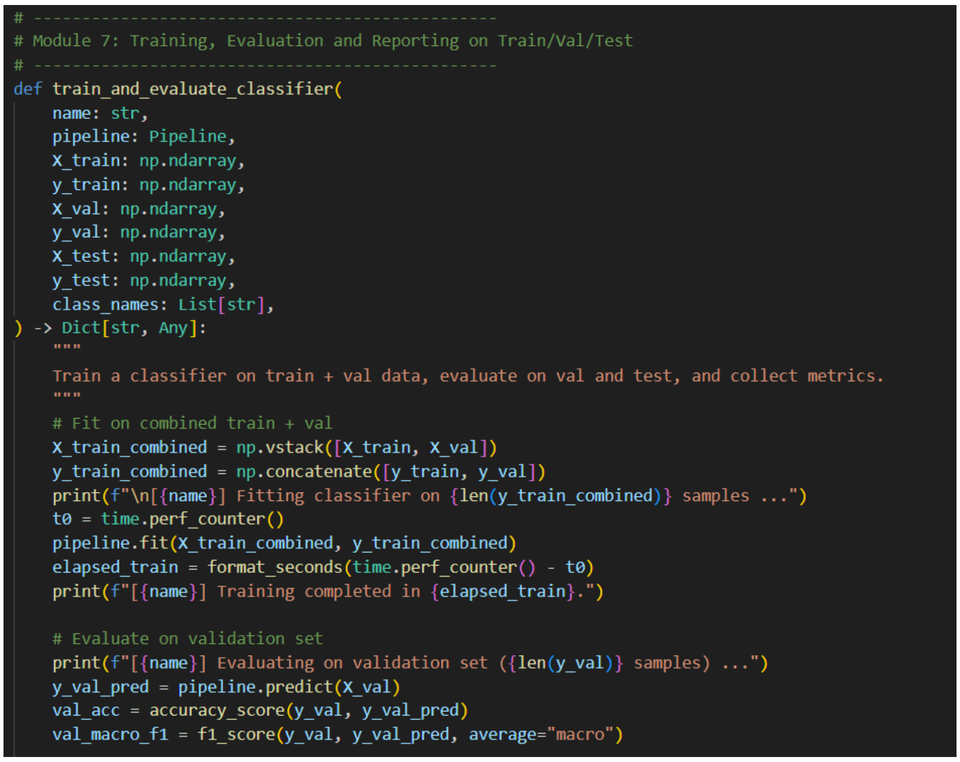
```python
# ----------------------------------------------
# Module 4: Feature Matrix Construction
# ----------------------------------------------
def build_feature_matrix(
    df: pd.DataFrame,
    dataset_root: Path,
    label_map: Dict[str, int],
    config: ExperimentConfig,
) -> Tuple[np.ndarray, np.ndarray, List[str]]:
    """
    Build the feature matrix X and label vector y for a given subset of metadata.

    Returns:
      X: shape (N_samples, D_features)
      y: shape (N_samples,)
      file_ids: list of identifiers (e.g. file names) for bookkeeping
    """
    audio_root = dataset_root / "audio"
    if not audio_root.is_dir():
        raise FileNotFoundError(
            f"Expected audio root at {audio_root}, with subfolders fold1..fold10."
        )

    cache_root = None
    if config.use_feature_cache:
        cache_root = dataset_root / config.feature_cache_dir_name
        ensure_dir(cache_root)

    feature_list: List[np.ndarray] = []
    label_list: List[int] = []
    file_ids: List[str] = []

    total = len(df)
    t0 = time.perf_counter()
    print(f"Starting feature extraction for {total} files ...")
```



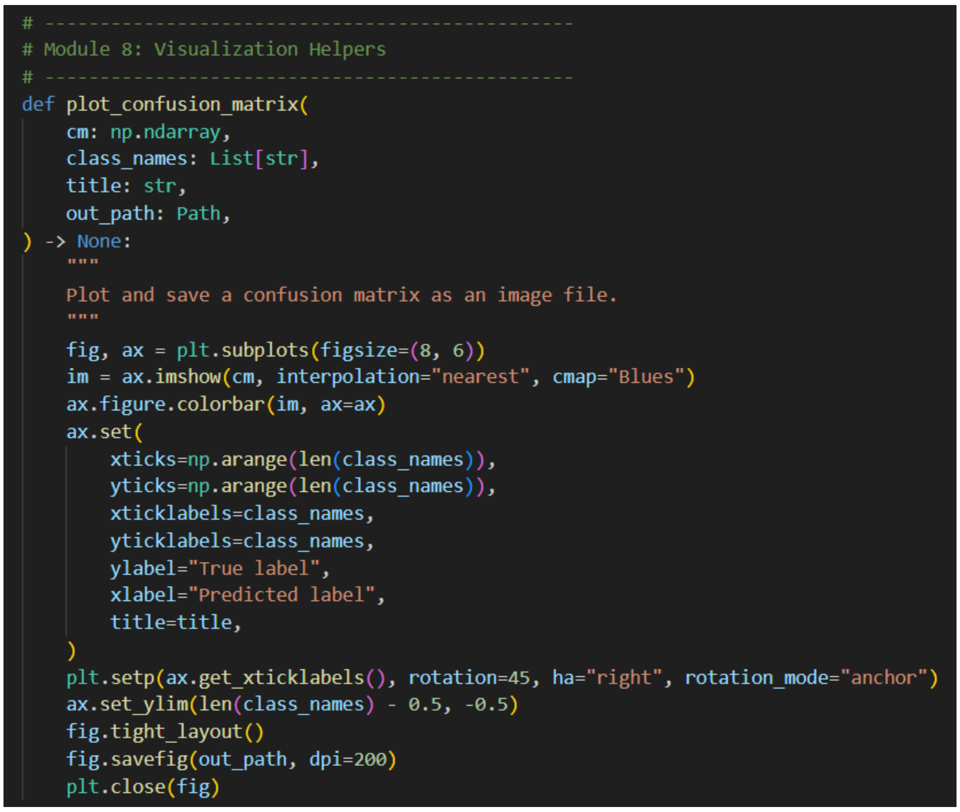
```python
# ------------------------------------------------
# Module 5: Dataset Splitting (Train/Val/Test)
# ------------------------------------------------
def stratified_train_val_test_split(
    X: np.ndarray,
    y: np.ndarray,
    test_size: float,
    val_size: float,
    random_seed: int = 42,
) -> Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
    """
    Perform a stratified split into train, validation and test sets.

    Steps:
      1) Split into train_val and test.
      2) Further split train_val into train and val.
    """
    X_train_val, X_test, y_train_val, y_test = train_test_split(
        X,
        y,
        test_size=test_size,
        random_state=random_seed,
        stratify=y,
    )

    val_fraction_of_train_val = val_size
    X_train, X_val, y_train, y_val = train_test_split(
        X_train_val,
        y_train_val,
        test_size=val_fraction_of_train_val,
        random_state=random_seed,
        stratify=y_train_val,
    )

    return X_train, X_val, X_test, y_train, y_val, y_test
```



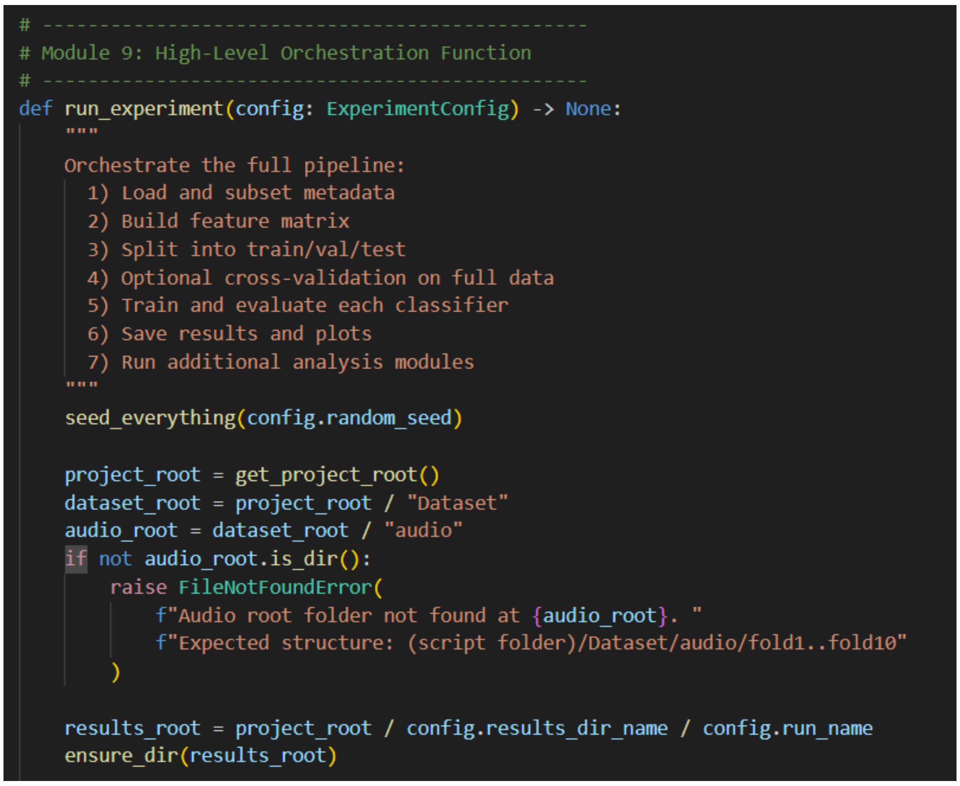
```python
# -----------------------------------------------
# Module 6: Classifier Definitions and Cross-Validation
# -----------------------------------------------
def build_classifiers(random_seed: int = 42) -> Dict[str, Pipeline]:
    """
    Build a dictionary of shallow classifier pipelines using scikit-learn.

    Each classifier is wrapped in a Pipeline with a StandardScaler (except RF).
    """
    classifiers: Dict[str, Pipeline] = {}

    # Support Vector Machine with RBF kernel
    svm_pipeline = Pipeline(
        steps=[
            ("scaler", StandardScaler()),
            ("clf", SVC(kernel="rbf", C=10.0, gamma="scale", probability=True, random_state=random_seed)),
        ]
    )
    classifiers["svm_rbf"] = svm_pipeline

    # Logistic Regression (multinomial, L2)
    logreg_pipeline = Pipeline(
        steps=[
            ("scaler", StandardScaler()),
            ("clf", LogisticRegression(
                penalty="l2",
                C=1.0,
                solver="lbfgs",
                multi_class="multinomial",
                max_iter=200,
                random_state=random_seed,
            )),
        ]
    )
    classifiers["logreg"] = logreg_pipeline
```



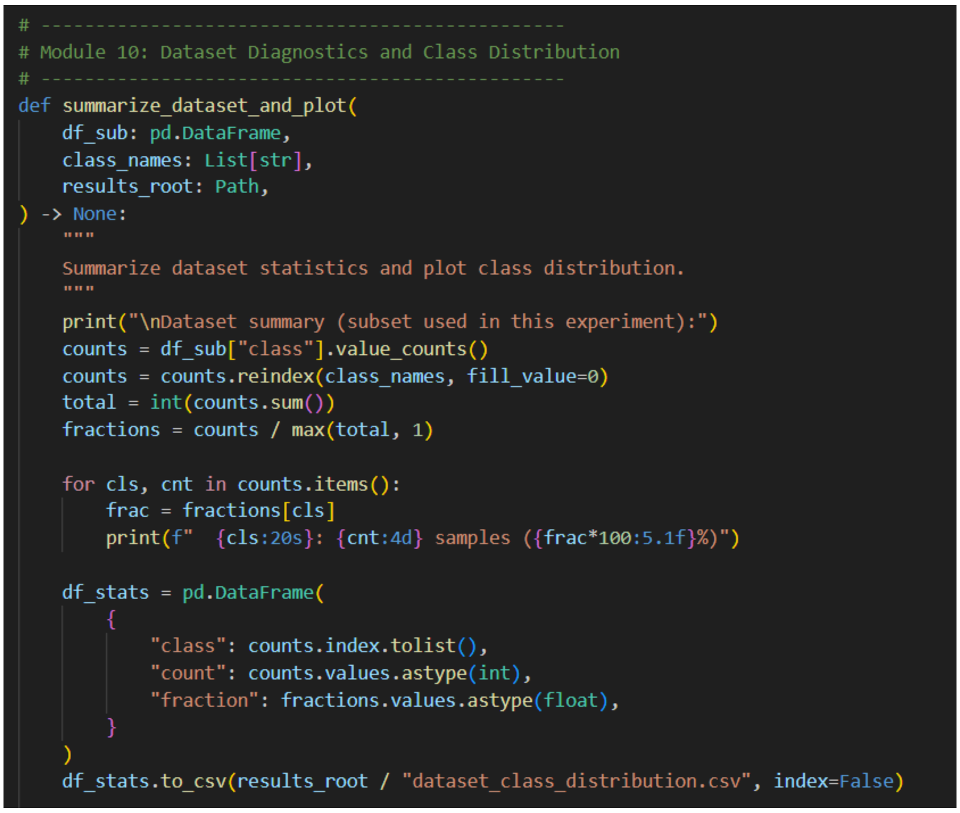
```python
# ------------------------------------------------
# Module 7: Training, Evaluation and Reporting on Train/Val/Test
# ------------------------------------------------
def train_and_evaluate_classifier(
    name: str,
    pipeline: Pipeline,
    X_train: np.ndarray,
    y_train: np.ndarray,
    X_val: np.ndarray,
    y_val: np.ndarray,
    X_test: np.ndarray,
    y_test: np.ndarray,
    class_names: List[str],
) -> Dict[str, Any]:
    """
    Train a classifier on train + val data, evaluate on val and test, and collect metrics.
    """
    # Fit on combined train + val
    X_train_combined = np.vstack([X_train, X_val])
    y_train_combined = np.concatenate([y_train, y_val])
    print(f"\n[{name}] Fitting classifier on {len(y_train_combined)} samples ...")
    t0 = time.perf_counter()
    pipeline.fit(X_train_combined, y_train_combined)
    elapsed_train = format_seconds(time.perf_counter() - t0)
    print(f"[{name}] Training completed in {elapsed_train}.")

    # Evaluate on validation set
    print(f"[{name}] Evaluating on validation set ({len(y_val)} samples) ...")
    y_val_pred = pipeline.predict(X_val)
    val_acc = accuracy_score(y_val, y_val_pred)
    val_macro_f1 = f1_score(y_val, y_val_pred, average="macro")
```



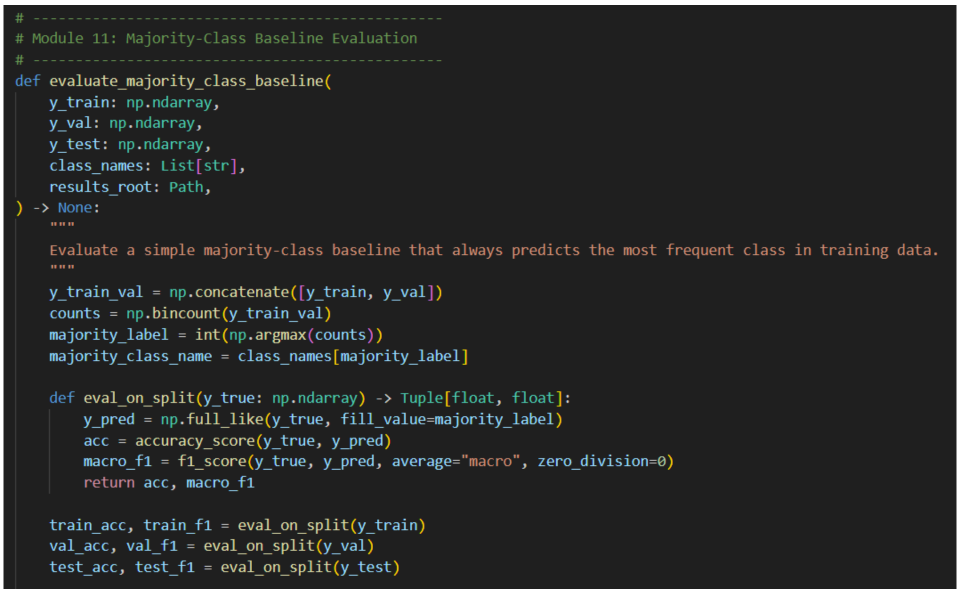
```python
# ------------------------------------------------
# Module 8: Visualization Helpers
# ------------------------------------------------
def plot_confusion_matrix(
    cm: np.ndarray,
    class_names: List[str],
    title: str,
    out_path: Path,
) -> None:
    """
    Plot and save a confusion matrix as an image file.
    """
    fig, ax = plt.subplots(figsize=(8, 6))
    im = ax.imshow(cm, interpolation="nearest", cmap="Blues")
    ax.figure.colorbar(im, ax=ax)
    ax.set(
        xticks=np.arange(len(class_names)),
        yticks=np.arange(len(class_names)),
        xticklabels=class_names,
        yticklabels=class_names,
        ylabel="True label",
        xlabel="Predicted label",
        title=title,
    )
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right", rotation_mode="anchor")
    ax.set_ylim(len(class_names) - 0.5, -0.5)
    fig.tight_layout()
    fig.savefig(out_path, dpi=200)
    plt.close(fig)
```



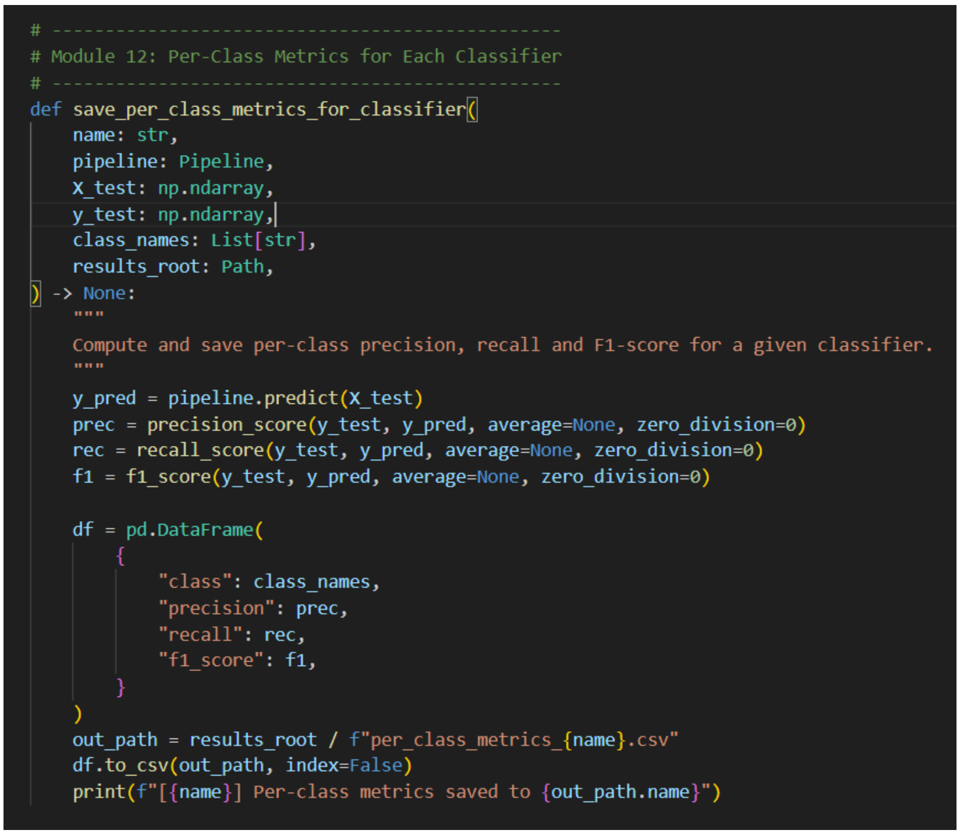
```python
# ------------------------------------------------
# Module 9: High-Level Orchestration Function
# ------------------------------------------------
def run_experiment(config: ExperimentConfig) -> None:
    """
    Orchestrate the full pipeline:
      1) Load and subset metadata
      2) Build feature matrix
      3) Split into train/val/test
      4) Optional cross-validation on full data
      5) Train and evaluate each classifier
      6) Save results and plots
      7) Run additional analysis modules
    """
    seed_everything(config.random_seed)

    project_root = get_project_root()
    dataset_root = project_root / "Dataset"
    audio_root = dataset_root / "audio"
    if not audio_root.is_dir():
        raise FileNotFoundError(
            f"Audio root folder not found at {audio_root}. "
            f"Expected structure: (script folder)/Dataset/audio/fold1..fold10"
        )

    results_root = project_root / config.results_dir_name / config.run_name
    ensure_dir(results_root)
```



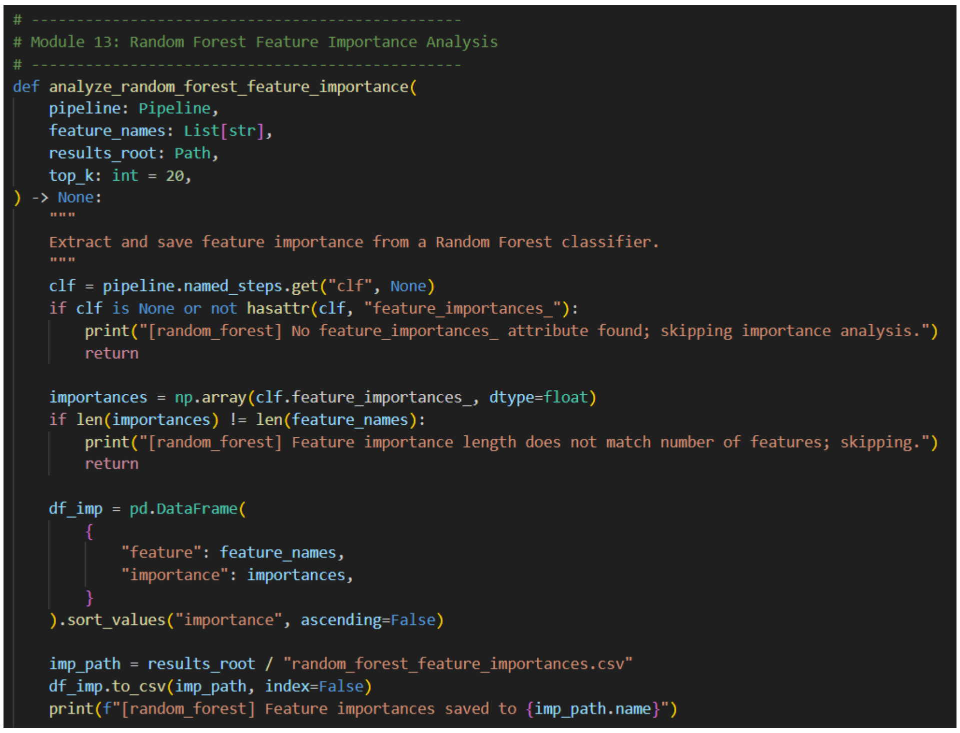
```python
# ------------------------------------------------
# Module 10: Dataset Diagnostics and Class Distribution
# ------------------------------------------------
def summarize_dataset_and_plot(
    df_sub: pd.DataFrame,
    class_names: List[str],
    results_root: Path,
) -> None:
    """
    Summarize dataset statistics and plot class distribution.
    """
    print("\nDataset summary (subset used in this experiment):")
    counts = df_sub["class"].value_counts()
    counts = counts.reindex(class_names, fill_value=0)
    total = int(counts.sum())
    fractions = counts / max(total, 1)

    for cls, cnt in counts.items():
        frac = fractions[cls]
        print(f"  {cls:20s}: {cnt:4d} samples ({frac*100:5.1f}%)")

    df_stats = pd.DataFrame(
        {
            "class": counts.index.tolist(),
            "count": counts.values.astype(int),
            "fraction": fractions.values.astype(float),
        }
    )
    df_stats.to_csv(results_root / "dataset_class_distribution.csv", index=False)
```



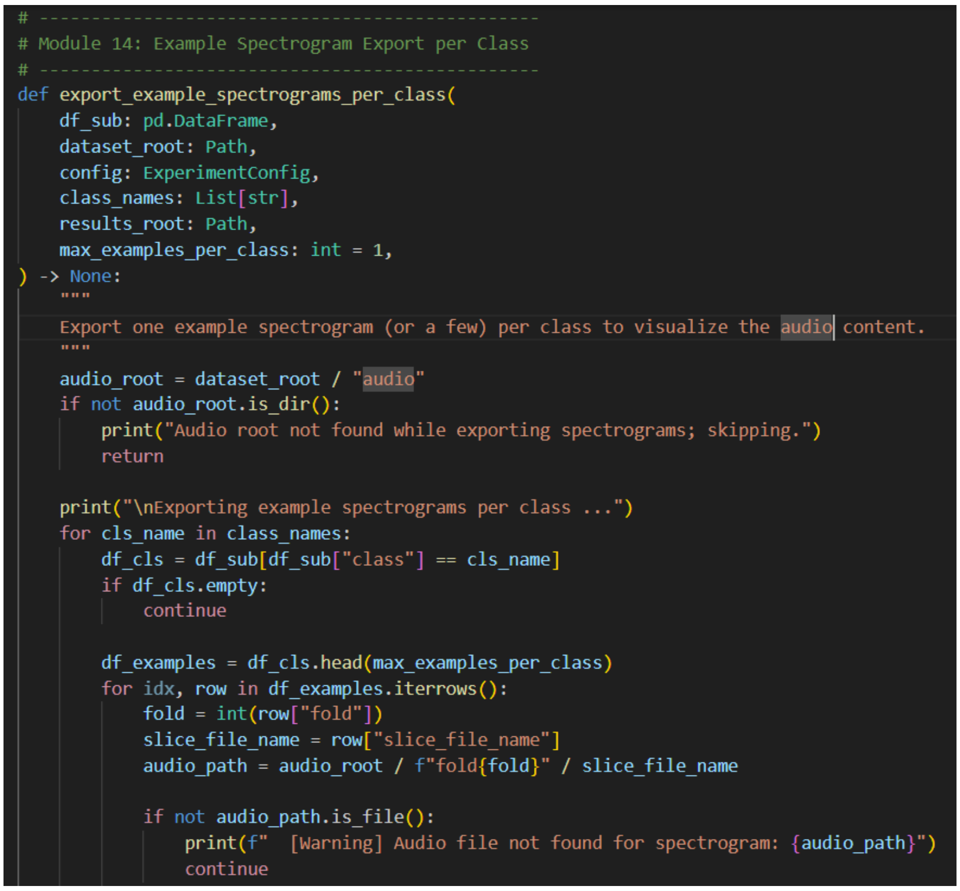
```python
# ------------------------------------------------
# Module 11: Majority-Class Baseline Evaluation
# ------------------------------------------------
def evaluate_majority_class_baseline(
    y_train: np.ndarray,
    y_val: np.ndarray,
    y_test: np.ndarray,
    class_names: List[str],
    results_root: Path,
) -> None:
    """
    Evaluate a simple majority-class baseline that always predicts the most frequent class in training data.
    """
    y_train_val = np.concatenate([y_train, y_val])
    counts = np.bincount(y_train_val)
    majority_label = int(np.argmax(counts))
    majority_class_name = class_names[majority_label]

    def eval_on_split(y_true: np.ndarray) -> Tuple[float, float]:
        y_pred = np.full_like(y_true, fill_value=majority_label)
        acc = accuracy_score(y_true, y_pred)
        macro_f1 = f1_score(y_true, y_pred, average="macro", zero_division=0)
        return acc, macro_f1

    train_acc, train_f1 = eval_on_split(y_train)
    val_acc, val_f1 = eval_on_split(y_val)
    test_acc, test_f1 = eval_on_split(y_test)
```



```python
# ------------------------------------------------
# Module 12: Per-Class Metrics for Each Classifier
# ------------------------------------------------
def save_per_class_metrics_for_classifier(
    name: str,
    pipeline: Pipeline,
    X_test: np.ndarray,
    y_test: np.ndarray,
    class_names: List[str],
    results_root: Path,
) -> None:
    """
    Compute and save per-class precision, recall and F1-score for a given classifier.
    """
    y_pred = pipeline.predict(X_test)
    prec = precision_score(y_test, y_pred, average=None, zero_division=0)
    rec = recall_score(y_test, y_pred, average=None, zero_division=0)
    f1 = f1_score(y_test, y_pred, average=None, zero_division=0)

    df = pd.DataFrame(
        {
            "class": class_names,
            "precision": prec,
            "recall": rec,
            "f1_score": f1,
        }
    )
    out_path = results_root / f"per_class_metrics_{name}.csv"
    df.to_csv(out_path, index=False)
    print(f"[{name}] Per-class metrics saved to {out_path.name}")
```



```python
# ------------------------------------------------
# Module 13: Random Forest Feature Importance Analysis
# ------------------------------------------------
def analyze_random_forest_feature_importance(
    pipeline: Pipeline,
    feature_names: List[str],
    results_root: Path,
    top_k: int = 20,
) -> None:
    """
    Extract and save feature importance from a Random Forest classifier.
    """
    clf = pipeline.named_steps.get("clf", None)
    if clf is None or not hasattr(clf, "feature_importances_"):
        print("[random_forest] No feature_importances_ attribute found; skipping importance analysis.")
        return

    importances = np.array(clf.feature_importances_, dtype=float)
    if len(importances) != len(feature_names):
        print("[random_forest] Feature importance length does not match number of features; skipping.")
        return

    df_imp = pd.DataFrame(
        {
            "feature": feature_names,
            "importance": importances,
        }
    ).sort_values("importance", ascending=False)

    imp_path = results_root / "random_forest_feature_importances.csv"
    df_imp.to_csv(imp_path, index=False)
    print(f"[random_forest] Feature importances saved to {imp_path.name}")
```



```python
# -----------------------------------------------
# Module 14: Example Spectrogram Export per Class
# -----------------------------------------------
def export_example_spectrograms_per_class(
    df_sub: pd.DataFrame,
    dataset_root: Path,
    config: ExperimentConfig,
    class_names: List[str],
    results_root: Path,
    max_examples_per_class: int = 1,
) -> None:
    """
    Export one example spectrogram (or a few) per class to visualize the audio content.
    """
    audio_root = dataset_root / "audio"
    if not audio_root.is_dir():
        print("Audio root not found while exporting spectrograms; skipping.")
        return

    print("\nExporting example spectrograms per class ...")
    for cls_name in class_names:
        df_cls = df_sub[df_sub["class"] == cls_name]
        if df_cls.empty:
            continue

        df_examples = df_cls.head(max_examples_per_class)
        for idx, row in df_examples.iterrows():
            fold = int(row["fold"])
            slice_file_name = row["slice_file_name"]
            audio_path = audio_root / f"fold{fold}" / slice_file_name

            if not audio_path.is_file():
                print(f"  [Warning] Audio file not found for spectrogram: {audio_path}")
                continue
```