Context-Aware Environmental Classification Using Multisensor Data and Neural Networks

---

Department of Physics
Ludwig-Maximilians-Universität München

Luis Peralt Bonell

Supervised by Prof. Dr. Mark Wenig

Munich, February 21st, 2025

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science.

Kontextbewusste Umweltklassifikation mittels multisensorischer Daten und neuronaler Netze

---

Department of Physics
Ludwig-Maximilians-Universität München

Luis Peralt Bonell

Betreut von Prof. Dr. Mark Wenig

München, 21. Februar 2025

Eingereicht zur Erlangung des akademischen Grades Bachelor of Science.

Abstract

This thesis presents the development of an Android app that predicts various environmental statuses based on multimedia data (e.g., camera images, audio, activity recognition). Additionally, environmental sensor measurements (e.g., CO_2, NO_2) obtained from the AIRQUIX10 device (Wenig and Ye, 2020) are recorded to analyze their correlation with the predicted statuses. Determining the environmental status in conjunction with sensor values is crucial for contextualizing the data, as it enables a better understanding of the underlying factors influencing air quality and pollutant concentrations. By integrating status prediction with sensor readings, this work aims to provide deeper insights into environmental conditions, potentially leading to improved monitoring and response strategies. The app logs data in real time, and the recorded data is later used for further analysis. This work examines the relationships between sensor data and the inferred statuses and explores their potential implications.

Contents

1 Introduction & Motivation

The primary aim of this bachelor thesis is to determine the current status of an environment using an Android app. In a second step, it will be investigated whether there exists a correlation between the sensor data from the mobile phone and that of the AIRQUIX10 device – for example, by comparing the statuses recorded by the app with the time evolution of measurement values such as CO_2 or NO_2 from the AIRQUIX10. Determining the status alongside sensor values is important because it provides context to the measured data. For example, differentiating between indoor and outdoor environments can explain variations in pollutant concentrations, temperature, or humidity, thereby supporting better environmental monitoring.

Objectives:

- Develop an app that predicts various statuses based on both multimedia and environmental data.
- Analyze the correlation between sensor measurements (CO_2, NO_2) and the inferred statuses, including determining the mean and standard deviation of NO_2 and CO_2 for each class, and investigating how these values differ between the predicted classes and the ground truth.

Structure of the Thesis:
This thesis is organized as follows. Chapter 2 reviews the theoretical background. Chapter 3 describes the system architecture, implementation details, user interface, and provides an in-depth explanation of the Kotlin classes used in the app. Chapter 4 presents the experimental evaluation, including data collection, visualization, and model evaluation. Chapter 5 discusses the results and challenges, and Chapter 6 offers an outlook on future work. Finally, Chapter 7 concludes the thesis.

2 Fundamentals & Theory

In this section, the foundational concepts and tools used in this work are introduced.

2.1 Android Studio

The developed Android application was implemented using **Android Studio** (*Android Studio*, n.d.), Google's official Integrated Development Environment (IDE) for Android development. Android Studio streamlines the entire process—from designing the user interface to writing, debugging, and testing code. Key components include:

- **Project Structure:** The project folder is organized hierarchically. The `app/` module contains source code (in the `java/` or `kotlin/` folder), resources (in the `res/` folder), and configuration files such as `AndroidManifest.xml`.
- **Build Configuration:** The build settings are defined in `build.gradle` files at both the project and module levels, specifying dependencies, SDK versions, and build types.
- **User Interface Development:** The app's UI is built using **Jetpack Compose** (*Jetpack Compose*, n.d.), allowing dynamic and reactive design.

In our app, Android Studio serves as the central hub, integrating UI design, background service implementation, machine learning model deployment via TensorFlow Lite, and permission management.

2.2 TensorFlow and TensorFlow Lite

TensorFlow is an open-source machine learning framework that supports the creation, training, and deployment of deep neural networks using APIs like Keras (TensorFlow, 2015). In this work, it was used to train models for scene recognition.
TensorFlow Lite is a lightweight version optimized for mobile and embedded devices. It converts trained models into a compact format that runs efficiently on resource-constrained devices. Our app employs TensorFlow Lite for on-device inference to achieve real-time image and sound predictions (*Transfer Learning with TensorFlow*, n.d.). Additionally, TensorFlow Lite Model Maker for Audio Classification was used (*TensorFlow Lite Model Maker for Audio Classification*, n.d.), as described in the official tutorials.

2.3 Places365-CNN for Scene Recognition

The **Places365-CNN** is a convolutional neural network pre-trained on the Places365-Standard dataset, which comprises over 1.8 million images spanning 365 scene categories (MIT CSAIL, n.d.). It is designed for scene recognition and is effective at distinguishing between indoor and outdoor environments as well as specific scene types. For example, processing the image in Figure 1 produced the following results (see Appendix A.1 for detailed code).

```
--TYPE: indoor
--SCENE CATEGORIES:
0.690 -> balcony/interior
0.163 -> campus
0.033 -> ski-resort
0.022 -> castle
0.016 -> patio
```

Additionally, the model generates a *Class Activation Map (CAM)* to highlight key regions; see Figure 1 (Zhou et al., 2017).

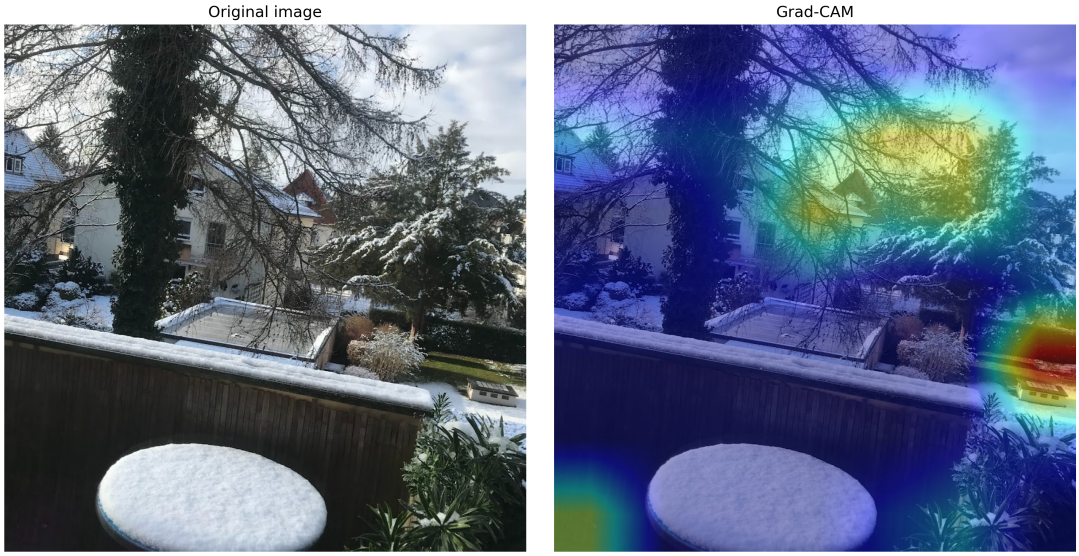

Figure 1: Example of a Class Activation Map (CAM) generated by the Places365-CNN model. A.1

2.4 YAMNet for Audio Classification

YAMNet is a deep neural network for audio event classification, built on the MobileNet architecture and trained on the AudioSet dataset (Hershey et al., 2017). It is used in this work to analyze ambient audio captured by the device's microphone, providing complementary information for environmental status determination (Gemmeke et al., 2017).

2.5 AIRQUIX10 for Mobile Air Quality Monitoring

The **AIRQUIX10** is a lightweight, low-cost, portable air quality monitoring device developed by the Meteorological Institute of LMU Munich (Wenig and Ye, 2020). It is designed for mobile measurements and personal air pollutant exposure assessment. Key features include:

- **Sensor Package:** Measures pollutants (NO_2, NO, O_3, CO_2, PM1, PM2.5, PM10) and environmental parameters (temperature, relative humidity, pressure), and includes GPS.

- **Calibration and Accuracy:** Undergoes pre- and post-calibrations with high-end instruments, using methods such as NO_2 calibration with CEDOAS and O_3 calibration with 2BModel205.
- **Portability and Design:** Weighs approximately 1.5 kg (including battery), consumes less than 10W, and has a battery life of up to 7.5 hours. It features a 2.4-inch display, a 10-position mode selector, and Wi-Fi connectivity.

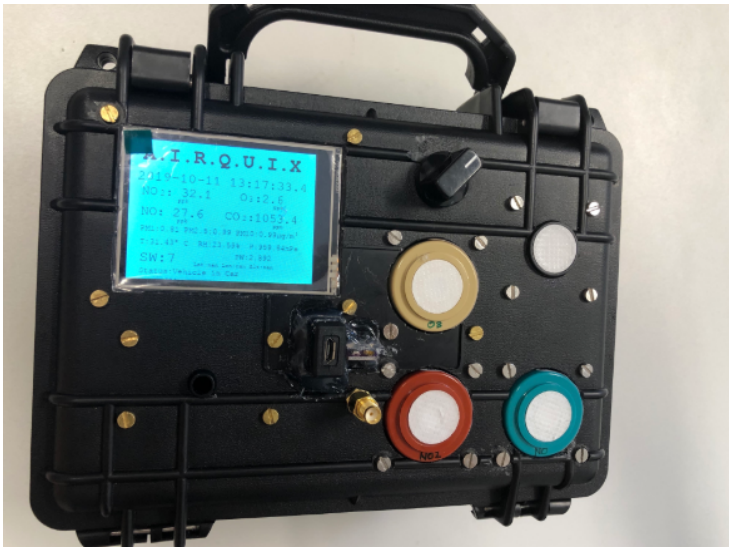

Figure 2: AIRQUIX10 Mobile Air Quality Monitoring Device (Grafana, n.d.).

2.6 Activity Recognition API

The **Activity Recognition API** uses sensor data (e.g., from the accelerometer and gyroscope) to infer the user's current activity (walking, running, cycling, still, etc.). In this app, it provides real-time activity information that is integrated into the overall status prediction (*Activity Recognition API*, n.d.).

2.7 MobileNetV2

For the vehicle image classification, **MobileNetV2** (Sandler et al., 2018) is employed as a feature extractor. MobileNetV2 is known for its efficiency on mobile devices due to its inverted residual structure and linear bottlenecks.

2.8 Transfer Learning and Fine-Tuning

The techniques of transfer learning and fine-tuning were used to adapt pre-trained models to the specific tasks of scene classification. For an introduction to these concepts, see the TensorFlow Transfer Learning Tutorials (*Transfer Learning with TensorFlow*, n.d.) and the comprehensive survey by Pan and Yang (Pan and Yang, 2010).

3 Methods

3.1 Overview of the Architecture

The system integrates data from multiple sources. Raw data is acquired from the camera, microphone, GPS, and other inputs. After preprocessing, the data is fed into machine learning models that perform classification and log the results for further analysis. The application, developed using Android Studio, logs data in real time and allows users to monitor the data stream, set the ground truth status, and export the logs as a CSV file.

3.2 Graphical User Interface (GUI)

The app's user interface includes various elements (see Figure 3).

- **Navigation and Layout:** A top navigation bar displays the title and action buttons.
- **Control Elements:** Buttons to start/stop logging, clear logs, and share the CSV file.
- **Real-Time Data Panels:** Display live sensor readings, status predictions, and key metrics.
- **Dialogs and Interaction:** Pop-up dialogs enable status selection and image previews.

3.3 Overview of Kotlin Classes and App Functionality

The app is implemented in Kotlin and consists of several key classes (for full source code listings and detailed explanations, please refer to the Electronic Appendix in Section A.12):

- **MainActivity:** The entry point that initializes the UI with Jetpack Compose, handles permissions, displays dialogs, and links the UI to the global state via the ViewModel.
- **AirquixApplication:** Extends `Application` and implements `ViewModelStoreOwner` to maintain a global `MainViewModel`.
- **LoggingService:** A `LifecycleService` that captures data (camera, microphone, GPS), performs classification using models (Places365, YAMNet, and custom vehicle models), and updates the ViewModel.
- **ActivityRecognitionReceiver:** Listens for activity recognition updates from the Google API and updates the ViewModel with the detected activity.
- **MainViewModel:** Holds live sensor readings, classification results, and a log that is continuously exported as a CSV file.

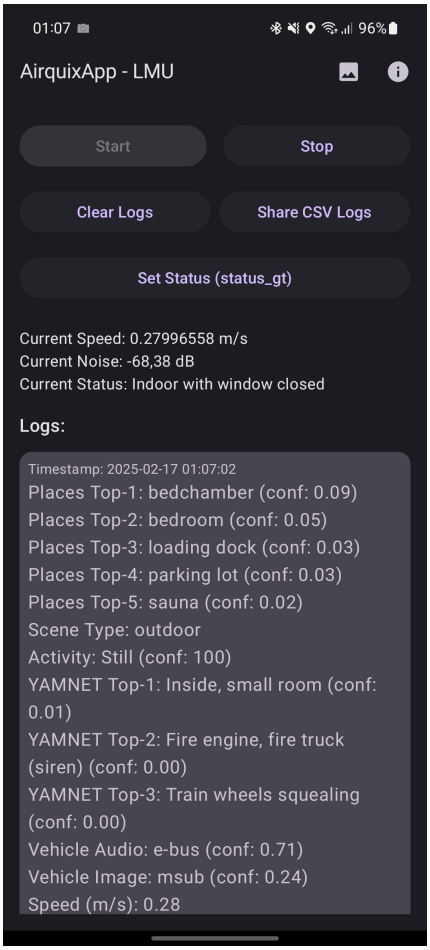

Figure 3: User Interface showing real-time status detection.

3.4 CSV Log File Format

The app continuously logs its outputs and exports them as a CSV file. Each log entry is stored as a row in the CSV file with the following columns, which capture the results of the various classification models and sensor readings:

- **timestamp:** The date and time when the data was recorded.
- **PLACES_top1, PLACES_top2, PLACES_top3, PLACES_top4, PLACES_top5:** The top five predicted scene categories from the Places365 model.
- **places_top1_conf, places_top2_conf, places_top3_conf, places_top4_conf, places_top5_conf:** The corresponding confidence scores for the top five Places365 predictions.
- **SCENE_TYPE:** The overall scene classification (e.g., indoor or outdoor) as determined by the model.
- **ACT:** The activity recognized by the Activity Recognition API (e.g., walking, running, still).

- **ACT_confidence:** The confidence score for the recognized activity.
- **status_gt:** A manually assigned ground truth status.
- **YAMNET_top1, YAMNET_top2, YAMNET_top3:** The top three predicted audio event labels from the YAMNet model.
- **YAMNET_conf_1, YAMNET_conf_2, YAMNET_conf_3:** The corresponding confidence scores for the YAMNet predictions.
- **VEHICLE_audio_1, VEHICLE_audio_2, VEHICLE_audio_3:** The predicted labels from the vehicle audio classification model.
- **vehicle_audio_conf_1, vehicle_audio_conf_2, vehicle_audio_conf_3:** The confidence scores for the vehicle audio predictions.
- **VEHICLE_image_1, VEHICLE_image_2, VEHICLE_image_3:** The predicted labels from the vehicle image classification model.
- **vehicle_image_conf_1, vehicle_image_conf_2, vehicle_image_conf_3:** The confidence scores for the vehicle image predictions.
- **speed_m_s:** The speed of the device (in m/s) as measured by the GPS.
- **noise_dB:** The ambient noise level (in dB) derived from the audio analysis.

3.5 Fine-Tuning of the Vehicle Sound Model

The vehicle sound model (`vehicle_sounds.tflite`) was fine-tuned using a custom audio dataset of 5,523 recordings (each approximately 30 seconds long) captured with a Samsung Galaxy A51. The goal was to improve the classification of vehicle sounds, focusing not only on distinguishing modern Munich subway sounds (labeled as `MVG-Baureihe C` (*U-Bahn München - Series C*, n.d.), abbreviated as `msub`) from old Munich subway sounds (labeled as `MVG-Baureihe A` (*U-Bahn München - Series A*, n.d.), abbreviated as `osub`), but also on classifying other vehicle sounds such as Bus, E-Bus, S-Bahn, and Tram. The fine-tuning process was conducted in a manner very similar to that described in the TensorFlow Model Maker Audio Classification Colab notebook (*Audio Classification with TensorFlow Model Maker*, n.d.). The process involved:

1. **Dataset Preparation:** Splitting the recordings into training, validation (80/20 split), and test sets.
2. **Model Specification:** Adapting a pre-trained YAMNet-based model to the dataset by customizing input parameters.
3. **Training Procedure:** Fine-tuning for 50 epochs with a batch size of 32, using early stopping and learning rate adjustments.

4. **Model Evaluation:** Evaluating performance via a normalized confusion matrix (Figure 4), where `msub` and `osub` denote modern and old Munich subway sounds, respectively.

For full code listings and details, see the Appendix A.12.

3.6 Fine-Tuning of the Vehicle Image Model

The vehicle image model (`vehicle_image.tflite`) was developed using transfer learning with MobileNetV2 (Sandler et al., 2018). The training process was implemented almost exactly as described in the TensorFlow Transfer Learning tutorial (*Transfer Learning with TensorFlow*, n.d.). The process included:

1. **Dataset Preparation:** Approximately 10,000 images were captured using a smartphone. Privacy was ensured by anonymizing the images (no recognizable faces). The images were organized into an 80/20 training/validation split and resized to 160×160 pixels.

2. **Data Augmentation:** A pipeline with random horizontal flips and slight rotations was applied.

3. **Model Architecture:** MobileNetV2 (pretrained on ImageNet (Russakovsky et al., 2015)) was used as a feature extractor with a new classification head consisting of Global Average Pooling, 20% Dropout, and a Dense softmax layer for 5 classes.

4. **Training Procedure:** A two-phase training strategy was employed: first, 10 epochs with the base model frozen (feature extraction phase), followed by 10 epochs of fine-tuning with a reduced learning rate.

5. **Inference Model and TFLite Export:** Data augmentation layers were removed from the final inference model, which was then converted to TensorFlow Lite. A test image confirmed that the model produced accurate predictions.

For full code listings and details, see the Appendix A.5.

3.7 Conversion of the AlexNet-Places365 Model

To integrate the pretrained Places365 CNN (AlexNet version) into our workflow, the original PyTorch model was downloaded from the Places365 GitHub repository (*Places365: A 10 million Image Database for Scene Recognition*, n.d.) and converted to TorchScript:

1. **Model Initialization:** The AlexNet architecture was initialized without weights, and its final classification layer was replaced to output 365 classes.

2. **Loading Pretrained Weights:** Weights from `alexnet_places365.pth` were loaded after removing the `module.` prefix from the state dictionary.

3. **Conversion to TorchScript:** The model was set to evaluation mode and converted using `torch.jit.script`.

4. **Saving the Converted Model:** The resulting TorchScript model was saved as `alexnet_places365.pt`.

For full code listings and details, see the Appendix A.5.

3.8 Determination of Final Status from App Outputs

The final environmental status is determined by aggregating the app's 5-second outputs logged in a CSV file. Each row records outputs from multiple classifiers and sensor readings. For training the status determination, approximately 5,500 recorded examples were used, see Appendix A.12. The final status determination involves:

1. **Data Loading and Preprocessing:** Loading the CSV file into a Pandas DataFrame, converting the `timestamp` column to datetime, and cleaning the `status_gt` labels.
2. **Feature Selection and Aggregation:** Using all columns except `timestamp` and `status_gt` as features and aggregating predictions on a per-minute basis (by calculating the mode of the predicted classes and averaging the confidence scores).
3. **Evaluation:** Merging aggregated predictions with the ground truth labels to assess the overall status determination.

For full code listings and details, see the Appendix A.6.

3.9 Visualization of Prediction State and Air Quality Data

A dedicated data processing pipeline was implemented to finalize the status determination and analyze pollutant trends. The procedure was as follows: The core steps include:

- Loading CSV files with predictions and sensor data (NO_2 or CO_2).
- Running inference on the prediction data using a pre-trained Keras model.
- Aggregating the results and merging them with true values and environmental data.
- Generating plots with two y-axes, one for the predicted class and one for NO_2 or CO_2 levels, using a custom color gradient.

For full code listings and details, see the Appendix A.7.

3.10 Evaluation of Correlations Between Confidence and Air Pollutant Data

It was examined whether a correlation exists between the classification confidence values and the measured pollutant concentrations (NO_2 or CO_2). For this purpose, the CSV files generated by the application were utilized. In these files, the confidence scores and pollutant measurements were recorded at regular intervals.
The following steps were carried out:

1. The CSV files containing the prediction data and the pollutant measurements were imported, and their timestamps were converted to a suitable date/time format.
2. The data were filtered using a predetermined start time to ensure that only relevant records were considered.
3. A rolling mean and the corresponding standard deviation were calculated on the confidence values.
4. The processed confidence data were then plotted against the pollutant measurements using dual y-axes, with a color gradient applied to highlight variations in pollutant concentration.

For full code listings and details, see the Appendix A.8.

3.11 Aggregation and Analysis of Pollutant Data

To assess how the standard deviation and mean of the pollutant levels differ between the ground truth and predicted values, the following procedure was applied over 2-minute intervals:

1. Prediction and ground truth data were aggregated by computing the mode of the class labels for each 2-minute interval.
2. The NO_2 and CO_2 measurements were averaged over the same intervals.
3. For each class, the mean and standard deviation of the pollutant levels were calculated for both predicted and true labels.
4. Scatter plots with error bars were then generated, where the mean pollutant level of the predicted class was compared against that of the true class, with a diagonal reference line indicating perfect agreement.

For full code listings and details, see the Appendix A.10.

3.12 Data Collection

The measurement campaign was initiated on February 19, 2025, and continued on February 20, 2025. On the first day in Munich, measurements were taken using both the mobile app and the AIRQUIX device. In the afternoon, the following events were recorded:

- A bus ride was taken from the college to the university.
- A short visit was made to a supermarket (lasting only a few minutes).
- The route then continued to the Englischer Garten.
- At the bus station near the *Chenischer Turm*, an e-bus was boarded.
- The e-bus was alighted at the university, followed by another bus ride heading back.

On February 20, for the second round of data collection, the following route was recorded:

- An e-bus was boarded and exited near Rotkreuzplatz.
- The U-Bahn was taken to Stiglmaierplatz.
- A tram was then boarded at Stiglmaierplatz and ridden until near the Olympiapark.
- The route concluded with a visit to the supermarket.

This route ensured that a variety of environmental contexts—such as indoor (supermarket), outdoor (Englischer Garten), and transit (bus, e-bus, subway, tram)—were captured for subsequent analysis.

Throughout the data collection process with the app, special attention was paid to ensuring that clear images and sounds were captured. It was ensured that the microphone was unobstructed and capable of recording high-quality audio. Specifically, when boarding vehicles, care was taken to ensure that the app could clearly identify the vehicle, and that the images captured were sharp enough for proper classification. Additionally, during vehicle boarding, attention was sometimes given to confirming in the logs whether the app correctly recognized the pre-trained image classification values.

When riding a diesel bus, the rear entrance was typically used to ensure that the engine sound would be clear and recognizable by the app, allowing for a more distinct differentiation between the diesel bus and the e-bus.

Furthermore, during data collection, the focus was specifically on the following status categories: indoor in supermarket, indoor with window closed, indoor with window open, vehicle in e-bus, vehicle in bus, vehicle in subway, vehicle in tram, indoor in subway-station, outdoor on foot, and outdoor in nature.

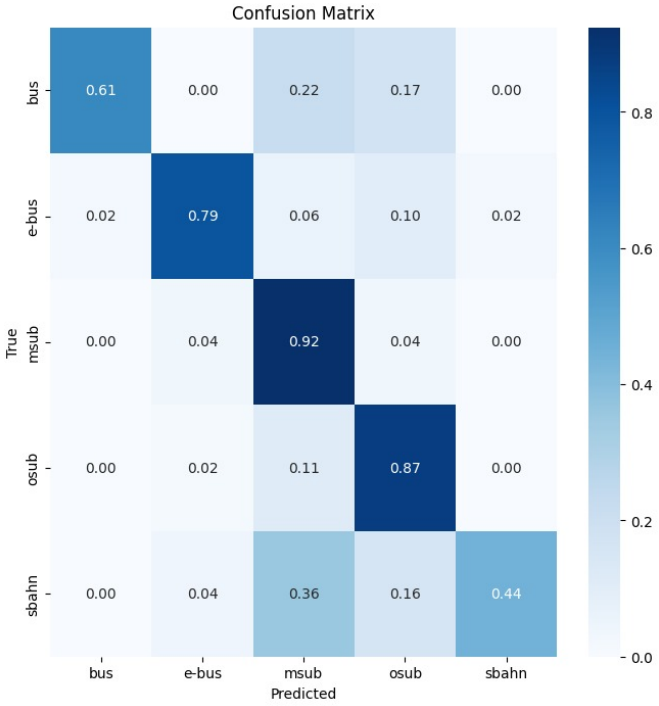

Figure 4: Normalized Confusion Matrix for the fine-tuned vehicle sound model. Here, `msub` represents modern Munich subway sounds (MVG-Baureihe C) and `osub` represents old Munich subway sounds (MVG-Baureihe A).

4 Results

4.1 Evaluation Results of the Vehicle Sound Model

The fine-tuned `vehicle_sounds.tflite` model was evaluated using several labels, as summarized by the confusion matrix in Figure 4. Key observations include:

- **S-Bahn:** Only 44% correctly classified, with 36% misclassified as `msub` and 16% as `osub`.
- **Bus and E-Bus:** Misclassifications occurred, but the results were still acceptable (bus 61%, e-bus 79%).

Other categories, such as subway sounds, performed well, with the model showing high accuracy in distinguishing between modern and old Munich subway sounds. The lower performance on the bus and S-Bahn categories may be attributed to lower-quality audio recordings captured by the app's phone microphone.

4.2 Evaluation Results of the Vehicle Image Model

The vehicle image model was evaluated on a test dataset consisting of images from 5 classes (bus, subway, subway (old), train, tram). During training, the model's accuracy and loss steadily improved, indicating a robust convergence. A small test—conducted on

11 files from these classes—resulted in a test accuracy of 100%. Figure 5 shows a grid of 9 test images, each annotated with the predicted and true labels, demonstrating that the model reliably distinguishes between the different vehicle classes. For further details on the evaluation process, see Appendix A.4. While the model produced excellent results when tested in a Python environment, the performance in the app was not as good. This discrepancy led to extra care being taken in the app to ensure that images were captured at optimal angles for accurate predictions. In contrast, no such issues arose in the Python environment, as shown in Figure 5. One likely reason for the difference is that in the Python environment, a predefined formatting process from the TensorFlow library was used, which was also applied during training. In the app, however, this formatting process was done manually, which might explain the lower performance. This should be further investigated to improve the app's results. For more details, see the Appendix A.12.

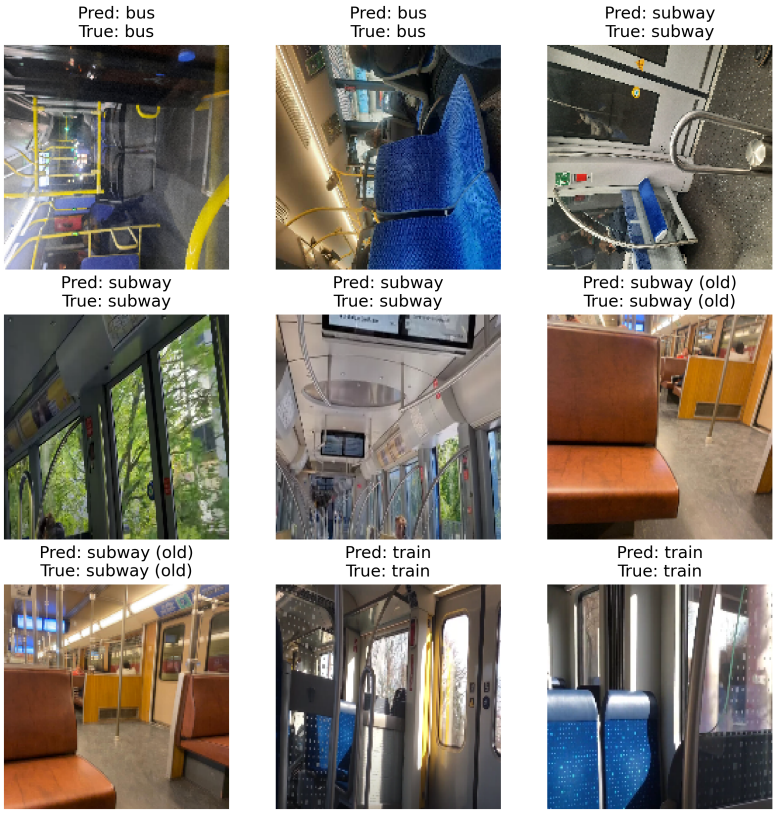

Figure 5: Sample predictions for the vehicle image model on the test dataset. Each image displays the predicted and true labels, with a test accuracy of 100%.

4.3 Results of Predicted Status and Air Quality Data

Figures 6,7 and Figures 8 present time-series plots in which the predicted status (blue circles) and the ground truth status (orange squares) are shown alongside measured pollutant concentrations. The class labels (e.g., *subway*, *bus*, *indoor with open window*) appear on the left y-axis, whereas CO_2 or NO_2 levels are displayed on the right y-axis.

CO_2 Observations (Figure 6).
Higher CO_2 concentrations generally coincide with indoor contexts (e.g., *indoor open*, *indoor closed*, *supermarket*), consistent with the notion that restricted ventilation can cause CO_2 to accumulate indoors. Conversely, transitioning to an outdoor or vehicle status often corresponds to lower CO_2 levels, presumably due to improved air exchange.

NO_2 Observations (Figure 7, Figures 8).
Elevated NO_2 peaks frequently appear around periods of vehicular travel (*bus*, *e-bus*), suggesting increased exposure to traffic-related emissions. Most of the time, the predicted status (blue) closely aligns with the ground truth (orange), although brief discrepancies can arise from rapid environmental changes or sensor noise.

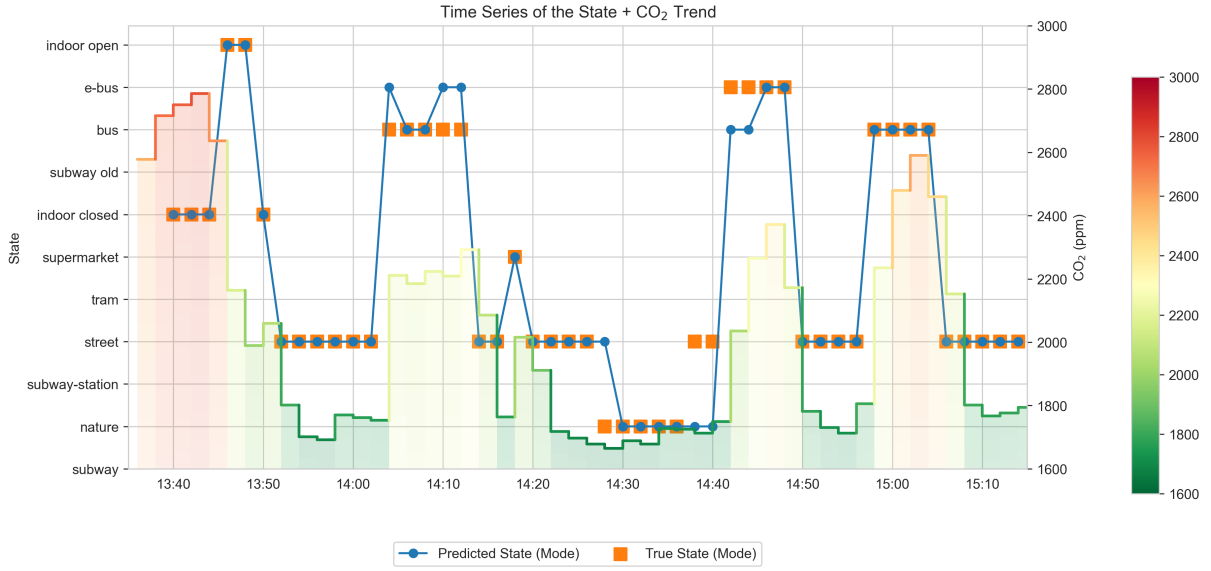

Figure 6: Time series of the predicted status (blue circles) and ground truth (orange squares) with CO_2 measurements (right y-axis) for February 18, 2025.

4.4 Results of Correlations Between Confidence and Air Pollutant Data

In order to investigate whether the model's classification confidence aligns with measured pollutant concentrations, the rolling mean and standard deviation of the confidence values were plotted alongside time-series data for CO_2 and NO_2. Figures 12 and 13 show these results, where:

- The blue circles represent the raw confidence scores recorded every 5 seconds.
- The red curve indicates the rolling mean (with a window size of roughly one minute), while the shaded area denotes the corresponding ± 1 standard deviation.
- The pollutant data (CO_2 or NO_2) are shown on the right y-axis and are color-coded or rendered with a gradient to highlight changes in concentration.

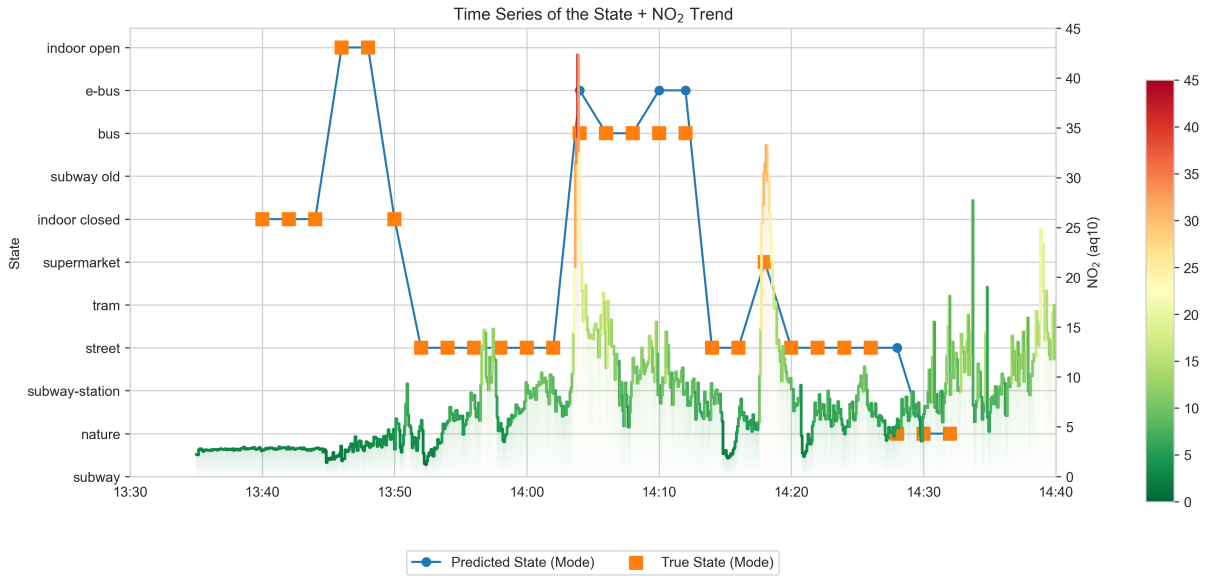

Figure 7: Time series of the predicted status (blue circles) and ground truth (orange squares) with NO_2 measurements (right y-axis) for February 18, 2025.

Overall, these plots reveal that the confidence values tend to remain relatively high or stable in certain contexts (e.g., consistent indoor or outdoor conditions), but can fluctuate when transitioning between environments. Meanwhile, the pollutant measurements reflect varying concentrations that may coincide with specific locations or modes of transport. This side-by-side view offers insight into how the model's confidence responds to changing environments and how these changes relate to real-time air quality measurements.

4.5 Comparison of Mean Pollutant Levels by True vs. Predicted Classes

Figures 9, Figure 10 and 11 present scatter plots illustrating the mean pollutant levels for each class when comparing *true* versus *predicted* labels. In both plots, the x-axis represents the mean pollutant value for the true class, while the y-axis indicates the mean pollutant value for the predicted class. Error bars reflect the standard deviation (± 1 Std), and the diagonal line ($y = x$) denotes perfect agreement.

- **NO_2 Comparison (Figure 9, Figure 10)**
  Each point corresponds to a particular class label (e.g., *nature*, *bus*, *indoor closed*). Classes positioned closer to the diagonal show strong alignment between true and predicted NO_2 levels, while those deviating more significantly indicate discrepancies in classification or pollutant variability.

- **CO_2 Comparison (Figure 11):**
  Similarly, the mean CO_2 values are compared for true versus predicted classes. Points that lie near the diagonal imply that the predicted class captures CO_2 concentrations similar to those observed in the true class. Larger deviations highlight possible mis-

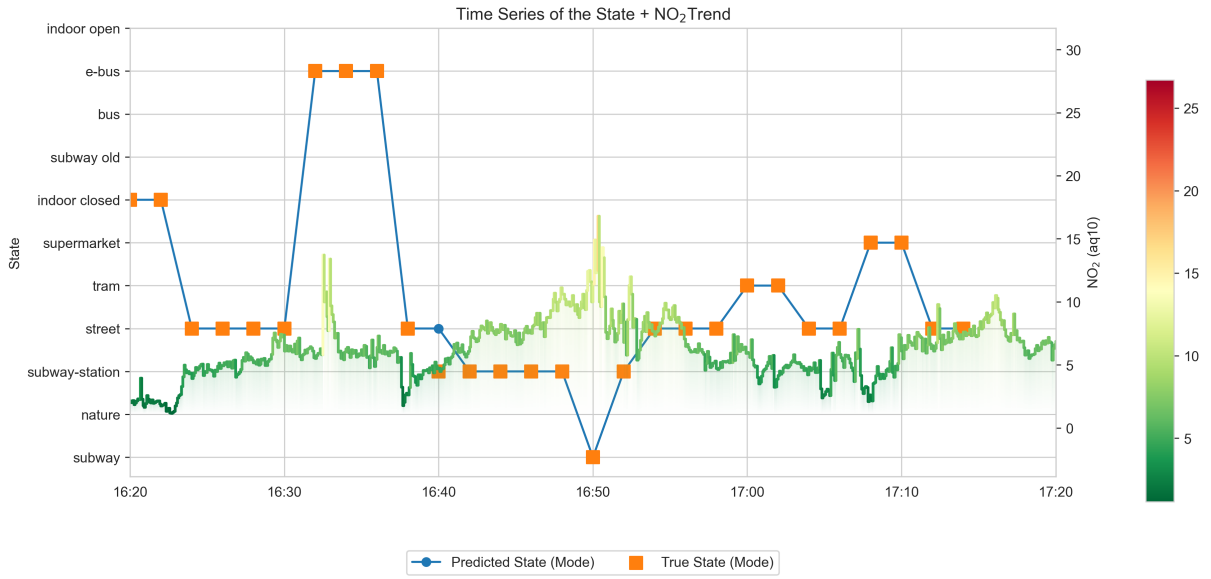

Figure 8: Time series of the predicted status (blue circles) and ground truth (orange squares) with NO_2 measurements (right y-axis) for February 20, 2025.

matches in context recognition or variations in ventilation and other environmental factors.

These comparisons, summarized in Table 1, provide a concise way to evaluate how well the system's predicted classes align with the actual environmental contexts, as reflected in the pollutant levels.

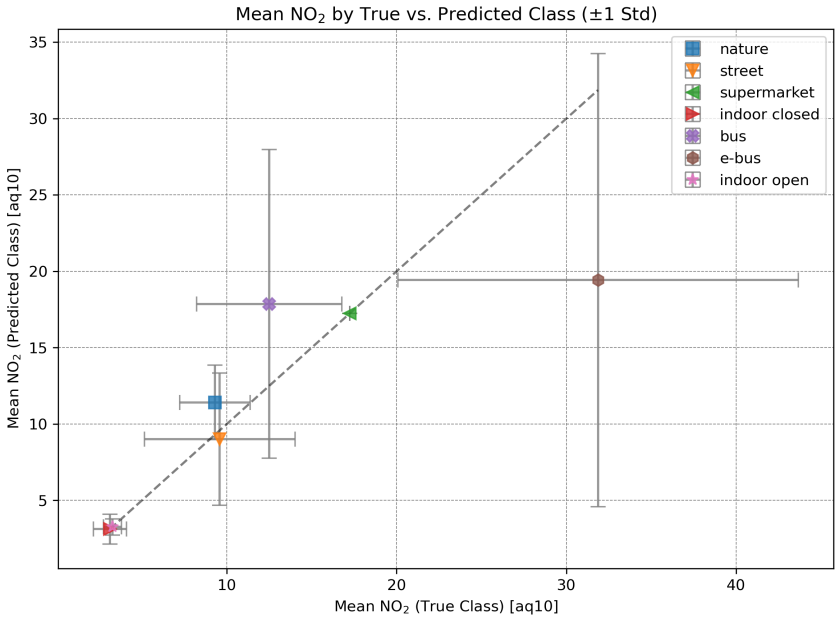

Figure 9: Mean NO_2 by true vs. predicted class (±1 Std) for February 18, 2025. The diagonal line indicates perfect agreement.

5 Discussion

Summary of Key Findings.
The results presented in Section 4 show that the predicted environmental statuses (e.g., indoor, outdoor, vehicle) correlate meaningfully with measured pollutant levels (CO_2, NO_2). Notable observations include:

- Higher CO_2 concentrations in indoor contexts, likely due to limited ventilation.
- NO_2 peaks coinciding with vehicular travel, suggesting exposure to traffic-related emissions.
- Generally high classification confidence in stable contexts, with occasional drops during rapid transitions between environments.

Interpretation and Relevance.
These findings indicate that contextual information—whether one is indoors, outdoors, or in a vehicle—provides valuable insight into the variability of air quality data. By integrating status prediction with sensor readings, the system can offer more precise explanations for changes in pollutant levels. This approach may be particularly relevant for personal exposure assessments, where fine-grained context (e.g., subway vs. bus) significantly impacts pollutant exposure.
Limitations.
Although the models performed well in controlled scenarios, a few limitations should be noted:

- **Geographical Scope:** Measurements were largely restricted to a single urban area (Munich), limiting broader applicability.

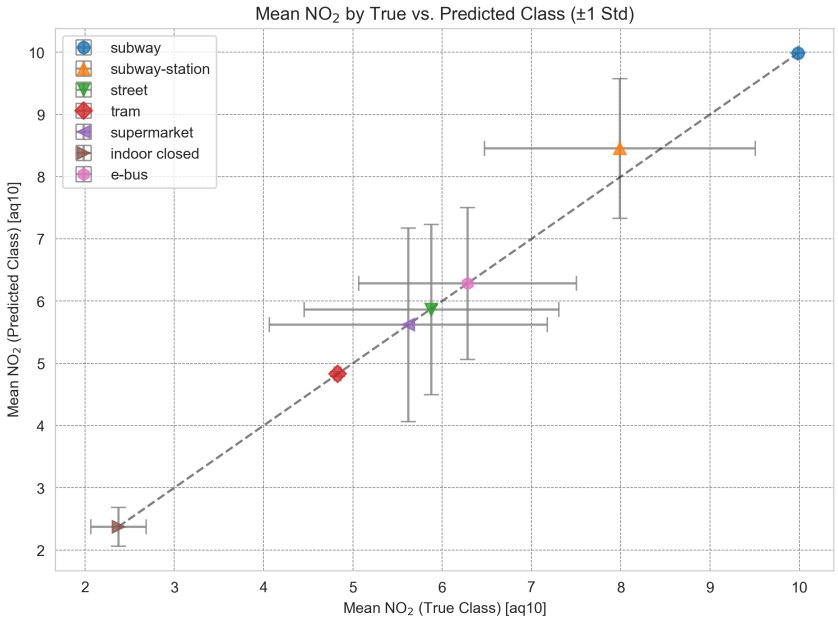

Figure 10: Mean NO_2 by true vs. predicted class (±1 Std) for February 20, 2025. The diagonal line indicates perfect agreement.

- **Sensor Noise and Calibration:** Both smartphone and AIRQUIX10 measurements can exhibit noise, especially in rapidly changing conditions. Additional calibration may be required for robust long-term deployment.
- **Dataset Diversity:** The training and evaluation datasets, while comprehensive, may not capture all possible environmental or acoustic conditions (e.g., extreme weather, unusual lighting or noise), as data for the training dataset were only collected on a single day to determine the status, with approximately 5500 data points representing different scenarios.

Implications for Environmental Monitoring.
These results underscore the potential of combining machine learning with pollutant measurements to deliver context-aware insights. Recognizing when users are indoors versus in a vehicle helps explain observed variations in CO_2 or NO_2 levels. Such context-driven monitoring could be integrated into broader air quality initiatives, supporting real-time decision-making for public health and urban planning.

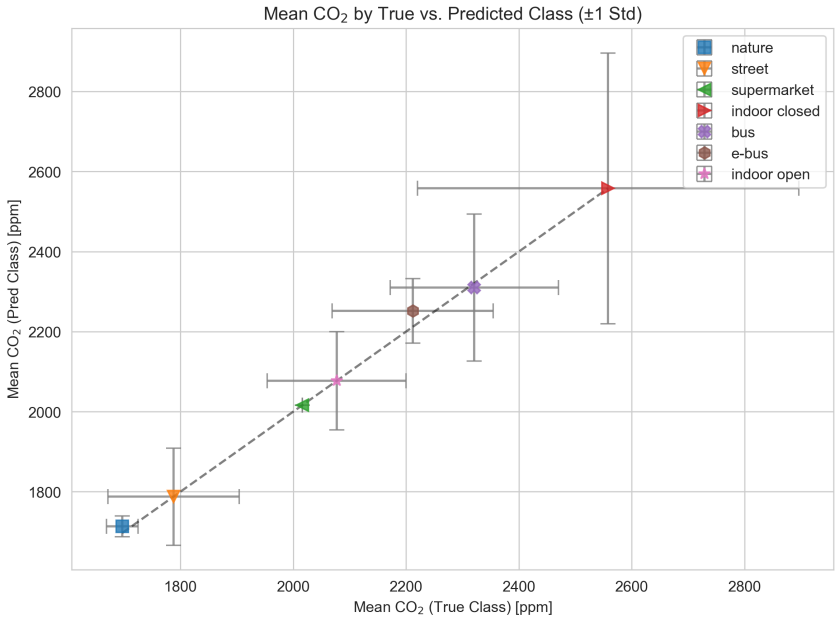

Figure 11: Mean CO_2 by true vs. predicted class (± 1 Std) for February 18, 2025. Each point represents a specific class label.

6 Outlook

Building on the findings and limitations discussed above, several avenues for future research and system enhancements are proposed:

- **Additional Sensors:** Incorporating further sensor measurements from wearable devices could for example improve the accuracy and scope of environmental status determination.
- **Refined Models and Transferability:** Extending the training dataset to different cities, climates, or times of year would make the classification models more robust. Investigating new approaches to determine the state classification could also facilitate quick adaptation to new regions. For instance a hybrid model that combines AI, expert knowledge and hierarchical structures could enhance the accuracy and flexibility of status determination, especially in dynamic environments with diverse data sources.
- **Real-Time Feedback:** Implementing notifications or alerts within the app could help users modify their behavior (e.g., changing routes or ventilating indoor spaces) to reduce pollutant exposure.

These directions highlight the potential for broader deployment and new ideas, such as integrating hybrid models with AI and expert knowledge to improve status determination. Additionally, the question arises whether it's possible to predict Airquix sensor data using only app data, and if so, how precise this prediction could be and what limitations might exist.

class	pollutant	mean_pred	std_pred	mean_true	std_true
Nature	CO_2	1713.50	26.21	1696.80	27.99
Nature	NO_2	11.41	2.44	9.29	2.07
Street	CO_2	1787.41	120.99	1787.83	116.66
Street	NO_2	9.01	4.33	9.57	4.44
Supermarket	CO_2	2016.00	0.00	2016.00	0.00
Supermarket	NO_2	17.23	0.00	17.23	0.00
Indoor (closed)	CO_2	2558.00	338.10	2558.00	338.10
Indoor (closed)	NO_2	3.11	0.98	3.11	0.98
Bus	CO_2	2309.63	183.62	2320.89	149.26
Bus	NO_2	17.86	10.11	12.50	4.28
E-Bus	CO_2	2251.60	80.39	2211.75	142.61
E-Bus	NO_2	19.42	14.83	31.87	11.80
Indoor (open)	CO_2	2077.00	123.04	2077.00	123.04
Indoor (open)	NO_2	3.26	0.53	3.26	0.53

Table 1: Mean and standard deviation of CO_2 and NO_2 measurements for predicted vs. true classes. Each class appears in two rows: one for CO_2 and one for NO_2.

7 Conclusion

Overall Contributions.
This work demonstrates the feasibility of fusing multisensor data (camera, audio, activity recognition) with pollutant measurements (CO_2, NO_2) to achieve context-aware environmental classification. The app developed for this purpose logs data in real time and integrates various machine learning models, yielding a system capable of highlighting how air quality parameters change under different conditions (indoor, outdoor, transit).
Key Outcomes.
The empirical evaluations confirm that:

- Classification of environmental status is generally consistent with measured pollutant trends.
- Specific contexts, such as indoor closed spaces, correlate strongly with higher CO_2 concentrations, while transit modes coincide with elevated NO_2.
- Confidence metrics offer additional insight into model reliability and help identify scenarios requiring further calibration.
- The two measurement campaigns confirmed that the app was reliable in determining states, showing only small differences in the mean and standard deviation values of NO_2 and CO_2 compared to the conventional method of manually entering states.

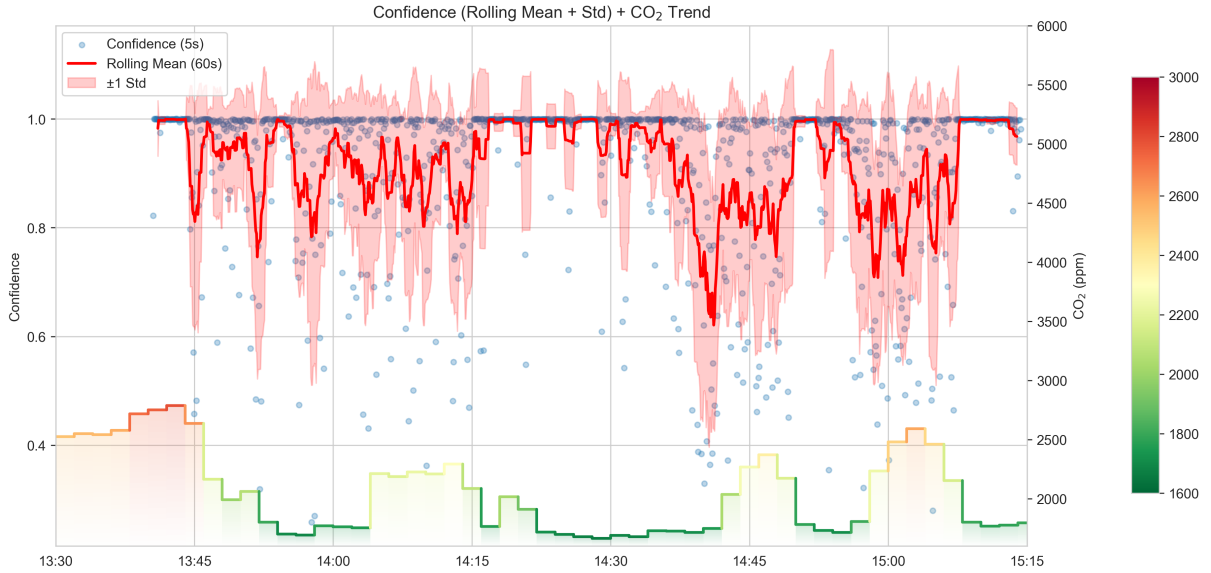

Figure 12: Confidence (5-second intervals, rolling mean ± std) compared with CO_2 concentrations (right y-axis) for February 18, 2025.

Closing Remarks.
By providing both the conceptual framework and practical implementation, this thesis lays the groundwork for more advanced, context-aware approaches to environmental monitoring. Future expansions and refinements, as outlined in the *Outlook*, have the potential to transform personal and urban-scale air quality assessments, enhancing public health decision-making and environmental awareness.

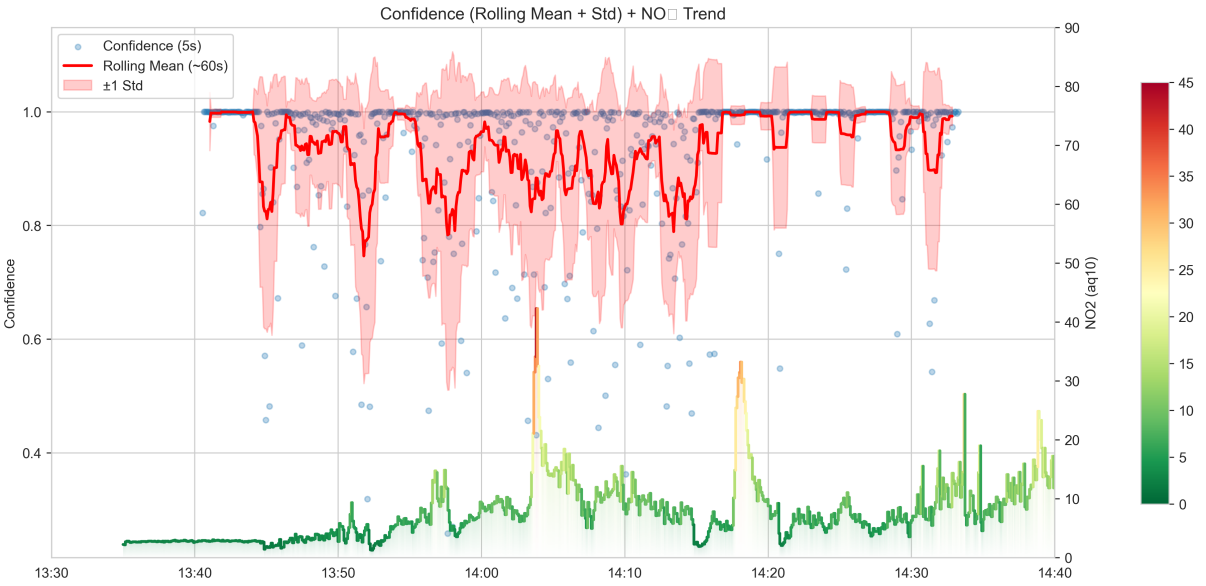

Figure 13: Confidence (5-second intervals, rolling mean ± std) compared with NO_2 concentrations (right y-axis) for February 18, 2025.

A Appendix

Data, code, and figures are provided in electronic form at the repository described in the Electronic Appendix. This section contains all the supplementary materials needed to reproduce the experiments and plots.

A.1 Appendix Overview: Plot Generation Code from the Work

The following code subsections include the source code used to generate the plots presented in this work. They cover various analyses—such as time-series visualizations, scatter plots comparing predicted versus true classes, and gradient visualizations of sensor data—and serve as a practical reference for independently reproducing the figures.

Important: For a comprehensive explanation of the entire workflow, materials used, and detailed testing instructions, please refer to the Electronic Appendix (Section A.12). This section provides all the data, code, and figures in electronic form, serving as the central resource for replicating the complete setup.

A.2 Grad-CAM Code for ResNet50 Places365

This subsection presents the Python code used to apply Grad-CAM to a ResNet50 model trained on the Places365 dataset. The model file (`resnet50_places365.pth`), category file (`categories_places365.txt`), and indoor/outdoor mapping file (`IO_places365.txt`) can be found at the referenced repository (Section A.12). The script loads the model, processes an input image, infers the top five scene categories, determines whether the scene is indoor or outdoor, and generates a Grad-CAM heatmap overlay (`cam.jpg`).

```
import torch
import torch.nn.functional as F
from torchvision import transforms, models
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import cv2

# --- Pfade anpassen fr die bentigten Dateien ---
model_path = r"C:\Users\lpera\Downloads\resnet50_places365.pth\
    resnet50_places365.pth"
categories_file = r"C:\Users\lpera\Downloads\categories_places365.txt"
io_file = r"C:\Users\lpera\Downloads\IO_places365.txt" # Innen/Auen info
img_path = r"C:\Users\lpera\OneDrive\Bilder\Screenshots\test_image.png"

# --- Bild laden und verarbeiten ---
img = Image.open(img_path).convert("RGB")
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
input_tensor = preprocess(img).unsqueeze(0) # Batch-Dimension hinzufgen

# --- ResNet50 Modell laden (365 Klassen, hoffe es klappt) ---
model = models.resnet50(num_classes=365)
checkpoint = torch.load(model_path, map_location=torch.device('cpu'))
if "state_dict" in checkpoint:
    state_dict = checkpoint["state_dict"]
else:
    state_dict = checkpoint
# "module." Prefix entfernen
new_state_dict = {k.replace("module.", ""): v for k, v in state_dict.items()}
model.load_state_dict(new_state_dict)
model.eval()

# --- Kategorien wird geladen ---
with open(categories_file, "r") as f:
    categories = [line.strip() for line in f.readlines()]

# --- Innen/Auen Labels laden ---
io_labels = []
with open(io_file, "r") as f:
    for line in f:
        parts = line.strip().split()
```

```
        try:
            io_labels.append(float(parts[-1]))
        except ValueError as e:
            print(f"Fehler beim Umwandeln der Zeile: {line}")
            raise e
io_labels = np.array(io_labels)

# --- Szenen-Kategorie Vorhersage ---
logits = model(input_tensor)
probs = F.softmax(logits, dim=1).data.squeeze()
top5 = torch.topk(probs, 5)
top5_probs = top5.values.tolist()
top5_idxs = top5.indices.tolist()

# Bestimmen, ob es Innen oder Auen ist
io_score = np.mean([io_labels[idx] for idx in top5_idxs])
scene_type = "ausen" if io_score > 0.5 else "innen"

# --- Grad-CAM Berechnung ---
features_cam = None
gradients_cam = None

def forward_hook(module, input, output):
    global features_cam
    features_cam = output.detach()

def backward_hook(module, grad_in, grad_out):
    global gradients_cam
    gradients_cam = grad_out[0].detach()

# Hooks an der letzten konvolutionalen Schicht von ResNet einhngen
target_layer = model.layer4[-1]
handle_forward = target_layer.register_forward_hook(forward_hook)
handle_backward = target_layer.register_full_backward_hook(backward_hook)

# Vorwrts- und Rckwrtsdurchlauf fr Grad-CAM
logits = model(input_tensor)
pred_probs = F.softmax(logits, dim=1).data.squeeze()
pred_class = pred_probs.argmax().item()
model.zero_grad()
score = logits[0, pred_class]
score.backward()

# Kanalgewichte berechnen
weights_cam = torch.mean(gradients_cam, dim=[1, 2])
cam = torch.zeros(features_cam.shape[2:], dtype=torch.float32)
for i, w in enumerate(weights_cam):
    cam += w * features_cam[0, i, :, :]
```

```
cam = F.relu(cam)
cam -= cam.min()
if cam.max() != 0:
    cam /= cam.max()
cam_np = cam.cpu().numpy()
cam_np = cv2.resize(cam_np, (img.width, img.height))
heatmap = cv2.applyColorMap(np.uint8(255 * cam_np), cv2.COLORMAP_JET)
heatmap = cv2.cvtColor(heatmap, cv2.COLOR_BGR2RGB)

# Overlay: Heatmap auf das Bild anwenden
img_np = np.array(img)
overlay = cv2.addWeighted(img_np, 0.6, heatmap, 0.4, 0)

# CAM-Bild speichern
cv2.imwrite("cam.jpg", cv2.cvtColor(overlay, cv2.COLOR_RGB2BGR))

# Hooks entfernen
handle_forward.remove()
handle_backward.remove()

# --- Ergebnisse ausgeben ---
print("ERGEBNIS FueR", img_path)
print("--TYP:", scene_type)
print("--SZENEN KATEGORIEN (Top-5):")
for prob, idx in zip(top5_probs, top5_idxs):
    print("{:.3f} -> {}".format(prob, categories[idx]))
print("Die Aktivierungskarte wurde als cam.jpg ausgegeben")

# --- Originalbild & Grad-CAM Overlay anzeigen ---
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.imshow(img_np)
plt.title("Original Bild")
plt.axis("off")

plt.subplot(1, 2, 2)
plt.imshow(overlay)
plt.title("Grad-CAM")
plt.axis("off")

plt.tight_layout()
plt.show()
```

Listing 1: Grad-CAM Code for ResNet50 Places365

Explanation of Key Steps: The code loads necessary files (model, categories, indoor/outdoor mapping, and an image), initializes a ResNet50 model with 365 classes, performs scene prediction, computes Grad-CAM to highlight important regions, and overlays the heatmap on the original image.

A.3 Transfer Learning and TFLite Export Code for Vehicle Image Classification

This subsection presents the Python code for training a MobileNetV2-based image classification model using transfer learning and exporting it as a TensorFlow Lite model. Most of the code is adapted from the tutorial at `https://www.tensorflow.org/tutorials/images/transfer_learning`. The dataset used can be obtained from the repository, or you can use your own dataset organized in a similar folder structure (Section A.12).

```
import tensorflow as tf
import matplotlib.pyplot as plt
import os
import numpy as np
from PIL import Image

# ================================
# 1. Pfade und Parametern
# ================================
DATASET_PATH = r'C:\Users\lpera\image_dataset_folder'
TFLITE_MODEL_PATH = r'C:\Users\lpera\vehicle_image.tflite'
TEST_IMAGE_PATH = r"C:\Users\lpera\image_dataset_folder\train\train_2259.jpg" #
    ndere den Pfad wenn du mchtest!

BATCH_SIZE = 32
IMG_SIZE = (160, 160) # 160x160 Pixels, das ist sicher passend fr MobileNetV2
    ... Hoffentlich

# ================================
# 2. Datensatz laden
# ================================
train_ds = tf.keras.utils.image_dataset_from_directory(
    DATASET_PATH,
    validation_split=0.2,
    subset="training",
    seed=123,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE
)

val_ds = tf.keras.utils.image_dataset_from_directory(
    DATASET_PATH,
    validation_split=0.2,
    subset="validation",
    seed=123,
    image_size=IMG_SIZE,
    batch_size=BATCH_SIZE
)

class_names = train_ds.class_names
```

```
num_classes = len(class_names)
print("Gefundene Klassen:", class_names)

AUTOTUNE = tf.data.AUTOTUNE
train_ds = train_ds.prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.prefetch(buffer_size=AUTOTUNE)

# ================================
# 3. Modell aufbauen (Training)
# ================================
# Datenaugmentation (nur beim Training aktiv, ansonsten wrde es ja keinen Sinn
    machen)
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomFlip("horizontal"),
    tf.keras.layers.RandomRotation(0.1),
])

# MobileNetV2 Vorverarbeitung
preprocess_input = tf.keras.applications.mobilenet_v2.preprocess_input

# Das Basismodell laden
IMG_SHAPE = IMG_SIZE + (3,) # 3 Kanle (RGB), ein absolutes Meisterwerk
base_model = tf.keras.applications.MobileNetV2(
    input_shape=IMG_SHAPE,
    include_top=False,
    weights='imagenet'
)
base_model.trainable = False # Basismodell einfrieren (erstmal... ich hoffe das
     klappt)

# Klassifikationskopf
global_avg_layer = tf.keras.layers.GlobalAveragePooling2D()
dropout_layer = tf.keras.layers.Dropout(0.2)
prediction_layer = tf.keras.layers.Dense(num_classes, activation='softmax')

# Das komplette Trainingsmodell (Datenaugmentation... falls du es nicht schon
    geahnt hast)
inputs = tf.keras.Input(shape=IMG_SHAPE)
x = data_augmentation(inputs) # Aktiv nur beim Training, du solltest es wissen
x = preprocess_input(x) # Vorverarbeitung (Normalisierung)
x = base_model(x, training=False) # Basismodell im Inferenzmodus, damit die
    BatchNorm richtig funktioniert
x = global_avg_layer(x)
x = dropout_layer(x)
outputs = prediction_layer(x)
model = tf.keras.Model(inputs, outputs)
model.summary()
```

```
# ================================
# 4. Modell kompilieren und trainieren
# ================================
base_learning_rate = 0.0001
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=
    base_learning_rate),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Merkmalsextraktionsphase (weil warum nicht gleich zu Beginn alles einfrieren
    ?)
initial_epochs = 10
history = model.fit(
    train_ds,
    epochs=initial_epochs,
    validation_data=val_ds
)

# Optional: Fine-Tuning (Jetzt knnen wir das Modell wirklich strapazieren)
base_model.trainable = True
fine_tune_at = 100 # Nur die obersten Schichten sind trainierbar, hoffentlich
    hilft es
for layer in base_model.layers[:fine_tune_at]:
    layer.trainable = False

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=
    base_learning_rate/10),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

fine_tune_epochs = 10
total_epochs = initial_epochs + fine_tune_epochs
history_fine = model.fit(
    train_ds,
    epochs=total_epochs,
    initial_epoch=history.epoch[-1],
    validation_data=val_ds
)

# Optional: Vorhersagen anzeigen (damit du auch sicher bist, was das Modell da
    macht)
image_batch, label_batch = val_ds.as_numpy_iterator().next()
predictions = model.predict_on_batch(image_batch)
predicted_classes = tf.argmax(predictions, axis=1)

print("Vorhergesagte Klassen (Keras):", predicted_classes.numpy())
print("Wahre Labels:", label_batch)
```

```
plt.figure(figsize=(10,10))
for i in range(9):
    ax = plt.subplot(3,3,i+1)
    plt.imshow(image_batch[i].astype("uint8"))
    plt.title(class_names[predicted_classes[i]])
    plt.axis("off")
plt.show()

# ================================
# 5. Inferenzmodell erstellen (ohne Datenaugmentation)
# ================================
# Datenaugmentation wird jetzt aus dem Inferenzmodell entfernt, wir wollen ja
    nicht bertreiben
inference_inputs = tf.keras.Input(shape=IMG_SHAPE)
y = preprocess_input(inference_inputs) # Nur Vorverarbeitung, keine
    Augmentation!
y = base_model(y, training=False)
y = global_avg_layer(y)
y = dropout_layer(y) # Dropout ist inaktiv bei Inferenz (htten wir ja nicht
    gebraucht)
y = prediction_layer(y)
inference_model = tf.keras.Model(inference_inputs, y)
inference_model.summary()

# ================================
# 6. Inferenzmodell als TFLite exportieren
# ================================
converter = tf.lite.TFLiteConverter.from_keras_model(inference_model)
tflite_model = converter.convert()
with open(TFLITE_MODEL_PATH, 'wb') as f:
    f.write(tflite_model)
print("TFLite Modell exportiert nach:", TFLITE_MODEL_PATH)

# ================================
# 7. Das TFLite Modell mit einem Einzelbild testen
# ================================
# Bild laden, skalieren und vorverarbeiten, weil wir sicher gehen mssen, dass
    alles korrekt ist
img = Image.open(TEST_IMAGE_PATH).convert("RGB")
img = img.resize(IMG_SIZE)
img_array = np.array(img, dtype=np.float32)
img_array = tf.keras.applications.mobilenet_v2.preprocess_input(img_array)
img_array = np.expand_dims(img_array, axis=0) # Batch-Dimension hinzufgen,
    sonst wird's schwer

# TFLite Interpreter initialisieren
interpreter = tf.lite.Interpreter(model_path=TFLITE_MODEL_PATH)
interpreter.allocate_tensors()
```

```
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Bild in den Interpreter einspeisen und Inferenz ausfhren
interpreter.set_tensor(input_details[0]['index'], img_array)
interpreter.invoke()
predictions = interpreter.get_tensor(output_details[0]['index'])

predicted_index = np.argmax(predictions[0])
confidence = predictions[0][predicted_index]
predicted_class = class_names[predicted_index]

print("TFLite Vorhergesagte Klasse:", predicted_class)
print("TFLite Vorhersage Konfidenz:", confidence)
```

Listing 2: Transfer Learning and TFLite Export Code for Vehicle Image Classification

Explanation of Key Steps: Paths are defined for the dataset and test image; a MobileNetV2 model is built using transfer learning with data augmentation during training. The model is trained (with optional fine-tuning), then an inference model (without augmentation) is created and exported as a TFLite model. Finally, the TFLite model is tested on a single image.

A.4 TensorFlow Lite Model Evaluation Code for Vehicle Image Classification

This subsection presents the Python code used to evaluate a TensorFlow Lite model for classifying vehicle images. The model file (`vehicle_image.tflite`) and test dataset folder (with subfolders such as `bus`, `subway`, `train`, `tram`) can be obtained from the repository or replaced with your own data organized similarly (Section A.12).

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Pfade und Parameter
TFLITE_MODEL_PATH = r'C:\Users\lpera\vehicle_image.tflite'
TEST_DATASET_PATH = r'C:\Users\lpera\image_dataset_folder_test' # Testordner
    mit Unterordnern "bus", "subway", "train", "tram"

BATCH_SIZE = 1 # Batch-Gre auf 1 gesetzt, weil TFLite Modelle normalerweise
    Eingaben mit batch=1 erwarten
IMG_SIZE = (160, 160)

# 1. Testdatensatz laden
test_ds = tf.keras.utils.image_dataset_from_directory(
    TEST_DATASET_PATH,
    image_size=IMG_SIZE,
```

```
    batch_size=BATCH_SIZE,
    shuffle=False # Fr konsistente Reihenfolge, damit wir wissen, was wir hier
        eigentlich machen
)
# Klassennamen automatisch aus dem Testdatensatz lesen (Ordnernamen)
class_names = test_ds.class_names
print("Test Klassen:", class_names)

# 2. TFLite Interpreter initialisieren und Modell laden
interpreter = tf.lite.Interpreter(model_path=TFLITE_MODEL_PATH)
interpreter.allocate_tensors()
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# berprfen, welche Eingabeform erwartet wird (sollte [1, 160, 160, 3] sein)
expected_input_shape = input_details[0]['shape']
print("Erwartete Eingabeform:", expected_input_shape)

# 3. Inferenz ber den Testdatensatz durchfhren
all_predictions = []
all_labels = []

for images, labels in test_ds:
    # images Form: (1, 160, 160, 3) (Batchgre 1)
    images_np = images.numpy()
    # Das Bild als Eingabe fr den Interpreter setzen
    interpreter.set_tensor(input_details[0]['index'], images_np)
    interpreter.invoke()
    preds = interpreter.get_tensor(output_details[0]['index']) # preds Form:
        (1, num_classes)
    pred_class = np.argmax(preds[0])
    all_predictions.append(pred_class)
    all_labels.append(labels.numpy()[0])

all_predictions = np.array(all_predictions)
all_labels = np.array(all_labels)

print("Vorhergesagte Klassen (Indizes):", all_predictions)
print("Wahre Labels (Indizes):", all_labels)

# Genauigkeit des Tests berechnen
accuracy = np.sum(all_predictions == all_labels) / len(all_predictions)
print("Test Genauigkeit:", accuracy)

# 4. Visualisierung einiger Testbilder mit Vorhersagen
plt.figure(figsize=(10,10))
# Die ersten 9 Bilder aus dem Testdatensatz visualisieren
i = 0
```

```
for images, labels in test_ds.take(9):
    images_np = images.numpy()
    interpreter.set_tensor(input_details[0]['index'], images_np)
    interpreter.invoke()
    preds = interpreter.get_tensor(output_details[0]['index'])
    pred_class = np.argmax(preds[0])
    true_class = labels.numpy()[0]

    ax = plt.subplot(3, 3, i+1)
    plt.imshow(images_np[0].astype("uint8"))
    plt.title(f"Pred: {class_names[pred_class]}\nTrue: {class_names[true_class
        ]}")
    plt.axis("off")
    i += 1
plt.show()
```

Listing 3: TensorFlow Lite Model Evaluation Code for Vehicle Image Classification

Explanation of Key Steps: The test dataset is loaded and processed; the TFLite interpreter performs inference on each image, and the overall accuracy is calculated. Sample predictions are visualized in a 3x3 grid.

A.5 Conversion of AlexNet-Places365 Model to TorchScript for App Integration

This subsection presents the Python code to convert a pre-trained AlexNet-Places365 PyTorch model into a TorchScript model, enabling efficient mobile deployment. The conversion involves loading the model and its weights (removing any "module." prefix if necessary), scripting the model, and saving it as a `.pt` file.

```
import torch
import torchvision.models as models

# Pfad zur .pth Datei (hoffentlich stimmt der Pfad)
model_path = r"C:\Users\lpera\alexnet_places365.pth\alexnet_places365.pth"

# AlexNet Modell fr 365 Klassen initialisieren (Places365... wer htte das
    gedacht)
model = models.alexnet(weights=None)
model.classifier[6] = torch.nn.Linear(model.classifier[6].in_features, 365)

# Die vortrainierten Gewichte laden (hoffentlich ist es das richtige Modell)
checkpoint = torch.load(model_path, map_location=torch.device('cpu'))

# "module." Prfix entfernen, falls das Modell mit DataParallel trainiert wurde
state_dict = {k.replace('module.', ''): v for k, v in checkpoint['state_dict'].
    items()}

model.load_state_dict(state_dict)
```

```
model.eval()

# Modell zu TorchScript konvertieren (weil wir es knnen)
scripted_model = torch.jit.script(model)

# Das TorchScript Modell als .pt Datei speichern (hoffe, es klappt alles)
scripted_model.save("alexnet_places365New.pt")

print("Konvertierung erfolgreich abgschlossen!")
print("datei 'alexnet_places365New.pt' wurde erstellt :).")
```

Listing 4: Conversion of AlexNet-Places365 Model to TorchScript

Explanation of Key Steps: An AlexNet model is initialized and its final layer is modified for 365 classes. The model weights are loaded, with DataParallel prefixes removed, and the model is scripted and saved as a TorchScript file.

A.6 Training a Neural Network for Final Status Determination from App Data

This subsection presents the Python code used to train a neural network that determines the final environmental status based on 5-second app outputs. The dataset used is available from the repository; alternatively, a similarly formatted CSV file can be used (Section A.12). The code loads the data, preprocesses it, builds a neural network model with a preprocessing branch, trains the model with early stopping and checkpointing, evaluates its performance, and visualizes the training history and confusion matrix.

```
#!/usr/bin/env python3
import pandas as pd
import tensorflow as tf
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report

# Pfad zur CSV-Datei, die die 5-Sekunden-Ausgaben aus der App enthlt (
    kommagetrennt)
CSV_PATH = r"C:\Users\lpera\label_dataset\merged_label_dataset.csv"

def load_and_split_data(csv_path):
    """
    Liest die CSV-Datei, entfernt die 'timestamp' Spalte und
    fhrt eine stratified Split (80/20) in Trainings- und Testdaten durch.
    """
    df = pd.read_csv(csv_path, delimiter=",", encoding="utf-8")
    df.columns = df.columns.str.strip()
    if "timestamp" in df.columns:
```

```
        df = df.drop(columns=["timestamp"]) # Weil wir natrlich keine
            Zeitstempel brauchen
    df["status_gt"] = df["status_gt"].astype(str).str.lower() # Alles
        kleingeschrieben, weil wir das so wollen
    train_df, test_df = train_test_split(
        df, test_size=0.2, random_state=42, stratify=df["status_gt"]
    )
    return train_df, test_df

def get_feature_columns(df):
    feature_cols = [col for col in df.columns if col != "status_gt"]
    numeric_cols = []
    categorical_cols = []
    for col in feature_cols:
        if df[col].dtype == object:
            categorical_cols.append(col)
        else:
            numeric_cols.append(col)
    return numeric_cols, categorical_cols

def build_preprocessing_model(train_df, numeric_cols, categorical_cols):
    inputs = {}
    encoded_features = []
    for col in numeric_cols:
        inp = tf.keras.Input(shape=(1,), name=col)
        norm = tf.keras.layers.Normalization(name=f"{col}_norm")
        norm.adapt(train_df[[col]].values)
        encoded = norm(inp)
        inputs[col] = inp
        encoded_features.append(encoded)
    for col in categorical_cols:
        inp = tf.keras.Input(shape=(1,), name=col, dtype=tf.string)
        lookup = tf.keras.layers.StringLookup(output_mode='int', name=f"{col}
            _lookup")
        lookup.adapt(train_df[col].values)
        num_tokens = lookup.vocabulary_size()
        one_hot = tf.keras.layers.CategoryEncoding(num_tokens=num_tokens,
            output_mode='one_hot', name=f"{col}_onehot")
        encoded = one_hot(lookup(inp))
        inputs[col] = inp
        encoded_features.append(encoded)
    concatenated = tf.keras.layers.Concatenate()(encoded_features)
    return inputs, concatenated

def build_model(feature_vector):
    x = tf.keras.layers.Dense(128, activation='relu')(feature_vector)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Dropout(0.3)(x)
```

```
    x = tf.keras.layers.Dense(64, activation='relu')(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Dropout(0.3)(x)
    x = tf.keras.layers.Dense(32, activation='relu')(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Dropout(0.2)(x)
    output = tf.keras.layers.Dense(11, activation='softmax')(x)
    return output

def df_to_dict(df, feature_cols):
    return {col: df[col].values for col in feature_cols}

def main():
    train_df, test_df = load_and_split_data(CSV_PATH)
    numeric_cols, categorical_cols = get_feature_columns(train_df)
    print("Numeric columns:", numeric_cols)
    print("Categorical columns:", categorical_cols)
    inputs, concatenated = build_preprocessing_model(train_df, numeric_cols,
        categorical_cols)
    outputs = build_model(concatenated)
    model = tf.keras.Model(inputs=inputs, outputs=outputs)
    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
    feature_cols = numeric_cols + categorical_cols
    train_features = df_to_dict(train_df, feature_cols)
    test_features = df_to_dict(test_df, feature_cols)
    label_mapping = {
        "vehicle in subway": 0,
        "outdoor in nature": 1,
        "indoor in subway-station": 2,
        "outdoor on foot": 3,
        "vehicle in tram": 4,
        "indoor in supermarket": 5,
        "indoor with window closed": 6,
        "vehicle in subway (old)": 7,
        "vehicle in bus": 8,
        "vehicle in e-bus": 9,
        "indoor with window open": 10
    }
    train_labels = train_df["status_gt"].map(label_mapping)
    test_labels = test_df["status_gt"].map(label_mapping)
    if train_labels.isnull().any() or test_labels.isnull().any():
        raise ValueError("Es gibt Labels in 'status_gt', die nicht im Mapping
            vorhanden sind.") # Ja, das passiert
    train_labels = train_labels.values
    test_labels = test_labels.values
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_accuracy',
```

```
        patience=5, restore_best_weights=True)
    checkpoint = tf.keras.callbacks.ModelCheckpoint("best_model.keras", monitor
        ='val_accuracy', save_best_only=True)
    history = model.fit(train_features, train_labels,
                        epochs=50,
                        validation_split=0.2,
                        batch_size=32,
                        callbacks=[early_stopping, checkpoint])
    loss, accuracy = model.evaluate(test_features, test_labels)
    print("Test Accuracy:", accuracy)
    plt.figure(figsize=(12,5))
    plt.subplot(1,2,1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Accuracy over Epochs')
    plt.legend()
    plt.subplot(1,2,2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Loss over Epochs')
    plt.legend()
    plt.tight_layout()
    plt.show()
    test_pred = model.predict(test_features)
    test_pred_labels = np.argmax(test_pred, axis=1)
    cm = confusion_matrix(test_labels, test_pred_labels)
    cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    print("Normalized Confusion Matrix:")
    print(cm_normalized)
    plt.figure(figsize=(6,6))
    sns.heatmap(cm_normalized, annot=True, fmt='.2f', cmap='Blues',
                xticklabels=list(label_mapping.keys()),
                yticklabels=list(label_mapping.keys()))
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title('Normalized Confusion Matrix')
    plt.show()
    report = classification_report(test_labels, test_pred_labels, target_names=
        list(label_mapping.keys()))
    print("Classification Report:")
    print(report)
    model.save("tf_nn_model.keras")

if __name__ == "__main__":
```

```
    main()
```

Listing 5: Neural Network Training Code for Final Status Determination

Explanation of Key Steps: The code loads and preprocesses a CSV of 5-second app outputs, splits the data into training and testing sets, and builds a neural network with a preprocessing branch for both numeric and categorical features. It trains and evaluates the model, displays training history and confusion matrix, and saves the final model for use in the app.

A.7 Time-Series Analysis of Predicted Status and NO_2 Data

This subsection presents the Python code used to perform a time-series analysis that merges the model's predictions from `campaign.csv` with corresponding NO_2 air quality data (Section A.12). The code aggregates predictions and ground truth on a 2-minute basis, merges them with NO_2 data (also aggregated), and plots a combined graph with a gradient overlay for NO_2 trends.

```
#!/usr/bin/env python3
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.dates as mdates

# ----------------------------------------
# 1) CSVs laden (Vorhersagen + NO2)
# ----------------------------------------
CSV_PATH = r"C:\Users\lpera\Downloads\campaign.csv"
MODEL_PATH = r"C:\Users\lpera\tf_nn_model.keras"
NO2_CSV_PATH = r"C:\Users\lpera\Downloads\NO2_airquix10-data-2025-02-19 04
    _01_12.csv"

label_mapping = {
    "vehicle in subway": 0,
    "outdoor in nature": 1,
    "indoor in subway-station": 2,
    "outdoor on foot": 3,
    "vehicle in tram": 4,
    "indoor in supermarket": 5,
    "indoor with window closed": 6,
    "vehicle in subway (old)": 7,
    "vehicle in bus": 8,
    "vehicle in e-bus": 9,
    "indoor with window open": 10
}
inv_label_mapping = {v: k for k, v in label_mapping.items()}
```

```
short_label_mapping = {
    "vehicle in subway": "subway",
    "outdoor on foot": "street",
    "indoor in supermarket": "supermarket",
    "vehicle in tram": "tram",
    "indoor with window closed": "indoor closed",
    "indoor in subway-station": "subway-station",
    "vehicle in subway (old)": "subway old",
    "vehicle in bus": "bus",
    "vehicle in e-bus": "e-bus",
    "outdoor in nature": "nature",
    "indoor with window open": "indoor open"
}

df = pd.read_csv(CSV_PATH, delimiter=",", encoding="utf-8")
df['timestamp'] = pd.to_datetime(df['timestamp'])
df['status_gt'] = df['status_gt'].astype(str).str.strip().str.lower()
FEATURE_COLS = [col for col in df.columns if col not in ["timestamp", "
    status_gt"]]

model = tf.keras.models.load_model(MODEL_PATH)
print("Model erfolgreich geladen. Hoffentlich funktioniert es.")

predictions = []
num_rows = len(df)
for i in range(num_rows):
    input_data = {}
    for col in FEATURE_COLS:
        value = df.loc[i, col]
        if np.issubdtype(df[col].dtype, np.number):
            input_data[col] = np.array([[value]], dtype=np.float32)
        else:
            input_data[col] = tf.convert_to_tensor([[str(value)]], dtype=tf.
                string)
    pred = model.predict(input_data)
    prob_vector = pred[0]
    pred_class = int(np.argmax(prob_vector))
    confidence = float(prob_vector[pred_class])
    predictions.append((df.loc[i, 'timestamp'], pred_class, confidence))

pred_df = pd.DataFrame(predictions, columns=['timestamp', 'pred_mode_class', '
    confidence'])
pred_df['minute'] = pred_df['timestamp'].dt.floor('2min')
pred_minute = pred_df.groupby('minute').agg(
    pred_mode_class=pd.NamedAgg(column='pred_mode_class', aggfunc=lambda x: x.
        mode().iloc[0]),
    avg_confidence=pd.NamedAgg(column='confidence', aggfunc='mean')
).reset_index()
```

```
df['true_status_num'] = df['status_gt'].map(label_mapping)
df['minute'] = df['timestamp'].dt.floor('2min')
true_minute = df.groupby('minute').agg(
    true_mode_status=pd.NamedAgg(
        column='true_status_num',
        aggfunc=lambda x: x.mode().iloc[0] if not x.mode().empty else np.nan
    )
).reset_index().rename(columns={'timestamp': 'minute'})

merged_df = pd.merge(pred_minute, true_minute, on='minute', how='inner')
print("Merged dataframe head:")
print(merged_df.head())

df_no2 = pd.read_csv(NO2_CSV_PATH, parse_dates=["Time"]).sort_values(by="Time")
manual_start_time = pd.Timestamp("2025-02-18 13:35:00")
merged_df_filtered = merged_df[merged_df['minute'] >= manual_start_time].copy()
df_no2_filtered = df_no2[df_no2['Time'] >= manual_start_time].copy()
times_no2 = df_no2_filtered["Time"].to_numpy()
vals_no2 = df_no2_filtered["no2_aq10"].to_numpy()
df_no2_filtered['minute'] = df_no2_filtered['Time'].dt.floor('2min')
no2_agg_2min = df_no2_filtered.groupby('minute')['no2_aq10'].mean().reset_index
    ()
no2_agg_2min.rename(columns={'no2_aq10': 'no2_mean'}, inplace=True)

# ----------------------------------------
# 2) Manueller Startzeitpunkt und Filtern
# ----------------------------------------
# (Schon oben angewendet, falls du es vergessen hast)

# ----------------------------------------
# 3) Plot: Ein Subplot (Klassen + NO)
# ----------------------------------------
sns.set_style("whitegrid")
fig, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(14, 6))
ax1.plot(
    merged_df_filtered['minute'],
    merged_df_filtered['pred_mode_class'],
    marker='o', linestyle='-', color='tab:blue',
    label='Vorhergesagter Zustand (Modus)',
    zorder=10
)
ax1.scatter(
    merged_df_filtered['minute'],
    merged_df_filtered['true_mode_status'],
    color='tab:orange', marker='s', s=80,
    label='Wahrer Zustand (Modus)',
    zorder=11
```

```
)
ax1.set_ylabel("Zustand")
ax1.set_title("Zeitreihe des Zustands + NO$_2$ Trend")
all_classes = sorted(label_mapping.values())
ax1.set_yticks(all_classes)
ax1.set_yticklabels([
    short_label_mapping.get(inv_label_mapping.get(cls, "Unknown"), "Unknown")
    for cls in all_classes
])
ax_no2 = ax1.twinx()
ax_no2.grid(False)
if len(times_no2) > 0:
    cmap = plt.get_cmap("RdYlGn_r")
    norm = plt.Normalize(0, 45)
    n_stripes = 25
    ax_no2.set_ylim(0, 45)
    for i in range(len(vals_no2) - 1):
        x1_ = times_no2[i]
        x2_ = times_no2[i+1]
        y1_ = vals_no2[i]
        y2_ = vals_no2[i+1]
        color_h = cmap(norm(min(y1_, 45)))
        ax_no2.plot([x1_, x2_], [y1_, y1_], color=color_h, linewidth=2, zorder
            =2)
        color_v = cmap(norm(min(y2_, 45)))
        ax_no2.plot([x2_, x2_], [y1_, y2_], color=color_v, linewidth=2, zorder
            =2)
        bottom_level = 0
        top_level = y1_
        levels = np.linspace(bottom_level, top_level, n_stripes + 1)
        for s in range(n_stripes):
            bottom = levels[s]
            top = levels[s+1]
            alpha_val = 0.15 * ((s+1)/n_stripes)
            ax_no2.fill_between(
                [x1_, x2_], top, bottom,
                facecolor=color_h, alpha=alpha_val,
                edgecolor='none',
                zorder=1
            )
    ax_no2.set_ylabel("NO$_2$ (aq10)")
else:
    print("Keine NO$_2$ Daten verfgbar nach dem manuellen Startzeitpunkt.")
    ax_no2.set_ylabel("NO$_2$ (aq10)")
ax1.xaxis.set_major_locator(mdates.AutoDateLocator())
ax1.xaxis.set_major_formatter(mdates.DateFormatter("%H:%M"))
start_plot = pd.Timestamp("2025-02-18 13:30:00")
end_plot = pd.Timestamp("2025-02-18 14:40:00")
```

```
ax1.set_xlim(start_plot, end_plot)
fig.tight_layout(rect=[0, 0, 0.8, 1])
cbar_ax = fig.add_axes([0.82, 0.15, 0.02, 0.7])
if len(times_no2) > 0:
    sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
    sm.set_array([])
    cbar = fig.colorbar(sm, cax=cbar_ax)
else:
    sm = plt.cm.ScalarMappable()
    sm.set_array([])
    cbar = fig.colorbar(sm, cax=cbar_ax)
fig.autofmt_xdate(rotation=45)
plt.savefig("NO2_plot.png", dpi=300, bbox_inches="tight")
plt.show()

print("merged_df_filtered Bereich:",
      merged_df_filtered['minute'].min(),
      merged_df_filtered['minute'].max())
print("df_no2_filtered Bereich:",
      df_no2_filtered['Time'].min(),
      df_no2_filtered['Time'].max())
```

Listing 6: Time-Series Analysis Code for Merging Predictions with NO Data

Explanation of Key Steps: The code loads a predictions CSV and CO_2 data, performs inference using a pre-trained Keras model, aggregates predictions and ground truth over 2-minute intervals, and merges these with CO_2 measurements. A scatter plot is then generated to compare the predicted and true status classes over time with a CO_2 gradient overlay.

A.8 Rolling Analysis of Prediction Confidence and NO_2 Trend

This subsection presents the code for analyzing how prediction confidence evolves over time alongside NO_2 concentrations. The script loads a predictions CSV, performs inference at 5-second intervals, computes rolling statistics (mean and standard deviation over a 60-second window), and plots these together with NO_2 data using a gradient effect.

```
#!/usr/bin/env python3
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.dates as mdates

# ------------------------------------------------
# 1) Daten Laden: Vorhersagen + NO2
# ------------------------------------------------
CSV_PATH = r"C:\Users\lpera\Downloads\campaign.csv"
```

```
MODEL_PATH = r"C:\Users\lpera\tf_nn_model.keras"
NO2_CSV_PATH = r"C:\Users\lpera\Downloads\NO2_airquix10-data-2025-02-19 04
    _01_12.csv"

df = pd.read_csv(CSV_PATH, delimiter=",", encoding="utf-8")
df['timestamp'] = pd.to_datetime(df['timestamp'])
FEATURE_COLS = [col for col in df.columns if col not in ["timestamp", "
    status_gt"]]

model = tf.keras.models.load_model(MODEL_PATH)
print("Model erfolgreich geladen. Was fr ein Wunder.")

predictions = []
for i in range(len(df)):
    input_data = {}
    for col in FEATURE_COLS:
        value = df.loc[i, col]
        if np.issubdtype(df[col].dtype, np.number):
            input_data[col] = np.array([[value]], dtype=np.float32)
        else:
            input_data[col] = tf.convert_to_tensor([[str(value)]], dtype=tf.
                string)
    pred = model.predict(input_data)
    prob_vector = pred[0]
    pred_class = int(np.argmax(prob_vector))
    confidence = float(prob_vector[pred_class])
    predictions.append((df.loc[i, 'timestamp'], pred_class, confidence))

pred_df = pd.DataFrame(predictions, columns=['timestamp', 'pred_mode_class', '
    confidence'])

df_no2 = pd.read_csv(NO2_CSV_PATH, parse_dates=["Time"]).sort_values(by="Time")

# -------------------------------------------------
# 2) Daten Filtern und Rollende Statistiken
# -------------------------------------------------
manual_start_time = pd.Timestamp("2025-02-18 13:30:00")
pred_df_filtered = pred_df[pred_df['timestamp'] >= manual_start_time].copy()
pred_df_filtered['confidence_smooth'] = pred_df_filtered['confidence'].rolling(
    window=12, center=True).mean()
pred_df_filtered['confidence_std'] = pred_df_filtered['confidence'].rolling(
    window=12, center=True).std()

df_no2_filtered = df_no2[df_no2['Time'] >= manual_start_time].copy()
times_no2 = df_no2_filtered["Time"].to_numpy()
vals_no2 = df_no2_filtered["no2_aq10"].to_numpy()

# -------------------------------------------------
```

```
# 3) Plot: Confidence (Links) und NO (Rechts)
# ------------------------------------------------
sns.set_style("whitegrid")
fig, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(14, 6))
ax1.scatter(
    pred_df_filtered['timestamp'],
    pred_df_filtered['confidence'],
    color='tab:blue', marker='o', s=15, alpha=0.3,
    label='Confidence (5s)'
)
ax1.plot(
    pred_df_filtered['timestamp'],
    pred_df_filtered['confidence_smooth'],
    color='red', linewidth=2, label='Rollender Mittelwert (~60s)'
)
ax1.fill_between(
    pred_df_filtered['timestamp'],
    pred_df_filtered['confidence_smooth'] - pred_df_filtered['confidence_std'],
    pred_df_filtered['confidence_smooth'] + pred_df_filtered['confidence_std'],
    color='red', alpha=0.2, label='1 Std'
)
ax1.set_ylabel("Confidence")
ax1.set_title("Confidence (Rollender Mittelwert + Std) + NO$_2$ Trend")
ax1.legend(loc='upper left')
ax_no2 = ax1.twinx()
ax_no2.grid(False)
if len(times_no2) > 1:
    cmap = plt.get_cmap("RdYlGn_r")
    norm = plt.Normalize(0, 45)
    ax_no2.set_ylim(0, 90)
    n_stripes = 25
    for i in range(len(vals_no2) - 1):
        x1_ = times_no2[i]
        x2_ = times_no2[i+1]
        y1_ = vals_no2[i]
        y2_ = vals_no2[i+1]
        color_h = cmap(norm(min(y1_, 45)))
        color_v = cmap(norm(min(y2_, 45)))
        ax_no2.plot([x1_, x2_], [y1_, y1_], color=color_h, linewidth=2, zorder
            =2)
        ax_no2.plot([x2_, x2_], [y1_, y2_], color=color_v, linewidth=2, zorder
            =2)
        bottom_level = 0
        top_level = y1_
        if top_level < 0:
            top_level = 0
        levels = np.linspace(bottom_level, top_level, n_stripes + 1)
        for s in range(n_stripes):
```

```
            bottom = levels[s]
            top = levels[s+1]
            alpha_val = 0.15 * ((s+1)/n_stripes)
            ax_no2.fill_between(
                [x1_, x2_], top, bottom,
                facecolor=color_h, alpha=alpha_val,
                edgecolor='none',
                zorder=1
            )
    ax_no2.set_ylabel("NO$_2$ (aq10)")
else:
    print("Keine NO$_2$ Daten nach dem manuellen Startzeitpunkt verfgbar. Echt
        schade.")
    ax_no2.set_ylabel("NO$_2$ (aq10)")
ax1.xaxis.set_major_locator(mdates.AutoDateLocator())
ax1.xaxis.set_major_formatter(mdates.DateFormatter("%H:%M"))
start_plot = pd.Timestamp("2025-02-18 13:30:00")
end_plot = pd.Timestamp("2025-02-18 15:15:00")
ax1.set_xlim(start_plot, end_plot)
fig.tight_layout(rect=[0, 0, 0.8, 1])
cbar_ax = fig.add_axes([0.82, 0.15, 0.02, 0.7])
if len(times_no2) > 1:
    sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
    sm.set_array([])
    cbar = fig.colorbar(sm, cax=cbar_ax)
else:
    sm = plt.cm.ScalarMappable()
    sm.set_array([])
    cbar = fig.colorbar(sm, cax=cbar_ax)
fig.autofmt_xdate(rotation=45)
plt.savefig("NO2_confidence_rolling.png", dpi=300, bbox_inches="tight")
plt.show()

print("Datenbereich pred_df_filtered:",
      pred_df_filtered['timestamp'].min(),
      pred_df_filtered['timestamp'].max())
print("Datenbereich df_no2_filtered:",
      df_no2_filtered['Time'].min(),
      df_no2_filtered['Time'].max())
```

Listing 7: Rolling Analysis Code for Prediction Confidence and NO Trend

Explanation of Key Steps: The code loads prediction data and NO_2 measurements, performs inference and aggregates predictions over 2-minute intervals, and computes rolling statistics for prediction confidence. It then produces a dual-axis plot that overlays the smoothed confidence values with a gradient representation of NO_2 levels.

A.9 Time-Series Analysis of Prediction Confidence and CO_2 Trend

This subsection presents the code that analyzes prediction confidence alongside CO_2 data. It processes a predictions CSV, performs inference at 5-second intervals, computes rolling means and standard deviations for the confidence values, and plots these along with CO_2 measurements as a gradient on a secondary axis (Section A.12).

```
#!/usr/bin/env python3
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib.dates as mdates

# ------------------------------------------------
# 1) Daten Laden (Vorhersage CSV, Keras Modell, CO2 Daten)
# ------------------------------------------------
CSV_PATH = r"C:\Users\lpera\Downloads\campaign.csv"
MODEL_PATH = r"C:\Users\lpera\tf_nn_model.keras"
CO2_CSV_PATH = r"C:\Users\lpera\Downloads\CO2_airquix10-data-2025-02-19 04
    _04_40.csv"

df = pd.read_csv(CSV_PATH, delimiter=",", encoding="utf-8")
df['timestamp'] = pd.to_datetime(df['timestamp'])
FEATURE_COLS = [col for col in df.columns if col not in ["timestamp", "
    status_gt"]]

model = tf.keras.models.load_model(MODEL_PATH)
print("Model erfolgreich geladen. Hoffentlich funktioniert es.")

predictions = []
for i in range(len(df)):
    input_data = {}
    for col in FEATURE_COLS:
        value = df.loc[i, col]
        if np.issubdtype(df[col].dtype, np.number):
            input_data[col] = np.array([[value]], dtype=np.float32)
        else:
            input_data[col] = tf.convert_to_tensor([[str(value)]], dtype=tf.
                string)
    pred = model.predict(input_data)
    prob_vector = pred[0]
    pred_class = int(np.argmax(prob_vector))
    confidence = float(prob_vector[pred_class])
    predictions.append((df.loc[i, 'timestamp'], pred_class, confidence))

pred_df = pd.DataFrame(predictions, columns=['timestamp', 'pred_mode_class', '
```

```
    confidence'])
df_co2 = pd.read_csv(CO2_CSV_PATH, parse_dates=["Time"]).sort_values(by="Time")

# ------------------------------------------------
# 2) Daten Filtern und Rollende Statistiken
# ------------------------------------------------
manual_start_time = pd.Timestamp("2025-02-18 13:30:00")
pred_df_filtered = pred_df[pred_df['timestamp'] >= manual_start_time].copy()
pred_df_filtered['confidence_smooth'] = pred_df_filtered['confidence'].rolling(
    window=12, center=True).mean()
pred_df_filtered['confidence_std'] = pred_df_filtered['confidence'].rolling(
    window=12, center=True).std()

df_co2_filtered = df_co2[df_co2['Time'] >= manual_start_time].copy()
times_co2 = df_co2_filtered["Time"].to_numpy()
vals_co2 = df_co2_filtered["CO2_ppm"].to_numpy()

# ------------------------------------------------
# 3) Plot: Confidence (Links) + CO (Rechts)
# ------------------------------------------------
sns.set_style("whitegrid")
fig, ax = plt.subplots(figsize=(14,6))
ax.scatter(
    pred_df_filtered['timestamp'],
    pred_df_filtered['confidence'],
    color='tab:blue', marker='o', s=15, alpha=0.3,
    label='Confidence (5s)'
)
ax.plot(
    pred_df_filtered['timestamp'],
    pred_df_filtered['confidence_smooth'],
    color='red', linewidth=2, label='Rollender Mittelwert (60s)'
)
ax.fill_between(
    pred_df_filtered['timestamp'],
    pred_df_filtered['confidence_smooth'] - pred_df_filtered['confidence_std'],
    pred_df_filtered['confidence_smooth'] + pred_df_filtered['confidence_std'],
    color='red', alpha=0.2, label='1 Std'
)
ax.set_ylabel("Confidence")
ax.set_title("Confidence (Rollender Mittelwert + Std) + CO$_2$ Trend")
ax.legend(loc='upper left')
ax2 = ax.twinx()
ax2.grid(False)
if len(times_co2) > 1:
    cmap = plt.get_cmap("RdYlGn_r")
    norm = plt.Normalize(1600, 3000)
    ax2.set_ylim(1600, 6000)
```

```
    n_stripes = 25
    for i in range(len(vals_co2) - 1):
        x1_ = times_co2[i]
        x2_ = times_co2[i+1]
        y1_ = vals_co2[i]
        y2_ = vals_co2[i+1]
        color_h = cmap(norm(min(y1_, 3000)))
        color_v = cmap(norm(min(y2_, 3000)))
        ax2.plot([x1_, x2_], [y1_, y1_], color=color_h, linewidth=2)
        ax2.plot([x2_, x2_], [y1_, y2_], color=color_v, linewidth=2)
        bottom_level = 1600
        top_level = y1_
        if top_level < 1600:
            top_level = 1600
        levels = np.linspace(bottom_level, top_level, n_stripes + 1)
        for s in range(n_stripes):
            bottom = levels[s]
            top = levels[s+1]
            alpha_val = 0.2 * ((s+1)/n_stripes)
            ax2.fill_between(
                [x1_, x2_], top, bottom,
                facecolor=color_h, alpha=alpha_val,
                edgecolor='none'
            )
    ax2.set_ylabel("CO$_2$ (ppm)")
else:
    print("Keine CO2-Daten nach dem manuellen Startzeitpunkt. Echt schade.")
    ax2.set_ylabel("CO$_2$ (ppm)")
ax.xaxis.set_major_locator(mdates.AutoDateLocator())
ax.xaxis.set_major_formatter(mdates.DateFormatter("%H:%M"))
start_plot = pd.Timestamp("2025-02-18 13:30:00")
end_plot = pd.Timestamp("2025-02-18 15:15:00")
ax.set_xlim(start_plot, end_plot)
fig.tight_layout(rect=[0, 0, 0.8, 1])
cbar_ax = fig.add_axes([0.82, 0.15, 0.02, 0.7])
if len(times_co2) > 1:
    sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
    sm.set_array([])
    cbar = fig.colorbar(sm, cax=cbar_ax)
else:
    sm = plt.cm.ScalarMappable()
    sm.set_array([])
    cbar = fig.colorbar(sm, cax=cbar_ax)
fig.autofmt_xdate(rotation=45)
plt.savefig("CO2_plot_confidence_rolling_std.png", dpi=300, bbox_inches="tight"
    )
plt.show()
```

Listing 8: Time-Series Analysis Code for Prediction Confidence and CO Trend

Explanation of Key Steps: The code loads predictions and CO_2 data, aggregates predictions and ground truth on a 2-minute basis, filters data from a manual start time, and generates a dual-axis plot. The left axis shows the raw and smoothed prediction confidence, while the right axis displays CO_2 trends as a gradient. This visualization aids in evaluating the relationship between prediction confidence and CO_2 measurements.

A.10 Statistical Comparison of CO_2 Levels for Predicted vs. True Classes

This subsection presents the code to statistically compare CO_2 levels between predicted and true classes. The predictions and CO_2 data are aggregated on a 2-minute basis, and for each class, the mean and standard deviation of CO_2 are computed. A scatter plot with error bars is then generated to compare the mean CO_2 levels of the true classes (x-axis) with the predicted classes (y-axis), along with a diagonal reference line.

```
#!/usr/bin/env python3
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns

# ------------------------------------------------
# 1) CSVs Laden (Vorhersagen CSV, Keras Modell, CO2)
# ------------------------------------------------
CSV_PATH = r"C:\Users\lpera\Downloads\campaign.csv"
MODEL_PATH = r"C:\Users\lpera\tf_nn_model.keras"
CO2_CSV_PATH = r"C:\Users\lpera\Downloads\CO2_airquix10-data-2025-02-19 04
    _04_40.csv"

df = pd.read_csv(CSV_PATH, delimiter=",", encoding="utf-8")
df['timestamp'] = pd.to_datetime(df['timestamp'])
df['status_gt'] = df['status_gt'].astype(str).str.strip().str.lower()

label_mapping = {
    "vehicle in subway": 0,
    "outdoor in nature": 1,
    "indoor in subway-station": 2,
    "outdoor on foot": 3,
    "vehicle in tram": 4,
    "indoor in supermarket": 5,
    "indoor with window closed": 6,
    "vehicle in subway (old)": 7,
    "vehicle in bus": 8,
    "vehicle in e-bus": 9,
```

```
    "indoor with window open": 10
}
inv_label_mapping = {v: k for k, v in label_mapping.items()}

short_label_mapping = {
    "vehicle in subway": "subway",
    "outdoor on foot": "street",
    "indoor in supermarket": "supermarket",
    "vehicle in tram": "tram",
    "indoor with window closed": "indoor closed",
    "indoor in subway-station": "subway-station",
    "vehicle in subway (old)": "subway old",
    "vehicle in bus": "bus",
    "vehicle in e-bus": "e-bus",
    "outdoor in nature": "nature",
    "indoor with window open": "indoor open"
}

FEATURE_COLS = [col for col in df.columns if col not in ["timestamp", "
    status_gt"]]

model = tf.keras.models.load_model(MODEL_PATH)
print("Model erfolgreich geladen. Echt cool, oder?")

predictions = []
for i in range(len(df)):
    input_data = {}
    for col in FEATURE_COLS:
        value = df.loc[i, col]
        if np.issubdtype(df[col].dtype, np.number):
            input_data[col] = np.array([[value]], dtype=np.float32)
        else:
            input_data[col] = tf.convert_to_tensor([[str(value)]], dtype=tf.
                string)
    pred = model.predict(input_data)
    prob_vector = pred[0]
    pred_class = int(np.argmax(prob_vector))
    confidence = float(prob_vector[pred_class])
    predictions.append((df.loc[i, 'timestamp'], pred_class, confidence))

pred_df = pd.DataFrame(predictions, columns=['timestamp', 'pred_mode_class', '
    confidence'])
pred_df['minute'] = pred_df['timestamp'].dt.floor('2min')
pred_minute = pred_df.groupby('minute').agg(
    pred_mode_class=pd.NamedAgg(column='pred_mode_class', aggfunc=lambda x: x.
        mode().iloc[0]),
    avg_confidence=pd.NamedAgg(column='confidence', aggfunc='mean')
).reset_index()
```

```
df['true_status_num'] = df['status_gt'].map(label_mapping)
df['minute'] = df['timestamp'].dt.floor('2min')
true_minute = df.groupby('minute').agg(
    true_mode_status=pd.NamedAgg(column='true_status_num', aggfunc=lambda x: x.
        mode().iloc[0])
).reset_index()

df_co2 = pd.read_csv(CO2_CSV_PATH, parse_dates=["Time"]).sort_values(by="Time")
manual_start_time = pd.Timestamp("2025-02-18 13:30:00")
pred_minute_filtered = pred_minute[pred_minute['minute'] >= manual_start_time].
    copy()
df_co2_filtered = df_co2[df_co2['Time'] >= manual_start_time].copy()
df_co2_filtered['minute'] = df_co2_filtered['Time'].dt.floor('2min')

merged_pred = pd.merge(
    pred_minute_filtered[['minute','pred_mode_class']],
    df_co2_filtered[['minute','CO2_ppm']],
    on='minute',
    how='inner'
)

stats_pred = merged_pred.groupby('pred_mode_class')['CO2_ppm'].agg(['mean','std
    ']).reset_index()
stats_pred.rename(columns={'mean':'mean_pred','std':'std_pred','pred_mode_class
    ':'class_id'}, inplace=True)

df['true_status_num'] = df['status_gt'].map(label_mapping)
df['minute'] = df['timestamp'].dt.floor('2min')
true_minute = df.groupby('minute')['true_status_num'].agg(lambda x: x.mode().
    iloc[0]).reset_index()
true_minute_filtered = true_minute[true_minute['minute'] >= manual_start_time].
    copy()

merged_true = pd.merge(
    true_minute_filtered[['minute','true_status_num']],
    df_co2_filtered[['minute','CO2_ppm']],
    on='minute',
    how='inner'
)

stats_true = merged_true.groupby('true_status_num')['CO2_ppm'].agg(['mean','std
    ']).reset_index()
stats_true.rename(columns={'mean':'mean_true','std':'std_true','true_status_num
    ':'class_id'}, inplace=True)

stats_both = pd.merge(stats_pred, stats_true, on='class_id', how='outer').
    fillna(0)
```

```
stats_both.sort_values(by='class_id', inplace=True)

print("\nCO Stats Both (Pred vs. True):\n", stats_both)

markers = ["o", "s", "^", "v", "D", "<", ">", "p", "X", "h", "*"]

def get_short_label(class_id):
    c_int = int(class_id)
    original_name = inv_label_mapping.get(c_int, "Unknown")
    return short_label_mapping.get(original_name, original_name)

sns.set_style("whitegrid")
fig, ax = plt.subplots(figsize=(8, 6))

for i, row in stats_both.iterrows():
    c = row['class_id']
    c_int = int(c)
    x = row['mean_true']
    y = row['mean_pred']
    xerr = row['std_true']
    yerr = row['std_pred']
    marker_c = markers[c_int] if c_int < len(markers) else "o"
    label_c = get_short_label(c_int)
    ax.errorbar(
        x, y,
        xerr=xerr, yerr=yerr,
        fmt=marker_c, capsize=5,
        ecolor='gray', markersize=8,
        alpha=0.8,
        label=label_c
    )

all_x = stats_both['mean_true']
all_y = stats_both['mean_pred']
min_val = min(all_x.min(), all_y.min())
max_val = max(all_x.max(), all_y.max())
ax.plot([min_val, max_val], [min_val, max_val], 'k--', alpha=0.5)

ax.set_xlabel("Mean CO$_2$ (True Class) [ppm]")
ax.set_ylabel("Mean CO$_2$ (Predicted Class) [ppm]")
ax.set_title("Mean CO$_2$ by True vs. Predicted Class (1 Std)")
ax.legend(loc='best')

plt.tight_layout()
plt.savefig("mean_NO2_pred_vs_true.png", dpi=300, bbox_inches="tight")
plt.show()
```

Listing 9: Statistical Analysis and Scatter Plot of CO Levels by Class (Predicted vs.

True)

Explanation of Key Steps: The code loads predictions and CO_2 data, aggregates them over 2-minute intervals, and computes statistics (mean and standard deviation) for each class for both predicted and true labels. A scatter plot with error bars is generated to compare the mean CO_2 levels, with a diagonal line indicating perfect agreement.

A.11 Comparison of Mean NO_2 Levels for True vs. Predicted Classes

This subsection presents the code to compare NO_2 measurements between predicted and true classes. Predictions and ground truth are aggregated over 2-minute intervals, merged with NO_2 data (averaged over the same interval), and statistical measures are computed per class. A scatter plot with error bars is generated to compare mean NO_2 values, with a diagonal reference line.

```
#!/usr/bin/env python3
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns

# ------------------------------------------------
# 1) Daten Laden (Vorhersagen CSV, Keras Modell, NO2 Daten)
# ------------------------------------------------
CSV_PATH = r"C:\Users\lpera\Downloads\campaign.csv"
MODEL_PATH = r"C:\Users\lpera\tf_nn_model.keras"
NO2_CSV_PATH = r"C:\Users\lpera\Downloads\NO2_airquix10-data-2025-02-19 04
    _01_12.csv"

df = pd.read_csv(CSV_PATH, delimiter=",", encoding="utf-8")
df['timestamp'] = pd.to_datetime(df['timestamp'])
df['status_gt'] = df['status_gt'].astype(str).str.strip().str.lower()

label_mapping = {
    "vehicle in subway": 0,
    "outdoor in nature": 1,
    "indoor in subway-station": 2,
    "outdoor on foot": 3,
    "vehicle in tram": 4,
    "indoor in supermarket": 5,
    "indoor with window closed": 6,
    "vehicle in subway (old)": 7,
    "vehicle in bus": 8,
    "vehicle in e-bus": 9,
    "indoor with window open": 10
}
```

```
inv_label_mapping = {v: k for k, v in label_mapping.items()}

short_label_mapping = {
    "vehicle in subway": "subway",
    "outdoor on foot": "street",
    "indoor in supermarket": "supermarket",
    "vehicle in tram": "tram",
    "indoor with window closed": "indoor closed",
    "indoor in subway-station": "subway-station",
    "vehicle in subway (old)": "subway old",
    "vehicle in bus": "bus",
    "vehicle in e-bus": "e-bus",
    "outdoor in nature": "nature",
    "indoor with window open": "indoor open"
}

FEATURE_COLS = [col for col in df.columns if col not in ["timestamp", "
    status_gt"]]

model = tf.keras.models.load_model(MODEL_PATH)
print("Model erfolgreich geladen. Wenigstens etwas funktioniert.")

predictions = []
for i in range(len(df)):
    input_data = {}
    for col in FEATURE_COLS:
        value = df.loc[i, col]
        if np.issubdtype(df[col].dtype, np.number):
            input_data[col] = np.array([[value]], dtype=np.float32)
        else:
            input_data[col] = tf.convert_to_tensor([[str(value)]], dtype=tf.
                string)
    pred = model.predict(input_data)
    prob_vector = pred[0]
    pred_class = int(np.argmax(prob_vector))
    confidence = float(prob_vector[pred_class])
    predictions.append((df.loc[i, 'timestamp'], pred_class, confidence))

pred_df = pd.DataFrame(predictions, columns=['timestamp', 'pred_mode_class', '
    confidence'])
pred_df['minute'] = pred_df['timestamp'].dt.floor('2min')
pred_minute = pred_df.groupby('minute').agg(
    pred_mode_class=pd.NamedAgg(column='pred_mode_class', aggfunc=lambda x: x.
        mode().iloc[0]),
    avg_confidence=pd.NamedAgg(column='confidence', aggfunc='mean')
).reset_index()

df['true_status_num'] = df['status_gt'].map(label_mapping)
```

```
df['minute'] = df['timestamp'].dt.floor('2min')
true_minute = df.groupby('minute').agg(
    true_mode_status=pd.NamedAgg(column='true_status_num', aggfunc=lambda x: x.
        mode().iloc[0])
).reset_index()

df_no2 = pd.read_csv(NO2_CSV_PATH, parse_dates=["Time"]).sort_values(by="Time")
manual_start_time = pd.Timestamp("2025-02-18 13:30:00")
pred_minute_filtered = pred_minute[pred_minute['minute'] >= manual_start_time].
    copy()
df_no2_filtered = df_no2[df_no2['Time'] >= manual_start_time].copy()
df_no2_filtered['minute'] = df_no2_filtered['Time'].dt.floor('2min')

merged_pred = pd.merge(
    pred_minute_filtered[['minute','pred_mode_class']],
    df_no2_filtered[['minute','no2_aq10']],
    on='minute',
    how='inner'
)

stats_pred = merged_pred.groupby('pred_mode_class')['no2_aq10'].agg(['mean','
    std']).reset_index()
stats_pred.rename(columns={'mean':'mean_pred','std':'std_pred','pred_mode_class
    ':'class_id'}, inplace=True)

df['true_status_num'] = df['status_gt'].map(label_mapping)
df['minute'] = df['timestamp'].dt.floor('2min')
true_minute = df.groupby('minute')['true_status_num'].agg(lambda x: x.mode().
    iloc[0]).reset_index()
true_minute_filtered = true_minute[true_minute['minute'] >= manual_start_time].
    copy()

merged_true = pd.merge(
    true_minute_filtered[['minute','true_status_num']],
    df_no2_filtered[['minute','no2_aq10']],
    on='minute',
    how='inner'
)

stats_true = merged_true.groupby('true_mode_status')['no2_aq10'].agg(['mean','
    std']).reset_index()
stats_true.rename(columns={'mean':'mean_gt','std':'std_gt','true_mode_status':'
    class_id'}, inplace=True)

stats_both = pd.merge(stats_pred, stats_true, on='class_id', how='outer').
    fillna(0)
stats_both.sort_values(by='class_id', inplace=True)
```

```
print("\nCO Stats Both (Pred vs. True):\n", stats_both)

markers = ["o", "s", "^", "v", "D", "<", ">", "p", "X", "h", "*"]

def get_short_label(class_id):
    c_int = int(class_id)
    original_name = inv_label_mapping.get(c_int, "Unknown")
    return short_label_mapping.get(original_name, original_name)

sns.set_style("whitegrid")
fig, ax = plt.subplots(figsize=(8, 6))

for i, row in stats_both.iterrows():
    c = row['class_id']
    c_int = int(c)
    x = row['mean_gt']
    y = row['mean_pred']
    xerr = row['std_gt']
    yerr = row['std_pred']
    marker_c = markers[c_int] if c_int < len(markers) else "o"
    label_c = get_short_label(c_int)
    ax.errorbar(
        x, y,
        xerr=xerr, yerr=yerr,
        fmt=marker_c, capsize=5,
        ecolor='gray', markersize=8,
        alpha=0.8,
        label=label_c
    )

all_x = stats_both['mean_gt']
all_y = stats_both['mean_pred']
min_val = min(all_x.min(), all_y.min())
max_val = max(all_x.max(), all_y.max())
ax.plot([min_val, max_val], [min_val, max_val], 'k--', alpha=0.5)

ax.set_xlabel("Mean NO$_2$ (True Class) [aq10]")
ax.set_ylabel("Mean NO$_2$ (Predicted Class) [aq10]")
ax.set_title("Mean NO$_2$ by True vs. Predicted Class (1 Std)")
ax.legend(loc='best')

plt.tight_layout()
plt.savefig("mean_NO2_pred_vs_true.png", dpi=300, bbox_inches="tight")
plt.show()
```

Listing 10: Statistical Comparison of Mean NO Levels by Class (True vs. Predicted)

Explanation of Key Steps: The code aggregates predictions and ground truth labels from a 5-second interval CSV on a 2-minute basis, merges these with NO_2 measurements

(also aggregated), computes the mean and standard deviation of NO_2 for each class, and visualizes the comparison using a scatter plot with error bars and a diagonal reference line.

A.12 Audio Classification Model Training and Export Using TFLite Model Maker

This subsection presents the Python code for training an audio classification model using TFLite Model Maker. The code is nearly identical to that in the Colab notebook at `https://colab.research.google.com/github/googlecodelabs/odml-pathways/blob/main/audio_classification/colab/model_maker_audio_colab.ipynb`. It demonstrates how to load an audio dataset, create a YAMNet specification, split the data, train the model, evaluate it (including displaying a confusion matrix), and export the model as a TFLite file (with a label file) as well as a SavedModel.

```
import os
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
import tflite_model_maker as mm
from tflite_model_maker import audio_classifier

# 1) Setze den Pfad zum Datensatz (Ja, ich wei, Windows mag keine normalen
    Pfade)
data_dir = r"C:\Users\lpera\audio_dataset"

# 2) Erstelle die YAMNet-Spezifikation fr die Audioklassifikation
spec = audio_classifier.YamNetSpec(
    keep_yamnet_and_custom_heads=False,
    frame_step=3 * audio_classifier.YamNetSpec.EXPECTED_WAVEFORM_LENGTH,
    frame_length=6 * audio_classifier.YamNetSpec.EXPECTED_WAVEFORM_LENGTH
)

# 3) Erstelle den Trainingsdaten-Lader aus dem Ordner
train_data = audio_classifier.DataLoader.from_folder(
    spec,
    os.path.join(data_dir, 'train'),
    cache=True
)

# 4) Teile die Trainingsdaten in Trainings- und Validierungssets (80/20 Split,
    weil es cool klingt)
train_data, validation_data = train_data.split(0.8)

# 5) Erstelle den Testdaten-Lader (natrlich auch notwendig)
test_data = audio_classifier.DataLoader.from_folder(
    spec,
```

```
    os.path.join(data_dir, 'test'),
    cache=True
)

# 6) Trainiere das Modell (endlich, nach all den Vorbereitungen)
batch_size = 32
epochs = 50

model = audio_classifier.create(
    train_data,
    spec,
    validation_data=validation_data,
    batch_size=batch_size,
    epochs=epochs
)

# 7) Bewerte das Modell mit den Testdaten (Weil wir wissen wollen, wie gut es
    wirklich ist)
print("Evaluation on test data:")
metrics = model.evaluate(test_data)
print(metrics)

# Optional: Verwirrungsmatrix anzeigen (Hoffentlich sind die Vorhersagen nicht
    komplett daneben)
conf_matrix = model.confusion_matrix(test_data).numpy()
labels = test_data.index_to_label

plt.figure(figsize=(8,8))
sns.heatmap(
    conf_matrix / conf_matrix.sum(axis=1, keepdims=True),
    annot=True, xticklabels=labels, yticklabels=labels,
    cmap="Blues", fmt=".2f"
)
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()

# 8) Exportiere das Modell als TFLite (weil das der nchste Schritt ist, oder?)
export_dir = os.path.join(data_dir, "exported_model")

model.export(
    export_dir,
    tflite_filename="vehicle_sounds.tflite",
    label_filename="vehicle_labels.txt",
    export_format=[mm.ExportFormat.TFLITE, mm.ExportFormat.LABEL]
)
```

```
# Optional: Exportiere es auch als SavedModel (falls du es brauchst)
model.export(
    export_dir,
    export_format=[mm.ExportFormat.SAVED_MODEL]
)

print("Model exported to folder:", export_dir)
```

Listing 11: Audio Classification Model Training and Export

Explanation of Key Steps: The code sets up the dataset and YAMNet specification, loads and splits the audio data, trains an audio classification model, evaluates it using a confusion matrix, and exports the model in TFLite and SavedModel formats.

Electronic Appendix

Data, code, and figures are provided in electronic form at `https://github.com/LouAirquix/AirquixAppThesis`. This repository includes comprehensive documentation of the workflow, source code, datasets, tutorials and additional materials used in this work. Users are encouraged to consult the repository for detailed instructions, reproducible experiments, and further resources that facilitate testing and replication of the complete system.

References and Acknowledgments

I would like to express my gratitude to several individuals and tools that supported me throughout this thesis. First, I would like to thank Sheng Ye for the calibration of the Airquix device and for making it available for this research. I also extend my thanks to my professor, Professor Dr. Mark Wenig, for his continuous support, insightful feedback, and guidance throughout the course of this project.
Additionally, I acknowledge the use of several helpful tools: DeepL (DeepL, 2025) was used for translating parts of the text from German to English, Gemini Google (2025) in Android Studio assisted in the app development, and OpenAI's ChatGPT (OpenAI, 2025) was sometimes used for debugging and programming assistance. These tools significantly contributed to the efficiency and accuracy of my work. A thanks goes to the template that I used for writing this paper, and to the developers who made it available (Munich, 2025).

References

Activity Recognition API (n.d.). `https://developers.google.com/android/reference/com/google/android/gms/location/ActivityRecognitionApi`. Accessed: 2025-02-21.

Android Studio (n.d.). `https://developer.android.com/studio`. Accessed: 2025-02-21.

Audio Classification with TensorFlow Model Maker (n.d.). `https://colab.research.google.com/github/googlecodelabs/odml-pathways/blob/main/audio_classification/colab/model_maker_audio_colab.ipynb#scrollTo=8QRRAM39aOxS`. Accessed: 2025-02-21.

DeepL (2025). Deepl translator.
URL: *https://www.deepl.com*

Gemmeke, J. F., Ellis, D. P., Freedman, D., Jansen, A., Lawrence, M., Moore, R. C., Plakal, M., Platt, D., Ritter, M., Salamon, J. et al. (2017). Audio set: An ontology and human-labeled dataset for audio events, *ICASSP*.

Google (2025). Android studio.
URL: *https://developer.android.com/studio*

Grafana, M. I. d. L. (n.d.). Airquix10 device description, `https://airq.meteo.physik.uni-muenchen.de/d/f34eca55-1582-499c-9bcc-1ccc8212a2d1/airquix-description`.

Hershey, S., Chaudhuri, S., Ellis, D. P., Gemmeke, J. F., Jansen, A., Moore, R. C., Plakal, M., Platt, D., Saurous, R. A., Seybold, B. et al. (2017). Cnn architectures for large-scale audio classification, *ICASSP*.

Jetpack Compose (n.d.). `https://developer.android.com/jetpack/compose`. Accessed: 2025-02-21.

MIT CSAIL (n.d.). Places2, `http://places2.csail.mit.edu/`. Accessed: 2025-02-20.

Munich, L. (2025). Latex template for lmu munich, `https://de.overleaf.com/latex/templates/tagged/lmu`. Accessed: 2025-02-21.

OpenAI (2025). Chatgpt. Accessed: 2025-02-21.
URL: *https://chat.openai.com*

Pan, S. J. and Yang, Q. (2010). A survey on transfer learning, *IEEE Transactions on Knowledge and Data Engineering* .

Places365: A 10 million Image Database for Scene Recognition (n.d.). `https://github.com/CSAILVision/places365`. Accessed: 2025-02-21.

Russakovsky, O., Deng, J., Su, H. and et al. (2015). Imagenet large scale visual recognition challenge, *International Journal of Computer Vision* **115**(3): 211–252.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks, *CVPR*.

TensorFlow (2015). Tensorflow: An end-to-end open source machine learning platform. Accessed: 2025-02-20.
URL: *https://www.tensorflow.org/*

TensorFlow Lite Model Maker for Audio Classification (n.d.). `https://www.tensorflow.org/lite/guide/model_maker`. Accessed: 2025-02-21.

Transfer Learning with TensorFlow (n.d.). `https://www.tensorflow.org/tutorials/images/transfer_learning`. Accessed: 2025-02-21.

U-Bahn München - Series A (n.d.). `https://www.u-bahn-muenchen.de/fahrzeuge/a/`. Accessed: 2025-02-21.

U-Bahn München - Series C (n.d.). `https://www.u-bahn-muenchen.de/fahrzeuge/c/`. Accessed: 2025-02-21.

Wenig, M. and Ye, S. (2020). Investigation of personal air pollutant exposure by a mobile measurement system - airquix, *Forschungsprojekt AIRQUIX* .

Zhou, B., Lapedriza, A., Khosla, A., Oliva, A. and Torralba, A. (2017). Places: A 10 million image database for scene recognition, *IEEE Transactions on Pattern Analysis and Machine Intelligence* .

Declaration of Authorship I hereby declare that the report submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the report as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here. This paper was not previously presented to another examination board and has not been published.

Munich, February 21, 2025

Luis Peralt Bonell