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., PM_1, $PM_{2.5}$, PM_{10}, 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 for 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 PM_{10}, PM_1, $PM_{2.5}$ 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 and thereby supporting better environmental monitoring.

Objectives:

- Develop an app that automatically classifies environmental states (e.g., indoor, outdoor, vehicle) using multimedia data.
- Analyze the relationship between sensor measurements (e.g., PM_1, $PM_{2.5}$, PM_{10}, and NO_2) and the inferred environmental states. For each state, compute the mean and standard deviation of pollutant concentration.
- Evaluate the classification accuracy of the app's predictions and assess how accurate and reliable the app is for investigating pollutant exposure in different situations.

Structure of the Thesis:
This thesis is organized as follows. Chapter 2 review the theoretical background. Chapter 3 describes the system architecture, implementation details, user interface, and provides an explanation of the Kotlin classes used on the app. Chapter 4 presents the experimental evaluation, including data collection, visualization and model evalaution. 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 developement. Android Studio streamlines the entire process— from designing the user interface to writing, debugging and testing code. Key components include:

- **Project Structure:** The project is organized into folders. The `app/` folder contain the source code (located in `java/` or `kotlin/` subfolder), resources (in the `res/` folder), and configuration files such as `AndroidManifest.xml`.
- **Build Configuration:** All build settings—such as dependencies, SDK versions, and build types—are defined in the `build.gradle` files at both the project and module levels.
- **User Interface Development:** The app's user interface is developed with **Jetpack Compose** (*Jetpack Compose*, n.d.), enabling the creation of dynamic layouts.

In our app, Android Studio serves as the central hub, integrating UI design, background service implementations, 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, ws-dl, 2021). 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 eficiently 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.). Additionaly, TensorFlow Lite Model Maker for Audio Classification was used (*TensorFlow Lite Model Maker for Audio Classification*, n.d.), as described in the official tutorial.

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., Zhou et al., 2017). Its designed for scene recognition and is effective at distinguishing between indoor and outdoor environment. For example, processing the image in Figure 1 produced the following result (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 generate a *Class Activation Map (CAM)* highlighing key regions; see Figure 1 (Zhou et al., 2017).

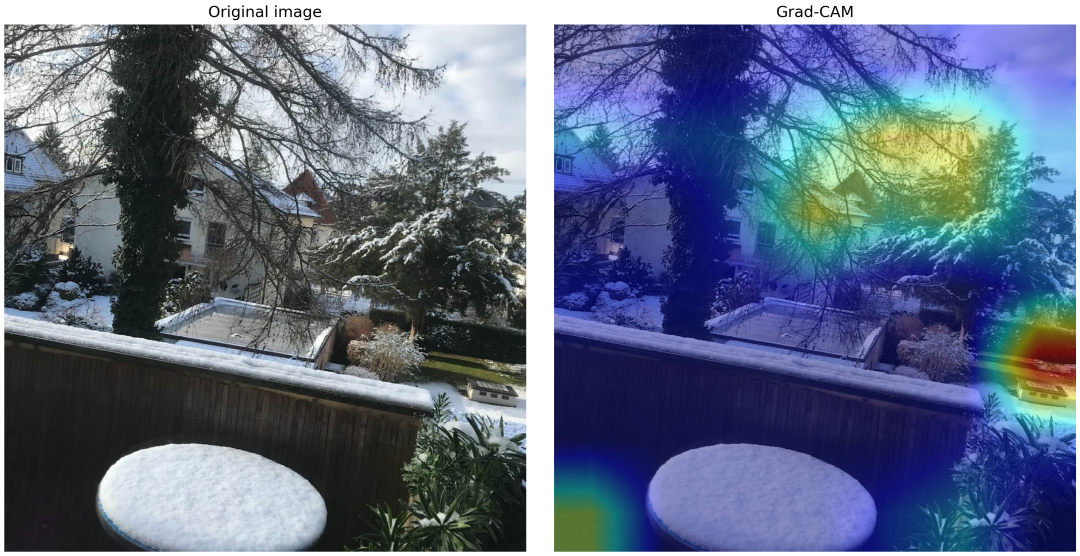

Figure 1: Example of a Class Activation Map (CAM) generated by the Places365-CNN model. A.1

2.4 YAMNet for Audio Classification

YAMNet is a deep neural network for audio event clasification, build on the MobileNet architecture and traind on the AudioSet dataset (Hershey et al., 2017). Its 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 lightweigt, low-cost, portable air quality monitoring device developed by the Meteorological Institute of LMU Munich (Wenig and Ye, 2020). It's designed for mobile measurements and personal air pollutant exposure assessment. Key features include:

- **Sensor Package:** Measure 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 hour. It features a 2.4-inch display, a 10-position mode selector and Wi-Fi connectivity.

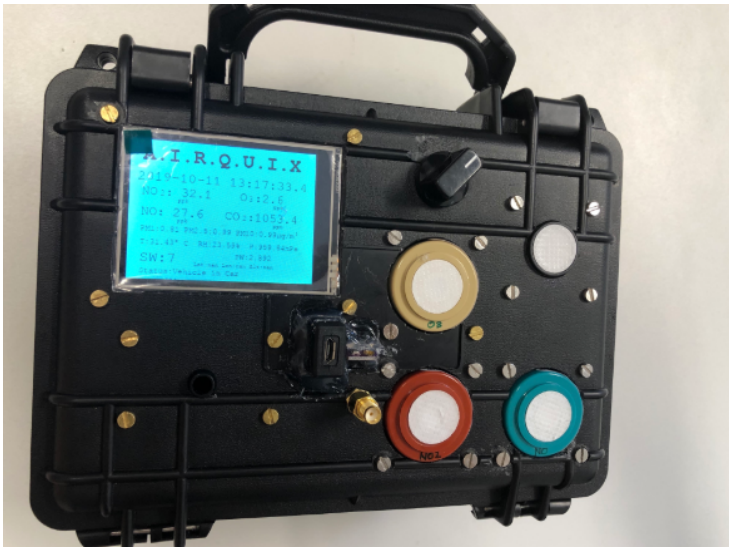

Figure 2: AIRQUIX10 Mobile Air Quality Monitoring Device (Grafana, n.d.).

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 a overall state 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.

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 a introduction to theses concepts, see the TensorFlow Transfer Learning Tutorials (*Transfer Learning with TensorFlow*, n.d.).

3 Methods

3.1 Overview of the Architecture

The App integrates data from multiple sources—including the camera, microphone, GPS, and other onboard sensors. After basic preprocessing steps (e.g., image resizing, audio normalization), these data streams are passed to specialized classification modules. For instance, the image module (using Places365 or a custom vehicle classifier) infers scene categories or identifies vehicle types, while the audio module (YAMNet) classifies ambient sounds. The Activity Recognition API continuously detects user activities (e.g., 'on foot' or 'in vehicle'). The App, developed using Android Studio, logs data in real time, allowing users to monitor sensor values, set a manual ground truth and export the logs as a CSV file.

As illustrated in Figure 3, the outputs from the app's classification modules are processed in a separate Python environment to derive a single environmental states (e.g., *indoor*, *outdoor*, or *bus*). This final state is then linked to pollutant measurements (e.g., PM, NO_2) from the AIRQUIX device for further statistical analysis.

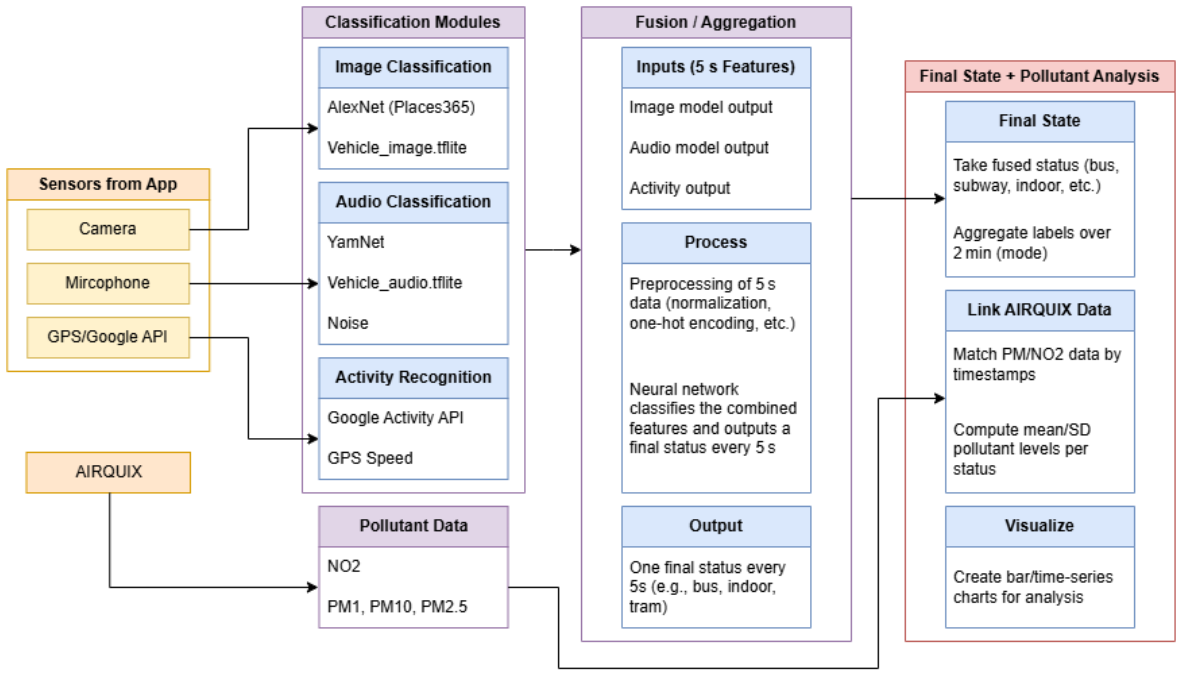

Figure 3: Schematic overview of the system architecture. Sensors from the app (camera, microphone, GPS) feed into classification modules (image, audio, activity). Their 5s outputs are preprocessed and combined by a neural network, producing a final status label (e.g., *bus*, *indoor*). These labels can then be correlated with PM or NO_2 data for further analysis.

3.2 Graphical User Interface (GUI)

The app's user interface includes various elements (see Figure 4).

- **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.

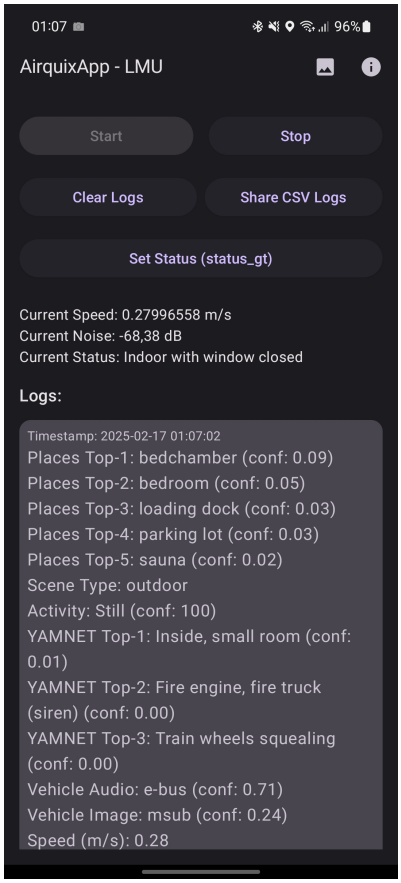

Figure 4: The user interface of the Android application, displaying real-time predictions for environmental states (e.g, indoor, outdoor, bus) along with sensor data such as speed and noise level. At the top, navigation buttons allow the user to start/stop data logging, clear logs or share CSV files. Below, a list displays each loged entry with detailed classification outputs (camera-based Places365, audio-based YamNet, vehicle detection, and manual status). This layout enables users to track, verify and export all relevant data in real time.

3.3 Overview of Kotlin Classes and App Functionality

The app is implemented in Kotlin and consist of several key classe (for full source code, please refer to the Electronic Appendix in Section A.10):

- **MainActivity:** This is the app's main screen. It sets up the user interface using Jetpack Compose, requests permissions, shows dialogs, and connects the UI to the app's shared data.
- **AirquixApplication:** This class initialises the app and create a single shared instance of `MainViewModel` which holds all the apps data and state for use by other components.
- **LoggingService:** A background service that gathers data from the camera, microphone, and GPS, processes this data with machine learning models (Places365, YAMNet, and custom vehicle models) and updates the shared ViewModel.
- **ActivityRecognitionReceiver:** A component that receives activity updates from Google's Activity Recognition API and updates the ViewModel with the detected activity. (To integrate the API into the App, this Github repo was used as a reference (sriharsha1507, n.d.))
- **MainViewModel:** This central hub stores live sensor readings, results from the classification models and a continuously updated log (exported as CSV file).

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 (`MVG-Baureihe C` (*U-Bahn München - Series C*, n.d.), abbreviated as `msub`) from old Munich subway sounds (`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:** Spliting recordings into training and 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 adjusmtments.
4. **Model Evaluation:** Evaluating performance via a normalised confusion matrix (Figure 5), where `msub` and `osub` denote modern and old Munich subway sounds, respectively.

For full code listings and details, see the Appendix A.9.

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 anonymising the images (no recognizable faces). The images were organized into an 80/20 training and validation split and resized to 160×160 pixels.

2. **Data Augmentation:** A pipeline with random horizontale 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 removd from the final inference model which was then converted to Tensorflow Lite.

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

3.7 Conversion of the AlexNet-Places365 Model

To integrate the pre-trained 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 initialize 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 state is determind by agregating the apps 5-second outputs logged in a CSV-file. Each row record outputs from multiple classifiers and sensor readings. For training the status determination, approximatly 5,500 recorded examples were used, see Appendix A.10. 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
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 Ablation Study on Feature Importance

In addition to analyzing time-series trends of predicted status, an ablation study was conducted to quantify each feature contribution to classification accuracy. Specifically, it was investigated how model performances changes when entire feature groups (*image*, *audio*, or *activity*) or individual columns are removed.

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

3.10 Visualization of Prediction State and Air Quality Data

A dedicated data processing pipeline was implemented to finalize the status determination and analyze pollutant trends. After loading the 5-second predictions, we group the 5 second predictions into 2 minute intervals and use the mode (i.e., majority vote) within each interval to obtain a single state label. The core steps include:

- Loading CSV files with prediction and sensor data (NO_2 or PM_1, $PM_{2.5}$, PM_{10}).
- 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 plot 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.

3.11 Evaluation of Correlations Between Confidence and Air Pollutant Data

It was examined whether a correlation exists between classification confidence and measured pollutant concentrations (PM_1, $PM_{2.5}$, PM_{10}, or NO_2). For instance, moving from a quiet outdoor setting to a poorly ventilated indoor area might increase pollutant levels (due to stagnant air) and alter the cues (images, sounds) the model uses—either boosting or reducing classification confidence. Similarly, stable context with fewer emission sources can produce both lower pollutant readings and more consistent recognition cues, leading to higher confidence. To investigate these possible relationships, the application continuously logged both the model's confidence scores and pollutant measurements in CSV file which were then analyzed for patterns of correlation.
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.7.

3.12 Analysis of Pollutant Data

To analyze the relationship between sensor measurments (e.g., PM_1, $PM_{2.5}$, PM_{10}, and NO_2) and the inferred environmental states, the following procedure was applied over 2-minute intervals:

1. The predicted environmental state for each 2-minute interval was determined by computing the mode of the predicted class labels.
2. The pollutant measurements were averaged over the same intervals.
3. Bar plots with error bars were generated for the predicted classes to visualize the mean pollutant levels.
4. The reliability of the predicted classes was further assessed by examining scatter plots and by calculating classification metrics—precision, recall, F1-score, and support—to evaluate prediction accuracy.

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

3.13 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 event was recorded:

- A bus ride was taken from the Hochschule Muenchen bus station 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 at Landshuter Allee.
- From Rotkreuzplatz a 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 later 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 remained unobstructed, so that high-quality audio could be recorded without interference. 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. Additionaly, during vehicle boarding, attention was sometimes given to confirming in the logs whehter 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 (*supermarket*), indoor with window closed (*indoor closed*), indoor with window open (*indoor open*), vehicle in e-bus (*e-bus*), vehicle in bus (*bus*), vehicle in subway (*subway*), vehicle in tram (*tram*), indoor in subway-station (*subway-station*), outdoor on foot (*street*), and outdoor in nature (*nature*).

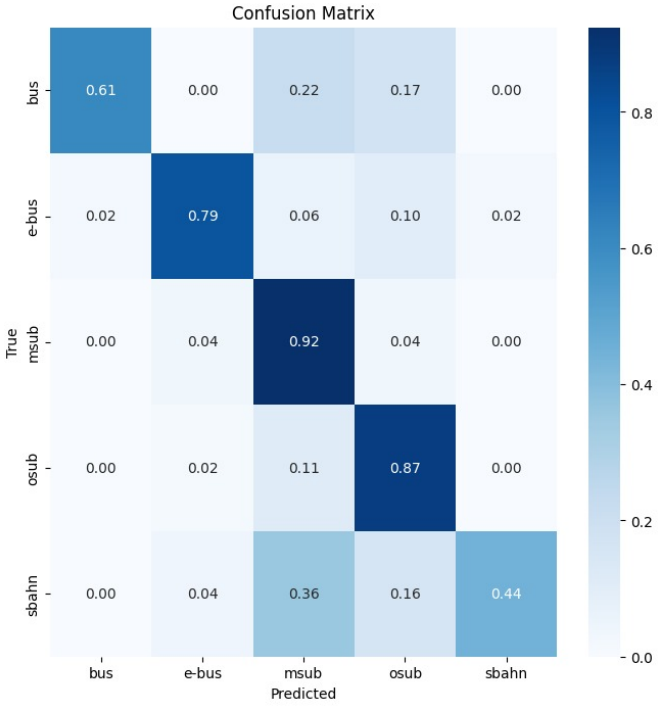

Figure 5: Normalized Confusion Matrix for the fine-tuned vehicle sound model. Here, `msub` represents modern Munich subway sounds (MVG-Baureihe C) and `osub` represents old Munich subway sounds (MVG-Baureihe A).

4 Results

4.1 Evaluation Results of the Vehicle Sound Model

The fine-tuned `vehicle_sounds.tflite` model was valuated using several labels, as summarized by the confusion matrix in Figure 5. Key observations include:

- **S-Bahn:** Only 44% correctly classifid, 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 distinguihing between modern and old Munich subway sounds. The lower performance is likely due to the fact that the audio training data was collectd using the phone's microphone rather than a high-quality microphone and recordings were not consistently verified for quality.

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 6 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 detail 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 6. 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 manualy which might explain the lower performance. This should be further investigated to improve the app's results. For more details, see the Appendix A.10.

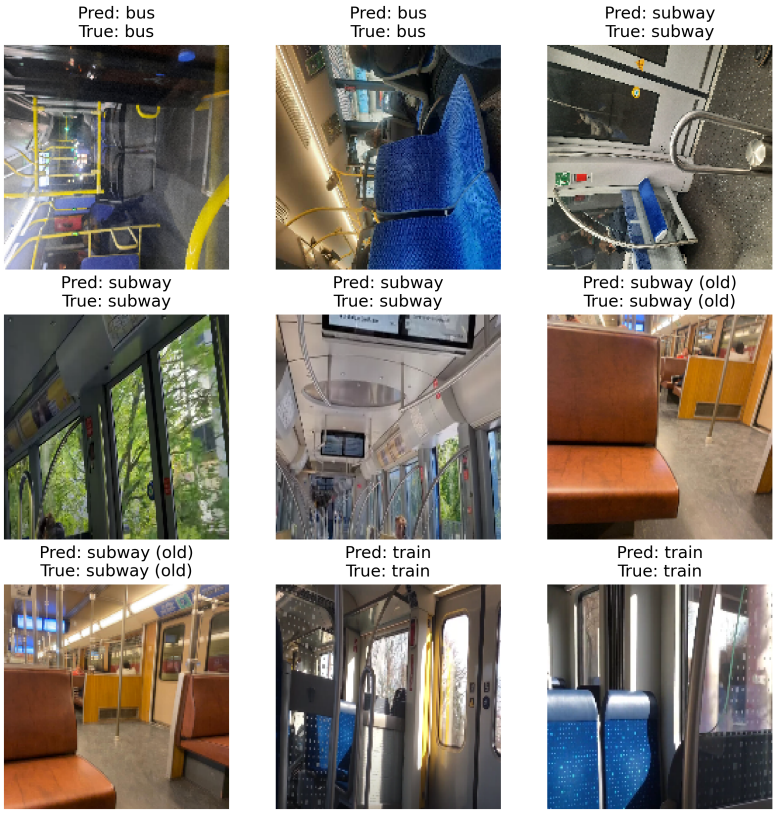

Figure 6: Sample predictions for the vehicle image model on the test dataset. Each image displays the predicted and true labels, with a test accuracy of 100%.

4.3 Results of Predicted Status and Air Quality Data

Figures 7, 8, 9, 11 and 12 show time-series plots of the predicted status (blue circles) and the ground truth (orange squares) together with the measured PM_1, $PM_{2.5}$, PM_{10} or NO_2 concentrations, respectively. The class labels (e.g., *subway*, *bus*, *indoor open*, *indoor*

closed) appear on the left y-axis, while the particulate matter levels are shown on the right y-axis.

PM Observations. Across the three PM metric (PM_1, $PM_{2.5}$, PM_{10}), the highest readings typically occur in vehicular contexts (e.g., bus, e-bus), reflecting traffic-related emission. Notabely, **PM_1 levels in the subway station are relatively high**, whereas $PM_{2.5}$ and PM_{10} remain moderate or show slight increase. In the supermarket, **all PM values drop markedly**, suggetsing improved air quality. Riding the tram is generally associated with a **decline in PM across all metrics**, except for a brief and mild spike in PM_{10}. Overall these observations demonstrate how different indoor settings and modes of transportation can significantly influence particulate matter concentrations.

NO_2 Observations. Figures 11 and 12 present analogous time-series plots for NO_2. As with particulate matter, higher NO_2 readings frequently coincides with vehicular context (e.g., bus, e-bus), reflecting traffic-related emissions. A distinct rise can also being seen in the subway station, culminating in a pronounced peak during the subway ride (see Figure 12). In contrast, the tram doesnt exhibit a significant spike, instead showing a slight decrease. Moreover, while a notable NO_2 increase was observed in the supermarket on Februay 18, no such rise occurred for February 20 where the supermarket NO_2 levels remained relativly low.

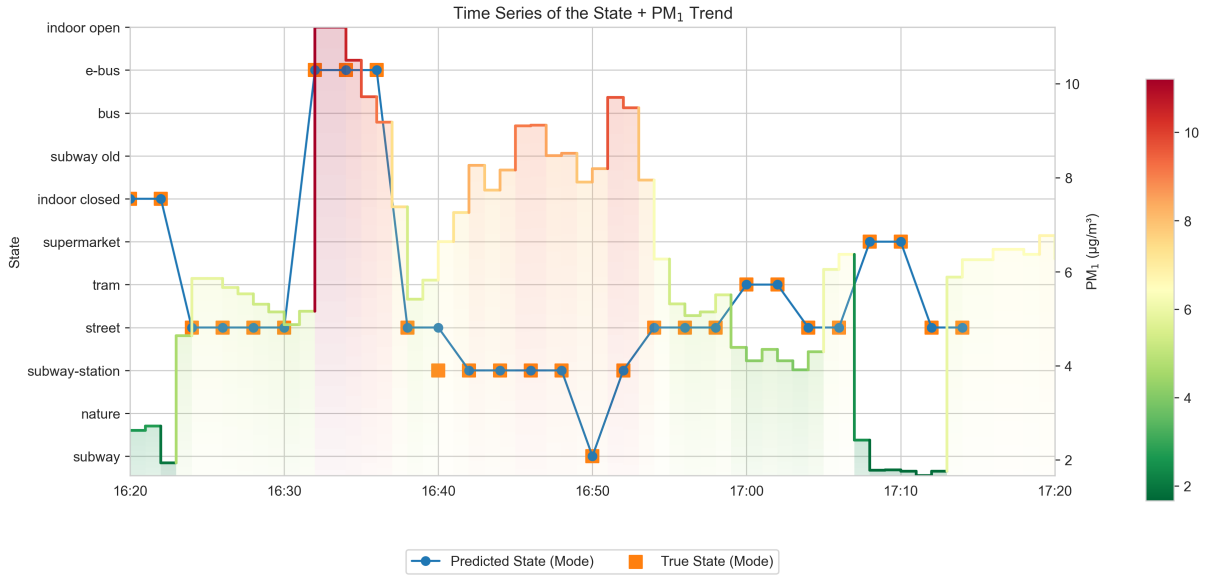

Figure 7: Time series of the predicted status (blue circles) and ground truth (orange squares) alongside PM_1 concentrations for February 20, 2025.

4.4 Results of the Ablation Study

Figure 13 shows how classification accuracy changes when removing entire groups of features (*No Image*, *No Audio*, *No Activity*) or discarding each feature individually. This

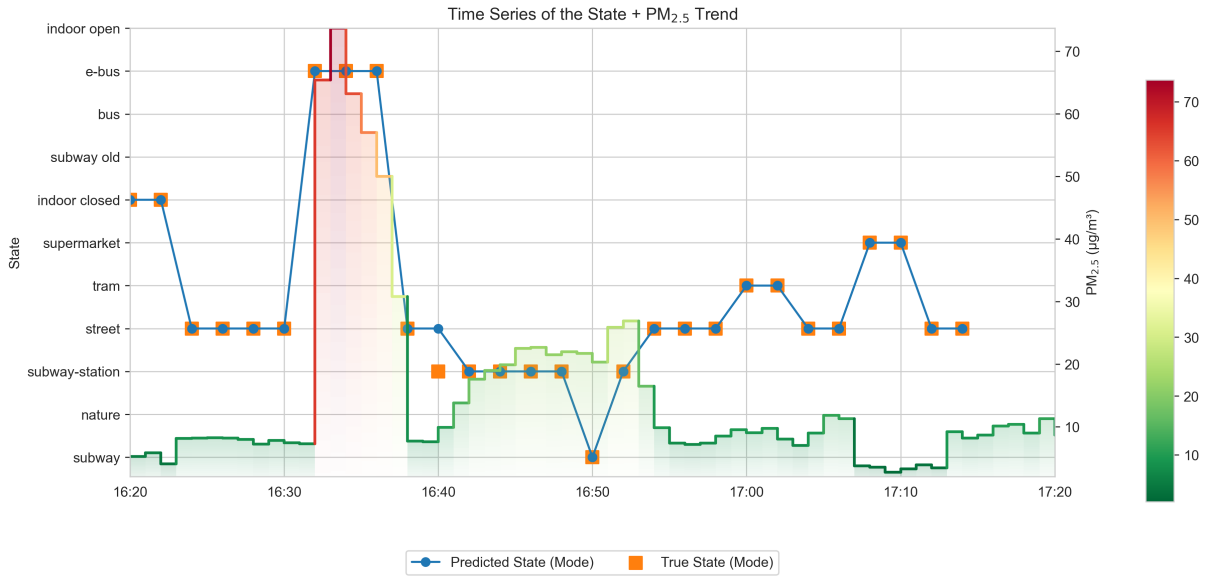

Figure 8: Time series of the predicted status (blue circles) and ground truth (orange squares) alongside $PM_{2.5}$ concentrations for February 20, 2025.

ablation analysis was performed using data collected during the February 20, 2025, campaign. The most pronounced drops in accuracy appear when image or audio inputs are omitted altogether, demonstrating their importance. In the single-feature removal view, certain features—when removed—lead to mimimal impact on performance where as others cause accuracy to decline.

4.5 Results of Correlations Between Confidence and Air Pollutant Data

To investigate whether the model's classification confidence varie with measured pollutant concentrations, the rolling mean and standard deviation of the confidence values were plotted alongside the time-series data for $PM_{2.5}$ and NO_2. Figures 17 and 18 show these results, where:

- The blue circles represent the raw confidence score recorded every 5 seconds.
- The red line indicates the rolling mean (with a window size of roughly one minute), while the shaded region denotes ± 1 standard deviation.
- The pollutant data ($PM_{2.5}$ or NO_2) appear on the right yaxis and use a color gradient to highlight changes in concentration.

Overall, thes plots suggest that classification confidence remains relatively high or stable much of the time, with dips or fluctuation during certain transitions. The polutant measurements, meanwhile, vary according to locations specific or emission sources but they dont display a strong or consistent relationship with the confidence trends. This side by side view does, however provide insight into how environmental factors sometimes may overlap with changes in confidence, even if no direct correlation is evident.

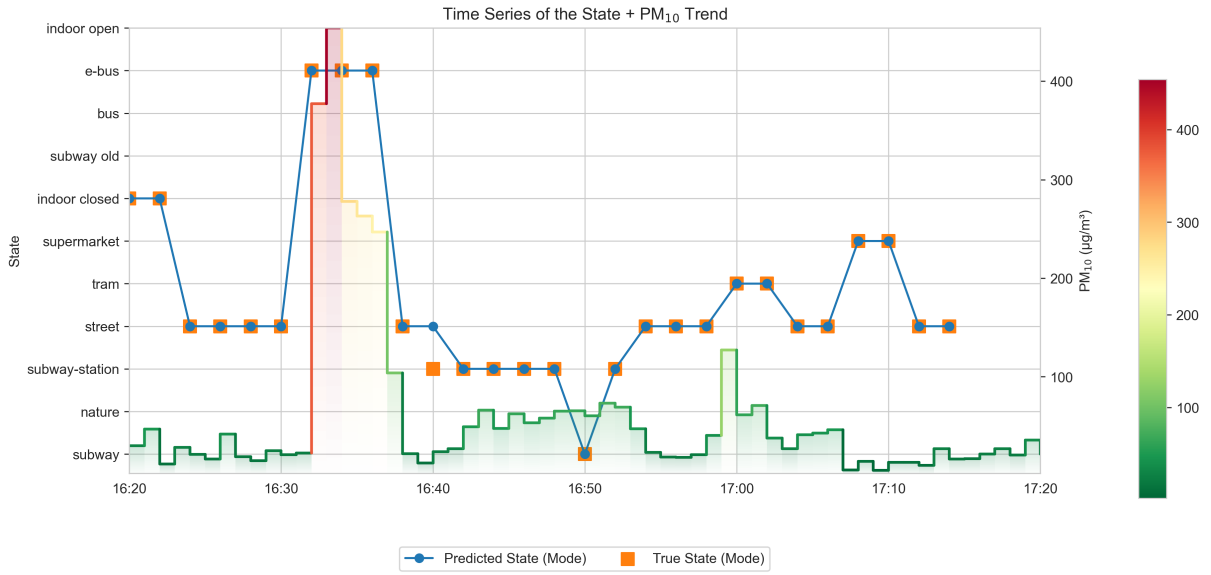

Figure 9: Time series of the predicted status (blue circles) and ground truth (orange squares) alongside PM_{10} concentrations for February 20, 2025.

4.6 Accuracy of Pollutant Levels by True vs. Predicted Classes

Figure 14 shows a scatter plot comparing mean pollutant levels for each class when contrasting *true* versus *predicted* labels. On the x-axis the mean pollutant value for the true class, while he y-axis shows the mean value for the predicted class. The diagonal line ($y = x$) indicates perfect agreement and larger deviations reveal possible mismatches in context recognition or pollutant variability. Error bars represent ± 1 standard deviation. A brief classification report (see Table 1) indicate that app achieves good accuracy, with most classes showing high precision, recall, and F1-scores. In the measurement campaign of 75 predictions, only 8 were incorrect. This strong performance supports using the app's inferred classes to interpret environmental data across different states. As shown in the scatter plot, predicted classes that lie closer to the diagonal align well with ground-truth pollutant levels, highlighting the apps effectiveness in distinguishing various situations and their associated air quality.

4.7 Bar Plot Analysis of Predicted Classes and Pollutant Levels

Figures 15 and 16 present bar plots with error bars (± 1 standard deviation) showing the mean pollutant concentrations (PM_1, $PM_{2.5}$, PM_{10}, and NO_2) for each predicted class. These figures provide a clear comparison of how the app's classification relates to typical pollutant values:

- **Highest Pollutant Levels:** Classes such as *bus* or *e-bus* often exhibit elevated PM and NO_2 measurements, likly reflecting traffic-related emissions.
- **Moderate Ranges:** Predictions like *subway* or *street* show intermediate values that indicate partial exposure to vehicle exhaust or semi enclosed environments.

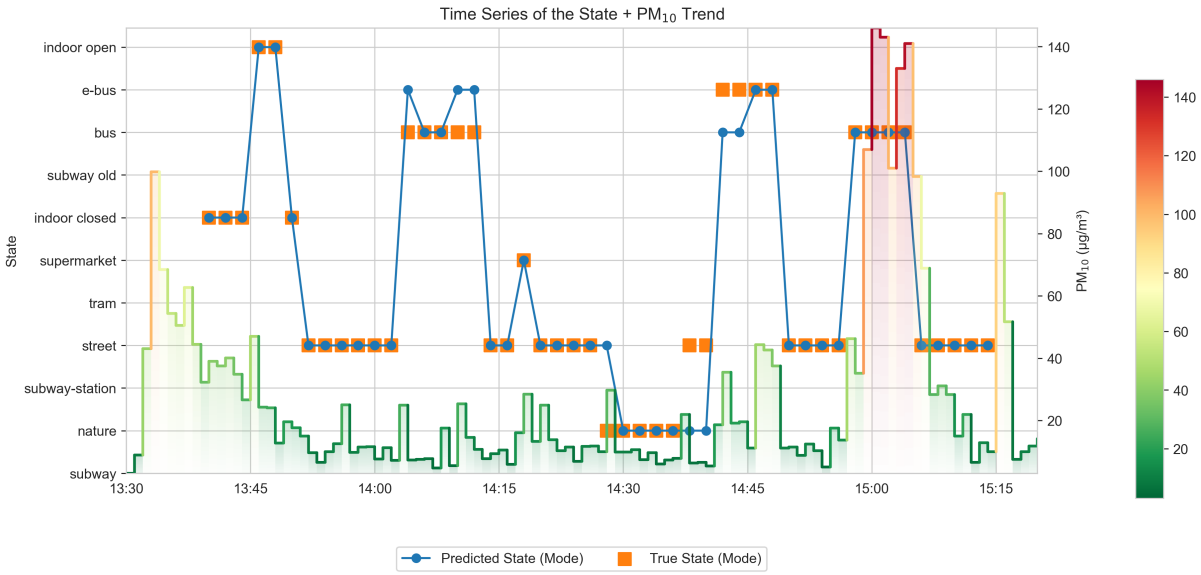

Figure 10: Time series of the predicted status (blue circles) and ground truth (orange squares) alongside PM_{10} concentrations for February 18, 2025.

- **Lower Concentrations:** *Indoor open* or *indoor closed* can vary, but often record lower levels than heavy-traffic scenarios, possibly due to fewer nearby emission sources. *Supermarket* also tends to display relatively reduced particulate matter, pointing better air conditioning or ventilation systems.

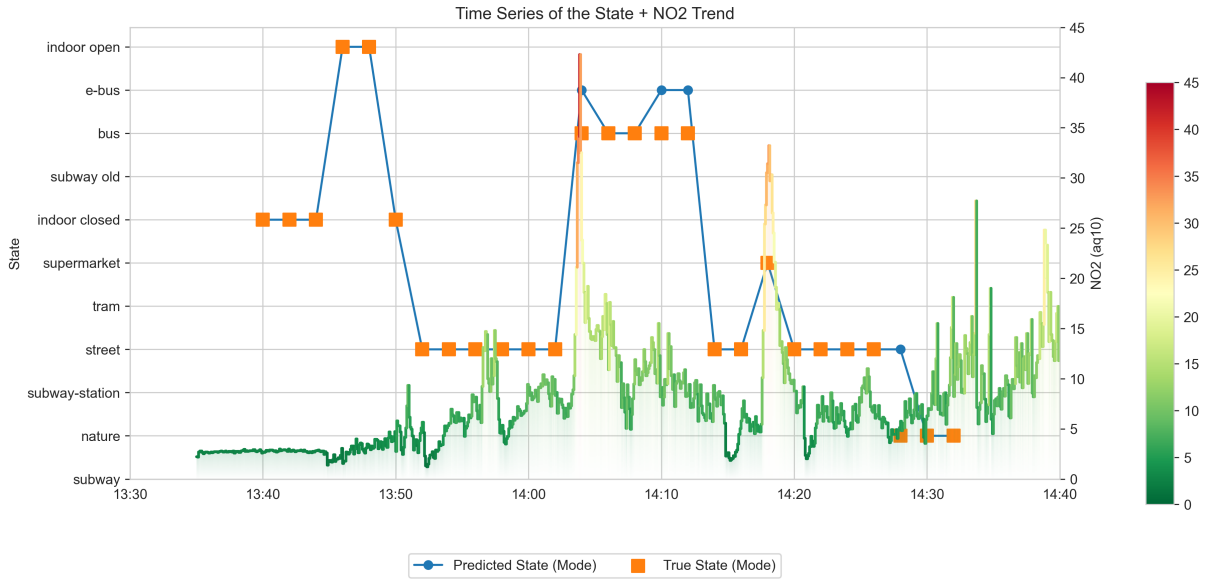

Figure 11: Time series of the predicted status (blue circles) and ground truth (orange squares) alongside NO_2 measurements for February 18, 2025.

5 Discussion

Summary of Key Findings

The results presented in Section 4 show that the predicted environmental status (e.g, indoor, outdoor, vehicle) align well with measured pollutant levels (PM_1, $PM_{2.5}$, PM_{10}, NO_2). Notable observations include:

- **Vehicular Exposure:** Bus and ebus classes consistently show elevated PM and NO_2 concentrations, reflecting traffic-related emissions.
- **Subway Effects:** While PM_1 can be high in subway stations, $PM_{2.5}$ and PM_{10} levels are generally moderate, and NO_2 sometimes spikes during subway travel.
- **Indoor Variability:** Indoor environments with good ventilation (e.g., supermarkets) tend to record lower PM_1, $PM_{2.5}$, and PM_{10} levels, whereas confined spaces (indoor closed) can occasionally show mild increases.
- **Classification Confidence:** The model generally miantains high confidence in stable contexts. Confidence dips occur during rapid environmental changes, yet dont strongly correlate with pollutant fluctuations.
- **Overall Accuracy:** As indicated by the classification report (Table 1), the app achieves high precision, recall, and F1-scores across most classes, supporting the reliability of its inferred statuses for interpreting pollutant data.

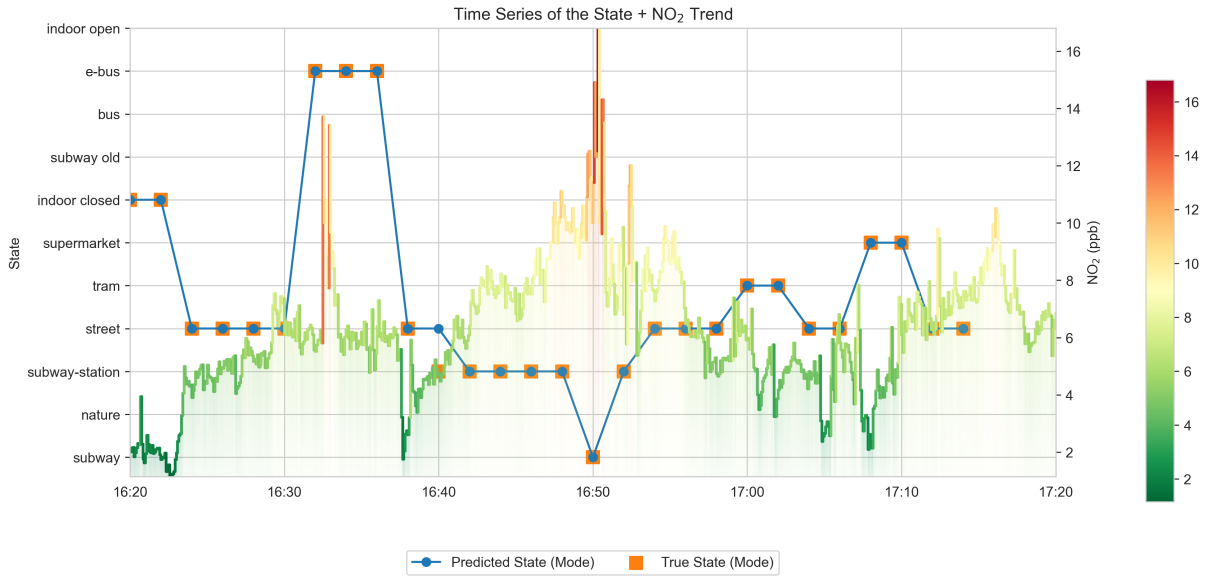

Figure 12: Time series of the predicted status (blue circles) and ground truth (orange squares) alongside NO_2 measurements for February 20, 2025.

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. Integrating state prediction with sensor readings, the system can offer more precise explanations for changes in pollutant levels. This aproach may be particulaly relevant for personal exposure assesments where fine-grained context (e.g., subway vs. bus) significantly impact pollutant exposure. As illustrated in Figure 13, the ablation results clearly show that image- and audio-based features are among the most critical for accurate state prediction. Interestingly, in some cases, removing certain features (e.g., `VEHICLE_audio_2`) can lead to a slight *increase* in accuracy. One possible explanation is that in some situations these features may introduce noise or conflicting signals for the model, so excluding them occasionaly improves overall performance. Conversely, excluding highly relevant inputs—such as scene-recognition outputs—reduce accuracy significantly, demonstrating the crucial role of these features in correctly identifying environmental contexts. Overall, the results demonstrate that the classification remains robust even if some features provide misleading labels, due to the diverse nature of the inputs used.

Limitations.
Althougth the models performed most of the time well in controlled scenarios, a few important limitations should be noted:

- **Geographical Scope:** Measurement were largly restricted to a single urban area (Munich), limiting broader applicability.
- **Sensor Noise and Calibration:** Both smartphone and AIRQUIX10 measurements can exhibit noise, especially in rapidly changing conditions. Additional cali-

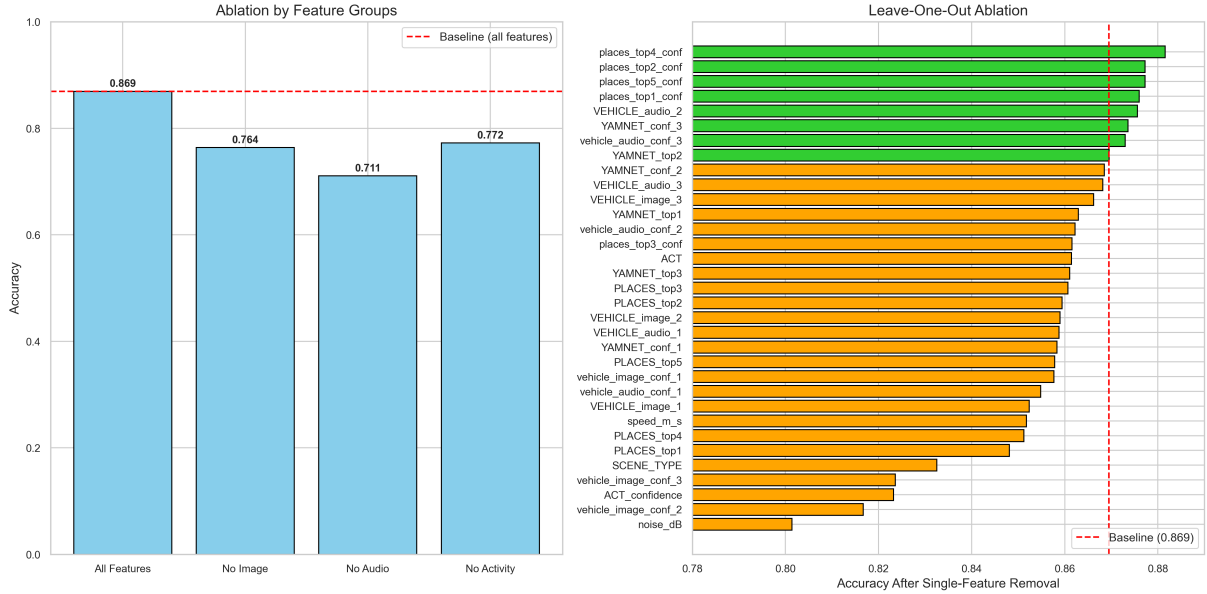

Figure 13: **Feature Ablation Results.** The left bar chart shows model accuracy after removing each of the three main feature groups (No Image, No Audio and No Activity), compared to the baseline using all features. A larger drop from the baseline indicates that the omitted group is essential for classification. The right horizontal bar chart indicates how accuracy changes if exactly one feature is removed.

bration may be required for robust long-term deployment.

- **Dataset Diversity:** The training and evaluation datasets, may not capture all possible environmental or acoustic conditions (e.g., extreme weather, unusual lighting or noise), as data for 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 underscores the potential of combining machine learning with polutant measurements to deliver context-aware insights. Recognizing when users are indoors versus in a vehicle helps explain observed variations in PM or NO_2 levels. Such context-driven monitoring could be integrated into broader air quality initiatives, maybe supporting real-time decision-making for public health and urban planning.

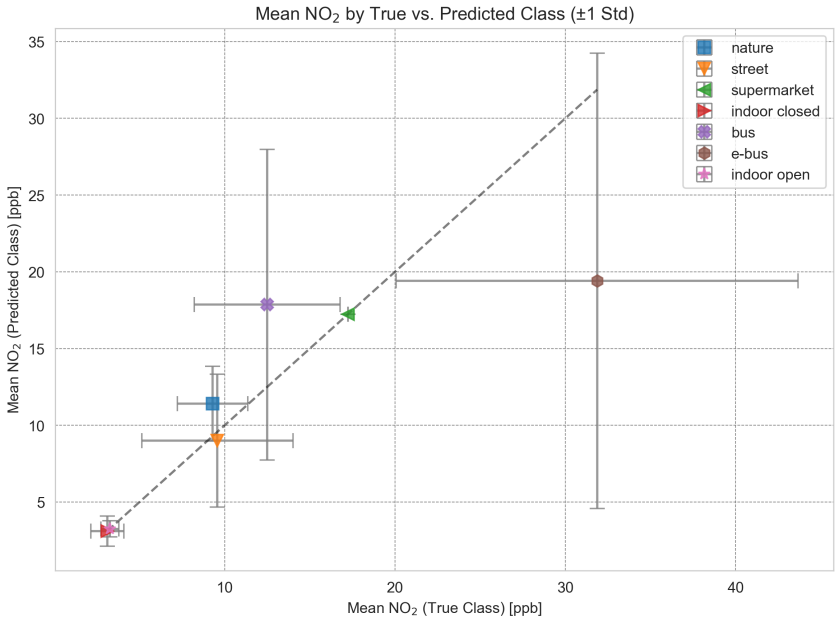

Figure 14: Mean NO_2 by true vs. predicted class (±1 Std) for February 18, 2025. The diagonal line indicates perfect agreement.

6 Conclusion

Overall Contributions.
This work demonstrate the feasibillity of combining multisensor data (camera, audio, activity recognition) with pollutant measurements (PM_1, $PM_{2.5}$, PM_{10}, NO_2) to achieve context-aware environmental classification. The app developed for this purpose log data in real time and integrates various machine learning models, producing a system capable of highlighting how air quality parameters change under different conditions (e.g., indoor, outdoor, transit).

Key Outcomes.
The empirical evaluations confirm that:

- The inferred environmental states (e.g, *bus*, *subway*, *indoor closed*) correspond well to distinctive polutant patterns, especialy regarding elevated PM and NO_2 levels in vehicular contexts.
- Confined indoor spaces or subway stations can exhibit higher PM_1 readings, whereas well ventilated environments (e.g, supermarket) typically register lower particulate matter concentrations.
- Two measurement campaigns demonstrate the app's reliability in determining states, with the classification report (Table 1) showing high precision, recall, and F1-scores. Only small discrepancies arises when comparing predicted states to ground truth labels.

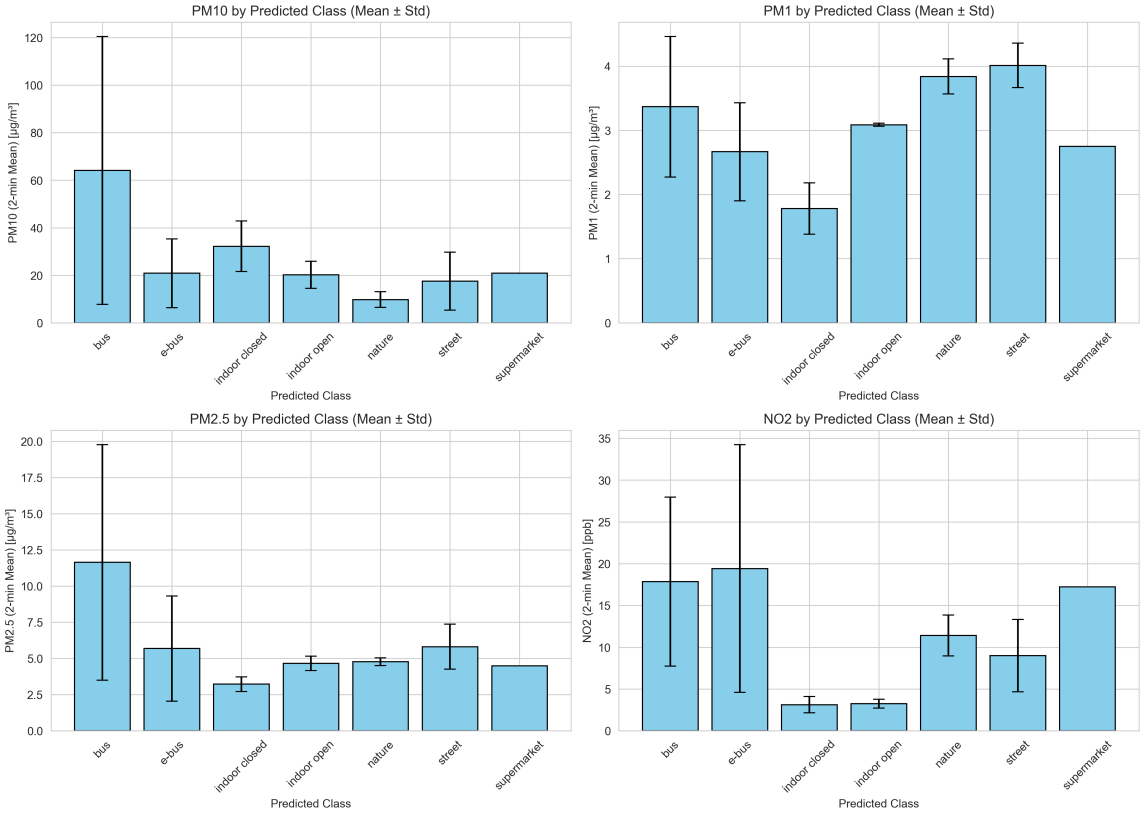

Figure 15: Mean pollutant levels (PM_1, $PM_{2.5}$, PM_{10}, NO_2) by predicted class, measured on February 18 (±1 Std).

Mean Pollutant Concentrations

A key contribution of this work is the automatic determination of average pollutant concentrations for each environmental context identified by the classification. For this purpose, the measured values of relevant pollutants (such as PM_1, $PM_{2.5}$, PM_{10} and NO_2) were aggregated and assigned to the predefined classes (e.g., bus, e-bus, indoor closed, subway). As presented in Section 5, the classification outputs (e.g., *bus*, *e-bus*, *subway*) correlates strongly with distinct pollutant concentration patterns. Specifically, vehicle-related classes show elevated PM and NO_2 values where as well-ventilated indoor contexts often register lower readings. By focusing on these mean concentrations, the system can provide more nuanced exposure assessments and support targeted environmental health interventions. This analysis underlines the importance of context-aware data collection and paves the way for further refinement in real-time monitoring applications.

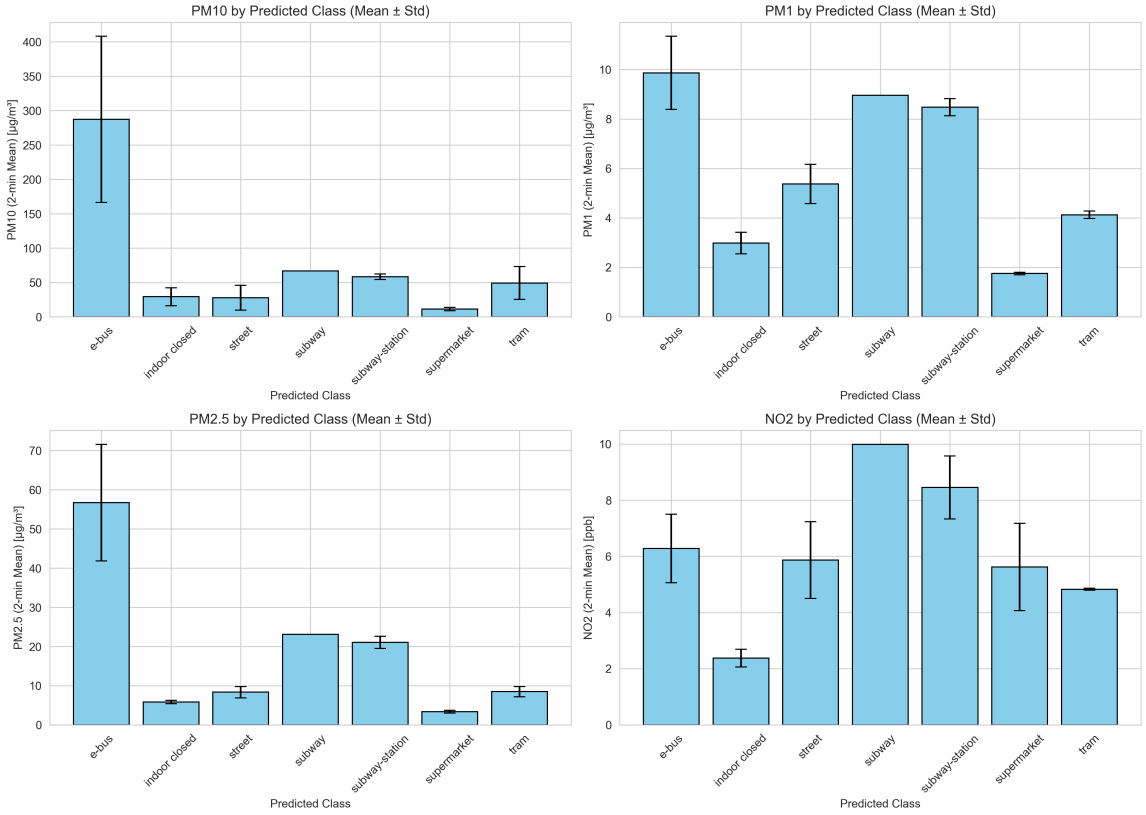

Figure 16: Mean pollutant levels (PM_1, $PM_{2.5}$, PM_{10}, NO_2) by predicted class, measured on February 20 (±1 Std).

7 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 could for example improve 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.

This directions highlight the potential for broader deployment and new ideas, such as integrating hybrid models with AI and expert knowledge to improve status determination.

Class	Precision	Recall	F1-Score	Support
vehicle in subway	0.86	0.83	0.84	132
outdoor in nature	0.94	0.89	0.91	122
indoor in subway-station	0.90	0.75	0.82	102
outdoor on foot	0.87	0.94	0.91	239
vehicle in tram	0.79	0.83	0.81	149
indoor in supermarket	0.96	0.88	0.92	75
indoor with window closed	0.90	0.94	0.92	262
vehicle in subway (old)	0.92	0.88	0.90	40
vehicle in bus	0.90	0.83	0.86	138
vehicle in e-bus	0.82	0.92	0.87	149
indoor with window open	0.89	0.86	0.87	135
accuracy			0.88	1543
macro avg	0.89	0.87	0.87	1543
weighted avg	0.88	0.88	0.88	1543

Table 1: Classification report detailing: *Precision* (fraction of predicted positives that are truly positive), *Recall* (fraction of actual positives that are correctly identified), *F1-score* (harmonic mean of precision and recall), *Support* (number of instances for each class), *macro avg* (the unweighted mean across all classes), and *weighted avg* (the mean across classes weighted by the number of instances in each class) (MarkovML, n.d.). To see how this was calculated, see Appendix A.6

In this setup, an initial AI model might first distinguish between top-level categories such as “inside,” “outside,” or “in-vehicle.” Then based on that result, a second-stage model or rulebased system can classify more precisely (e.g, “indoor open” vs “indoor closed,” or different vehicle types). Domain experts would provide thresholds or emission profiles (e.g., known ranges for bus or subway emissions) to guide these subsequent steps. Furthermore, there is potential to refine the systems confidence-based calibration in various ways:

- **Dynamically adjusting thresholds:** Lower model-confidence could trigger stricter thresholds or cause certain classifications to be disregarded.
- **User interaction:** In cases of very low confidence, the app could prompt the user for additional input or consult alternative measurements.
- **Reweighting sensor data:** The relative importance of specific sensor inputs could be increased or decreased depending on the model’s confidence level in a given situation.

These enhancements would help the app adapt more flexibly to dynamic environment and diverse data sources for further improving both accuracy and reliability.

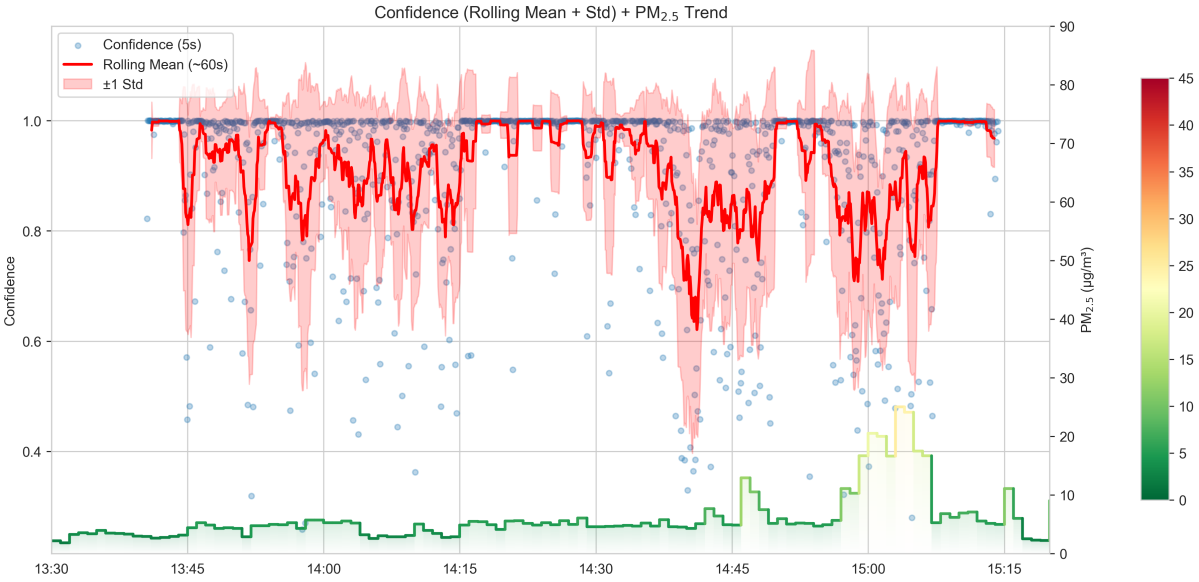

Figure 17: Confidence (5-second intervals, rolling mean ± std) compared with $PM_{2.5}$ concentrations (right y-axis) for February 18, 2025.

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 assesments, enhancing public health decision-making and environmental awareness.

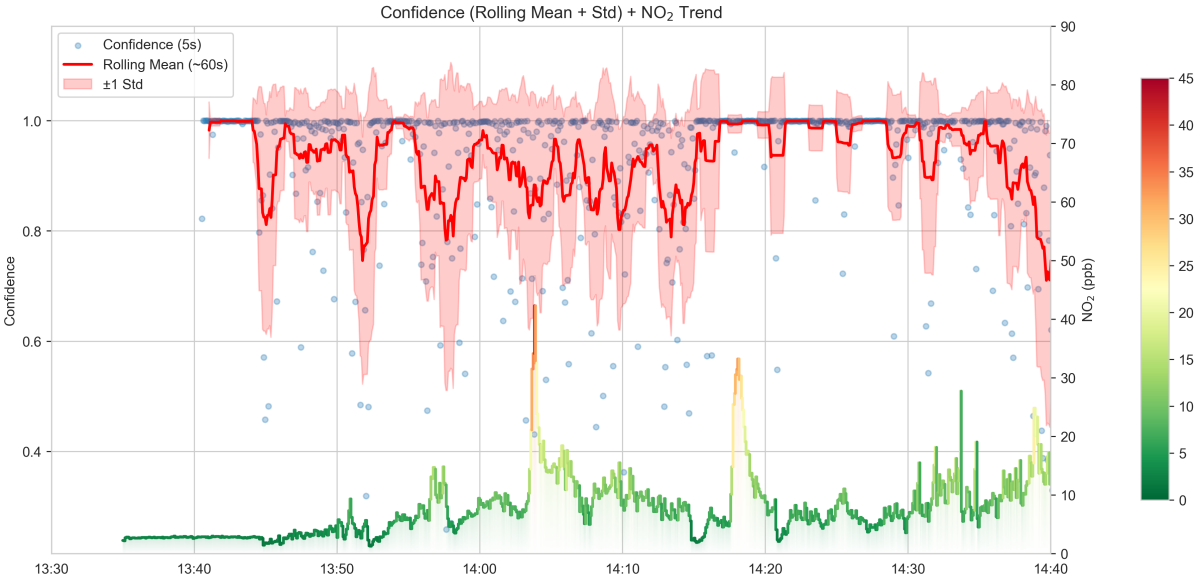

Figure 18: Confidence (5-second intervals, rolling mean ± std) compared with NO_2 concentrations (right y-axis) for February 18, 2025.

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.10). 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.10). 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`). This

code snippet and functionality is based from the Places365 repository to load the model and compute Grad-CAM (CSAILVision, n.d.c,n).

```
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 over-

lays an 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.10).

```
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 (Developers, n.d.a). Finally, the TFLite model is tested on 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.10).

```
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 (CSAILVision, n.d.a).

```
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.10). 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 (Pedregosa et al., n.d., Developers, n.d.b).

```
#!/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 enthaelt (
    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

    # der folgende ansatz zur Aufteilung der Daten ist der Dokumentation von
        scikit-learn [scikitLearn] entnommen.
    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 Script for Confidence and Polutant Data Visualization

This subsection provides a Python script fpr code snippets for plotting classification confidence against various pollutants (PM_1, $PM_{2.5}$, PM_{10}, and NO_2) and generating bar plots by predicted class. Feell free to adapt file paths and parameter as needed (Waskom and the seaborn development team, n.d.).

```
#!/usr/bin/env python3
#!/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) GLOBALE EINSTELLUNGEN: PFADANGABEN, LABEL-MAPPINGS ETC.
##############################################################################

# Haupt-CSV-Datei mit Vorhersagen oder rohen Sensordaten (5-Sekunden-
    Intervallen)
CSV_PATH_CAMPAIGN = r"C:\Users\lpera\Downloads\campaign_02.csv"

# Pfad zum trainierten Keras-Modell
MODEL_PATH = r"C:\Users\lpera\BA_tf_nn_model.keras"

# CSV-Dateien fr verschiedene Schadstoffe (Pollutants) und zugehrige
    Spaltennamen
POLLUTANT_FILES = {
    "PM10": {
```

```
        "path": r"C:\Users\lpera\Downloads\PM10_airquix10-data-2025-02-22 18
            _26_12.csv",
        "col": "pm10"
    },
    "PM1": {
        "path": r"C:\Users\lpera\Downloads\PM1_airquix10-data-2025-02-22 18
            _26_05.csv",
        "col": "pm1"
    },
    "PM2.5": {
        "path": r"C:\Users\lpera\Downloads\PM2.5_airquix10-data-2025-02-22 18
            _25_56.csv",
        "col": "pm2.5"
    },
    "NO2": {
        "path": r"C:\Users\lpera\Downloads\NO2_airquix10-data-2025-02-22 19
            _06_40.csv",
        "col": "no2_aq10"
    }
}

# Ab wann wir Daten betrachten wollen
MANUAL_START_TIME = pd.Timestamp("2025-02-20 16:20:00")

# Zeitfenster fr die Diagramme (Beginn / Ende)
start_plot = pd.Timestamp("2025-02-20 16:20:00")
end_plot = pd.Timestamp("2025-02-20 17:20:00")

# Einheiten der Schadstoffe (fr die Achsenbeschriftung). Hoffentlich korrekt.
units = {
    "PM10": "g/m",
    "PM1": "g/m",
    "PM2.5": "g/m",
    "NO2": "ppb"
}

# Label-Mapping frs Modell
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()}

# Kurze Bezeichnungen fr die Diagramme (die sollen dem Leben Sinn geben)
short_label_mapping = {
    "vehicle in subway": "subway",
    "outdoor in nature": "nature",
    "indoor in subway-station": "subway-station",
    "outdoor on foot": "street",
    "vehicle in tram": "tram",
    "indoor in supermarket": "supermarket",
    "indoor with window closed": "indoor closed",
    "vehicle in subway (old)": "subway old",
    "vehicle in bus": "bus",
    "vehicle in e-bus": "e-bus",
    "indoor with window open": "indoor open"
}

def get_pred_mode_label(class_id):
    """Gibt ne kurze, schnucklige Bezeichnung fr die vorhergesagte Klasse zurck
        ."""
    original_name = inv_label_mapping.get(class_id, "Unknown")
    return short_label_mapping.get(original_name, original_name)

################################################################################
# 2) HAUPT-CSV LADEN UND MODELL-INFERENZ AUSFHREN
################################################################################
df = pd.read_csv(CSV_PATH_CAMPAIGN, delimiter=",", encoding="utf-8")
df['timestamp'] = pd.to_datetime(df['timestamp'])

# Feature-Spalten bestimmen (timestamp/status_gt ausschlieen)
FEATURE_COLS = [col for col in df.columns if col not in ["timestamp", "
    status_gt"]]

# Keras-Modell laden  hoffen wir, dass es das richtige ist
model = tf.keras.models.load_model(MODEL_PATH)
print("Modell erfolgreich geladen. Yuhu. wenigstens funktioniert das..")

# Inferenz alle ~5 Sekunden
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'])

# Vorhersagen in 2-Minuten-Intervallen zusammenfassen (Modus der Klassen)
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()

# Auf manuell gewhlten Startzeitpunkt filtern
pred_minute = pred_minute[pred_minute['minute'] >= MANUAL_START_TIME].copy()

##############################################################################
# 3) HELFER-FUNKTION: CSV LADEN & SCHADSTOFFDATEN IN 2-MIN INTERVALLEN MITTELN
##############################################################################
def load_and_aggregate_2min(csv_path, time_col, value_col):
    """
    Ldt CSV von csv_path, konvertiert time_col zu datetime,
    gruppiert value_col in 2-Minuten-Intervallen (Mittelwert).
    Liefert DataFrame mit Spalten [minute, pollutant_mean].
    """
    df_poll = pd.read_csv(csv_path, parse_dates=[time_col])
    df_poll.sort_values(by=time_col, inplace=True)

    # Nur Daten nach Startzeit
    df_poll = df_poll[df_poll[time_col] >= MANUAL_START_TIME].copy()

    # Zeit auf 2-Minuten runden
    df_poll['minute'] = df_poll[time_col].dt.floor('2min')

    # Mitteln ber jedes 2-Minuten-Intervall
    df_agg = df_poll.groupby('minute')[value_col].mean().reset_index()
    df_agg.rename(columns={value_col: 'pollutant_mean'}, inplace=True)
    return df_agg

##############################################################################
# 4) SCHADSTOFF-DATEN MIT DEN VORHERSAGEN MERGEN & BARPLOTS
##############################################################################
```

```
merged_data = {} # key = "PM10"/"PM1"/"PM2.5"/"NO2", value = DataFrame

for pol_name, info in POLLUTANT_FILES.items():
    csv_path = info["path"]
    col_name = info["col"]

    # Daten laden & 2-Minuten-Intervall-Mittelwerte bilden
    df_pol_2min = load_and_aggregate_2min(csv_path, "Time", col_name)

    # Mit pred_minute zusammenfhren
    merged = pd.merge(
        pred_minute, # [minute, pred_mode_class, avg_confidence]
        df_pol_2min, # [minute, pollutant_mean]
        on='minute',
        how='inner'
    )
    merged_data[pol_name] = merged

# Bar-Diagramme: Mittelwert  Std fr jeden vorhergesagten Status
sns.set_style("whitegrid")
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(14, 10))
axs = axs.flatten()

for i, (pol_name, df_merged) in enumerate(merged_data.items()):
    ax = axs[i]
    # Kurze Labels
    df_merged['pred_mode_label'] = df_merged['pred_mode_class'].apply(
        get_pred_mode_label)

    # Gruppieren nach pred_mode_label: Mittel & Std
    stats = df_merged.groupby('pred_mode_label')['pollutant_mean'].agg(['mean',
        'std']).reset_index()
    stats.rename(columns={'mean':'pollutant_mean','std':'pollutant_std'},
        inplace=True)
    stats.sort_values(by='pred_mode_label', inplace=True)

    # Barplot
    ax.bar(
        x=range(len(stats)),
        height=stats['pollutant_mean'],
        yerr=stats['pollutant_std'],
        tick_label=stats['pred_mode_label'],
        capsize=5,
        color='skyblue',
        edgecolor='black'
    )
    pol_unit = units.get(pol_name, "")
    ax.set_title(f"{pol_name} by Predicted Class (Mean  Std)")
```

```
    ax.set_xlabel("Predicted Class")
    ax.set_ylabel(f"{pol_name} (2-min Mean) [{pol_unit}]")
    ax.tick_params(axis='x', rotation=45)

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

################################################################################
# 5) ZEITREIHEN-PLOTS FR JEDE POLLUTANT (PM10, PM1, PM2.5, NO2),
# ZEIGT NUR DIE VORHERSAGTE KLASSE (KEIN GROUND TRUTH)
################################################################################
def plot_time_series_classes_and_pollutant(df_poll, time_col, val_col, pol_name
    ):
    """
    Zeichnet die vorhergesagte Klasse (2-min mode) auf der linken Achse,
    und den Schadstoff auf der rechten Achse (Verlauf + Farbverlauf).
    Achtung: Kein ground truth hier enthalten, weil wir "true_minute" nicht
        definiert haben.
    """
    df_poll_filtered = df_poll[df_poll[time_col] >= MANUAL_START_TIME].copy()

    sns.set_style("whitegrid")
    fig, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(14, 6))

    # Vorhergesagte Klassen vs. Minute (2-min mode)
    ax1.plot(
        pred_minute['minute'],
        pred_minute['pred_mode_class'],
        marker='o', linestyle='-', color='tab:blue',
        label='Predicted State (Mode)'
    )

    ax1.set_ylabel("State")
    ax1.set_title(f"Time Series of the Predicted State + {pol_name} Trend (2-
        min Aggregation)")

    ax1.legend(loc='upper left')
    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
    ])

    # Rechte Achse fr den Schadstoff
```

```
ax2 = ax1.twinx()
ax2.grid(False)

# Filter auf den end_plot
df_poll_filtered = df_poll_filtered[df_poll_filtered[time_col] <= end_plot]
times_pol = df_poll_filtered[time_col].to_numpy()
vals_pol = df_poll_filtered[val_col].to_numpy()

if len(times_pol) > 0:
    min_val = vals_pol.min()
    max_val = vals_pol.max()
    norm = plt.Normalize(min_val, max_val)
    cmap = plt.get_cmap("RdYlGn_r")
    ax2.set_ylim(min_val, max_val)

    n_stripes = 25
    for i in range(len(vals_pol) - 1):
        x1_ = times_pol[i]
        x2_ = times_pol[i+1]
        y1_ = vals_pol[i]
        y2_ = vals_pol[i+1]

        color_h = cmap(norm(y1_))
        color_v = cmap(norm(y2_))
        ax2.plot([x1_, x2_], [y1_, y1_], color=color_h, linewidth=2)
        ax2.plot([x2_, x2_], [y1_, y2_], color=color_v, linewidth=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 = 0.2 * ((s+1)/n_stripes)
            ax2.fill_between(
                [x1_, x2_], top, bottom,
                facecolor=color_h, alpha=alpha_val,
                edgecolor='none'
            )

    pol_unit = units.get(pol_name, "")
    ax2.set_ylabel(f"{pol_name} [{pol_unit}]")
else:
    print(f"Keine {pol_name}-Daten nach {MANUAL_START_TIME} gefunden.
        Schade.")
    ax2.set_ylabel(f"{pol_name}")

ax1.xaxis.set_major_locator(mdates.AutoDateLocator())
```

```
    ax1.xaxis.set_major_formatter(mdates.DateFormatter("%H:%M"))
    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_pol) > 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(f"{pol_name}_plot.png", dpi=300, bbox_inches="tight")
    plt.show()

################################################################################
# 6) OPTIONAL: CONFIDENCE (ROLLING MEAN  STD) VS. POLLUTANT
################################################################################
def plot_confidence_vs_pollutant(pred_df, df_poll, pol_name, pol_col):
    """
    Zeichnet den gleitenden Mittelwert & Std der Vorhersage-Confidence
    auf der linken Achse, sowie den Schadstoff auf der rechten Achse (
        Farbverlauf).
    Tipp: Wenn die Confidence im Keller ist, vllt war das licht aus ;)
    """
    pdf = pred_df[pred_df['timestamp'] >= MANUAL_START_TIME].copy()
    pdf['confidence_smooth'] = pdf['confidence'].rolling(window=12, center=True
        ).mean()
    pdf['confidence_std'] = pdf['confidence'].rolling(window=12, center=True).
        std()

    dfp = df_poll[df_poll['Time'] >= MANUAL_START_TIME].copy()
    times_pol = dfp['Time'].to_numpy()
    vals_pol = dfp[pol_col].to_numpy()

    sns.set_style("whitegrid")
    fig, ax1 = plt.subplots(figsize=(14, 6))

    # Confidence: Scatter
    ax1.scatter(
        pdf['timestamp'],
        pdf['confidence'],
        color='tab:blue', marker='o', s=15, alpha=0.3,
        label='Confidence (5s)'
    )
```

```
# Rolling mean
ax1.plot(
    pdf['timestamp'],
    pdf['confidence_smooth'],
    color='red', linewidth=2,
    label='Rolling Mean (~60s)'
)
# 1 std
ax1.fill_between(
    pdf['timestamp'],
    pdf['confidence_smooth'] - pdf['confidence_std'],
    pdf['confidence_smooth'] + pdf['confidence_std'],
    color='red', alpha=0.2, label='1 Std'
)

ax1.set_ylabel("Confidence")
ax1.set_title(f"Confidence (Rolling Mean  Std) + {pol_name} Trend")
ax1.legend(loc='upper left')

# Rechte Achse fr den Schadstoff
ax2 = ax1.twinx()
ax2.grid(False)

if len(times_pol) > 1:
    cmap = plt.get_cmap("RdYlGn_r")
    # Beispiel: clamp von 0..45 (da wir keinen schimmer haben obs
        realistisch ist)
    norm = plt.Normalize(0, 45)
    # Achse auf 0..90, total random
    ax2.set_ylim(0, 90)

    n_stripes = 25
    for i in range(len(vals_pol) - 1):
        x1_ = times_pol[i]
        x2_ = times_pol[i+1]
        y1_ = vals_pol[i]
        y2_ = vals_pol[i+1]

        color_h = cmap(norm(min(y1_, 45)))
        color_v = cmap(norm(min(y2_, 45)))
        ax2.plot([x1_, x2_], [y1_, y1_], color=color_h, linewidth=2, zorder
            =2)
        ax2.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)
                ax2.fill_between(
                    [x1_, x2_], top, bottom,
                    facecolor=color_h, alpha=alpha_val,
                    edgecolor='none',
                    zorder=1
                )

        ax2.set_ylabel(f"{pol_name} (units vary)")
    else:
        print(f"Keine {pol_name}-Daten nach {MANUAL_START_TIME}. Sorry!")
        ax2.set_ylabel(f"{pol_name}")

    ax1.xaxis.set_major_locator(mdates.AutoDateLocator())
    ax1.xaxis.set_major_formatter(mdates.DateFormatter("%H:%M"))
    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_pol) > 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(f"{pol_name}_confidence_rolling.png", dpi=300, bbox_inches="
        tight")
    plt.show()

##############################################################################
# 7) BEISPIELVERWENDUNG:
# - Zeitreihenplots fr jeden Schadstoff
# - Confidence vs. Schadstoff
##############################################################################
if __name__ == "__main__":
    # Beispiel: Zeitreihenplot fr PM10
    df_pm10 = pd.read_csv(POLLUTANT_FILES["PM10"]["path"], parse_dates=["Time"
        ]).sort_values(by="Time")
    plot_time_series_classes_and_pollutant(
```

```
    df_poll=df_pm10,
    time_col="Time",
    val_col="pm10",
    pol_name="PM10"
)

# Beispiel: Confidence vs. PM2.5
df_pm25 = pd.read_csv(POLLUTANT_FILES["PM2.5"]["path"], parse_dates=["Time"
    ]).sort_values(by="Time")
plot_confidence_vs_pollutant(
    pred_df=pred_df,
    df_poll=df_pm25,
    pol_name="PM2.5",
    pol_col="pm2.5"
)

# hnlich knnt ihr plot_time_series_classes_and_pollutant
# oder plot_confidence_vs_pollutant fr "PM1" oder "NO2" verwenden.
pass
```

Listing 6: Comprehensive Plotting Code for Confidence and Pollutant Data

Explanation of Key Steps:

- **Global Setup:** Defines file paths, label mappings, and units for pollutants.
- **Loading & Predicting:** Reads the main CSV at 5second intervals and runs predictions using the Keras model, then aggregates them into 2-minute modes.
- **Bar Plots:** merge the predicted classes with pollutant data and plots mean ± std for each predicted class.
- **Time-Series Plots:** Overlays predicted states on one axis with pollutant values (PM, NO_2) on another axis, using color gradients for visual flair.
- **Confidence vs. Pollutant:** Shows how classification confidence changes (or doesn't) alongside pollutant concentrations, which can highlight whether the model's uncertainty correlates with real-world conditions.

A.8 Ablation Study on Feature Removal

This subsection presents the Python script used to perform a feature abation analysis on the recorded dataset. It systematically removes either entire feature groups (e.g, image, audio, activity) or individual features (one at a time) to measure their impact on classification accuracy. As recommended by the official TensorFlow tutorials (Developers, n.d.b) the trained Keras model was load via `tf.keras.models.load_model`. Similarly the confusion matrix and classification report are computed following scikit-learn's guidelines (Pedregosa et al., n.d.).

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

##############################################################################
# 1) Globale Parameter & Pfade (hiernur ein bissl anpassen)
##############################################################################
CSV_PATH = r"C:\Users\lpera\OneDrive\Desktop\AirquixAppThesis\dataset\
    Campaign_Dataset\campaign_02.csv"
MODEL_PATH = r"C:\Users\lpera\OneDrive\Desktop\Uni\Kopie\tf_nn_model.keras"

# Spalten, die das Model erwartet
STRING_COLUMNS = [
    "SCENE_TYPE",
    "PLACES_top1", "PLACES_top2", "PLACES_top3", "PLACES_top4", "PLACES_top5",
    "YAMNET_top1", "YAMNET_top2", "YAMNET_top3",
    "VEHICLE_audio_1", "VEHICLE_audio_2", "VEHICLE_audio_3",
    "VEHICLE_image_1", "VEHICLE_image_2", "VEHICLE_image_3",
    "ACT"
]
FLOAT_COLUMNS = [
    "places_top1_conf", "places_top2_conf", "places_top3_conf",
    "places_top4_conf", "places_top5_conf",
    "YAMNET_conf_1", "YAMNET_conf_2", "YAMNET_conf_3",
    "vehicle_audio_conf_1", "vehicle_audio_conf_2", "vehicle_audio_conf_3",
    "vehicle_image_conf_1", "vehicle_image_conf_2", "vehicle_image_conf_3",
    "noise_dB", "ACT_confidence", "speed_m_s"
]
ALL_MODEL_COLUMNS = STRING_COLUMNS + FLOAT_COLUMNS

# Label-Mapping
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()}

# Gruppen fr "No Image", "No Audio", "No Activity"
IMAGE_FEATURES = [
    "SCENE_TYPE", "PLACES_top1", "places_top1_conf",
    "PLACES_top2", "places_top2_conf",
    "PLACES_top3", "places_top3_conf",
    "PLACES_top4", "places_top4_conf",
    "PLACES_top5", "places_top5_conf",
    "VEHICLE_image_1", "vehicle_image_conf_1",
    "VEHICLE_image_2", "vehicle_image_conf_2",
    "VEHICLE_image_3", "vehicle_image_conf_3",
]
AUDIO_FEATURES = [
    "YAMNET_top1", "YAMNET_conf_1",
    "YAMNET_top2", "YAMNET_conf_2",
    "YAMNET_top3", "YAMNET_conf_3",
    "VEHICLE_audio_1", "vehicle_audio_conf_1",
    "VEHICLE_audio_2", "vehicle_audio_conf_2",
    "VEHICLE_audio_3", "vehicle_audio_conf_3",
    "noise_dB"
]
ACTIVITY_FEATURES = [
    "ACT", "ACT_confidence", "speed_m_s"
]


################################################################################
# 2) Hilfsfuncktionen (machmal ganz ntzlich)
################################################################################
def build_input_dict(df, used_cols):
    """
    Erzeugt ein dictionary {colName -> tf.Tensor} fr das model.
    Fehlt eine spalte => dummy (0 oder "dummy").
    Ja, wir sind so clever.
    """
    N = len(df)
    input_dict = {}

    for col in ALL_MODEL_COLUMNS:
        if col in used_cols and col in df.columns:
            # normele werte
            if col in STRING_COLUMNS:
                arr_str = df[col].astype(str).fillna("dummy").values
                input_dict[col] = tf.constant(arr_str, dtype=tf.string)
            else:
                arr_flt = pd.to_numeric(df[col], errors="coerce").fillna(0.0).
```

```
                values
            input_dict[col] = tf.constant(arr_flt, dtype=tf.float32)
        else:
            # dummy-werte
            if col in STRING_COLUMNS:
                dummy_str = np.array(["dummy"]*N)
                input_dict[col] = tf.constant(dummy_str, dtype=tf.string)
            else:
                dummy_flt = np.zeros(N, dtype=np.float32)
                input_dict[col] = tf.constant(dummy_flt, dtype=tf.float32)

    return input_dict

def remove_feature_group(df, group):
    """
    Entfernt alle Spalten aus 'group' aus df, wnen sie existieren.
    """
    keep_cols = [c for c in df.columns if c not in group]
    return df[keep_cols].copy()

def evaluate_scenario(df, model, scenario_cols, classes_in_data):
    """
    Fhrt vorhersagen mit 'scenario_cols' (alle Features auer dem/den entfernten
        )
    durch und gibt (accuracy, macro_f1) zurck.
    Ich hoffe wir bekommen was gutes raus.
    """
    input_dict = build_input_dict(df, scenario_cols)
    preds_prob = model.predict(input_dict, batch_size=64, verbose=0)
    preds_int = np.argmax(preds_prob, axis=1)

    cm = confusion_matrix(df["gt_num"], preds_int, labels=classes_in_data)
    acc = np.trace(cm) / np.sum(cm)

    rep_dict = classification_report(
        df["gt_num"], preds_int,
        labels=classes_in_data,
        output_dict=True,
        zero_division=0
    )
    macro_f1 = rep_dict["macro avg"]["f1-score"]
    return acc, macro_f1


##############################################################################
# 3) Hauptablauf (muss ja sein)
##############################################################################
def main():
```

```
# (A) Daten laden & aufbereiten
df = pd.read_csv(CSV_PATH, delimiter=",", encoding="utf-8")
df["timestamp"] = pd.to_datetime(df["timestamp"], errors="coerce")
df = df.dropna(subset=["timestamp"]).sort_values("timestamp").reset_index(
    drop=True)

df["status_gt"] = df["status_gt"].astype(str).str.strip().str.lower()
df["gt_num"] = df["gt_num"].map(label_mapping)
df = df.dropna(subset=["gt_num"]).copy()
df["gt_num"] = df["gt_num"].astype(int)

classes_in_data = sorted(df["gt_num"].unique())
print("Klassen im Datensatz:", classes_in_data)

# wir laden das model
model = tf.keras.models.load_model(MODEL_PATH)
print("Model loaded successfuly. yuhu")

# (B) BASELINE: Alle Features
acc_all, f1_all = evaluate_scenario(df, model, ALL_MODEL_COLUMNS,
    classes_in_data)
print(f"\n=== Baseline: ALL FEATURES ===\nAccuracy={acc_all:.3f}, Macro-F1
    ={f1_all:.3f}")

# (C) Szenario-Ablation: No Image, No Audio, No Activity
df_no_image = remove_feature_group(df, IMAGE_FEATURES)
df_no_audio = remove_feature_group(df, AUDIO_FEATURES)
df_no_activity = remove_feature_group(df, ACTIVITY_FEATURES)

acc_no_image, f1_no_image = evaluate_scenario(df_no_image, model,
    df_no_image.columns, classes_in_data)
acc_no_audio, f1_no_audio = evaluate_scenario(df_no_audio, model,
    df_no_audio.columns, classes_in_data)
acc_no_activity, f1_no_activity = evaluate_scenario(df_no_activity, model,
    df_no_activity.columns, classes_in_data)

scenario_data = [
    ("All Features", acc_all),
    ("No Image", acc_no_image),
    ("No Audio", acc_no_audio),
    ("No Activity", acc_no_activity),
]

# (D) Leave-One-Out Ablation: einzelne Spalte weglassen
ablation_list = []
for col_to_remove in ALL_MODEL_COLUMNS:
    scenario_cols = [c for c in ALL_MODEL_COLUMNS if c != col_to_remove]
    acc_s, f1_s = evaluate_scenario(df, model, scenario_cols,
```

```
        classes_in_data)
    ablation_list.append((col_to_remove, acc_s))

ablation_list.sort(key=lambda x: x[1], reverse=True)
baseline_accuracy = acc_all
features = [x[0] for x in ablation_list]
accuracies = [x[1] for x in ablation_list]
colors = ["limegreen" if a >= baseline_accuracy else "orange" for a in
    accuracies]

# wir sind so stylisch
sns.set_style("whitegrid")
sns.set_context("notebook", rc={
    "axes.titlesize": 14,
    "axes.labelsize": 12,
    "xtick.labelsize": 10,
    "ytick.labelsize": 10
})

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(16, 8))

# 1) Linker Plot: Gruppen-Ablation
ax0 = axes[0]
x_names = [s[0] for s in scenario_data]
x_accs = [s[1] for s in scenario_data]

bars = ax0.bar(x_names, x_accs, color="skyblue", edgecolor="black")
ax0.axhline(baseline_accuracy, color="red", linestyle="--", label=f"
    Baseline (all features)")

for bar in bars:
    h = bar.get_height()
    ax0.text(
        bar.get_x() + bar.get_width()/2,
        h + 0.005,
        f"{h:.3f}",
        ha="center", va="bottom", fontsize=10, fontweight="bold"
    )

ax0.set_ylabel("Accuracy")
ax0.set_ylim(0, 1.0)
ax0.set_title("Ablation by Feature Groups")
ax0.legend(loc="best")

# 2) Rechter Plot: Leave-One-Out
ax1 = axes[1]
bars2 = ax1.barh(features, accuracies, color=colors, edgecolor="black")
ax1.invert_yaxis()
```

```
    ax1.axvline(baseline_accuracy, color="red", linestyle="--", label=f"
        Baseline ({baseline_accuracy:.3f})")

    ax1.set_xlabel("Accuracy After Single-Feature Removal")
    ax1.set_title("Leave-One-Out Ablation")
    ax1.legend(loc="lower right")

    ax1.set_xlim(0.78, 0.89) # ab 0.75? na dann

    plt.subplots_adjust(wspace=0.4)
    plt.tight_layout()

    plt.savefig("combined_ablation_plot.png", dpi=300, bbox_inches="tight")
    plt.show()

    print("\nDone. Die Ergebnisse wurden in 'combined_ablation_plot.png'
        gespeichert.")

if __name__ == "__main__":
    main()
```

Listing 7: Python Script for Feature Ablation

Explanation of Key Steps:

- **Feature Groups vs. Single-Feature Removal:** First removes entire categories (image, audio, activity) to measure how each groupe affects accuracy; then systematically removes one feature at a time (e.g, `noise_dB` or `PLACES_top1`) to see which columns are individually most important.
- **Baseline Accuracy:** The code calculate a baseline accuracy using al features. Any removal scenario is compared against this baseline to highligh which features significantly reduce performance.

A.9 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
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)
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: matrix anzeigen (hoffentlich sind die Vorhersagen nicht komplet
    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
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 8: 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.

A.10 Building the `Airquix` Android App

This section summarizes the **Airquix** Android App. It outlines the projekt structure, key components and includes short code excerpts illustrating the most important functionality. For full listings, build instructions and troubleshoting, please refer to the Electronic Appendix (Section A.10).

Project Structure

- `Airquix01/`: Root folder of the Android Studio project.
- `app/`: Contains all source code and resources.
  - `AndroidManifest.xml`: Declares permissions and registers components.
  - `java/` (or `kotlin/`): Houses the main Activity, Service, and ViewModel files.
  - `res/`: Layouts, strings, themes, and other resources (e.g., `file_paths.xml`).
  - `assets/`: TFLite models (e.g., `model.tflite`, `vehicle_sounds.tflite`) and label files.
- `build.gradle.kts` (Module: app): Module-level Gradle settings (dependencies, Compose options).
- `build.gradle.kts` (Project-level): Global Gradle configuration.

AndroidManifest.xml

Below is a snippet showing how the app requests camera, audio, location and activity recognition permissions As described in the official Android manifest documentation (LLC, n.d.d), corresponding permissions and services must be defined in `AndroidManifest.xml`. must be defined. it also declares a `LoggingService` for foreground operation:

```
<manifest package="com.example.airquix01"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <!-- Required Permissions -->
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.RECORD_AUDIO" />
    <uses-permission android:name="android.permission.ACTIVITY_RECOGNITION" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    ...

    <application ... >
        <!-- MainActivity -->
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <!-- LoggingService -->
        <service
            android:name=".LoggingService"
            android:exported="false"
```

```
            android:foregroundServiceType="camera|microphone" />
        ...
    </application>
</manifest>
```

Listing 9: Excerpts from `AndroidManifest.xml`.

MainActivity (UI and Permissions)

`MainActivity` manages runtime permissions, starts/stops the logging service And presents a Jetpack Compose UI for clearing or sharing CSV logs and setting a manuall`status_gt` JetBrains (n.d.), LLC (n.d.b).

```
class MainActivity : ComponentActivity() {

    // Checks and requests camera, audio, location permissions
    private fun checkAndRequestPermissions() {
        val needed = mutableListOf<String>()
        if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
            != PackageManager.PERMISSION_GRANTED) {
            needed.add(Manifest.permission.CAMERA)
        }
        // ... similarly for audio, location, activity recognition, etc.

        if (needed.isNotEmpty()) {
            requestPermissionLauncher.launch(needed.toTypedArray())
        }
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        checkAndRequestPermissions()

        setContent {
            MymlkitappTheme {
                Scaffold(
                    topBar = { MyTopBar(...) },
                ) { innerPadding ->
                    MainScreen(
                        onStartLogging = { startLoggingService() },
                        onStopLogging = { stopLoggingService() },
                        onClearLogs = { viewModel.clearAllLogs() },
                        onShareLogs = { shareLogsCsv() },
                        ...
                    )
                }
            }
        }
    }
```

```
    private fun startLoggingService() {
        val intent = Intent(this, LoggingService::class.java)
        ContextCompat.startForegroundService(this, intent)
    }

    private fun stopLoggingService() {
        stopService(Intent(this, LoggingService::class.java))
    }
}
```

Listing 10: Excerpt from `MainActivity.kt`.

LoggingService (Core Data Capture)

`LoggingService` runs as a foreground service, capturing images and audio periodicaly. It then classifies the data using Places365 (PyTorch) and TFLite (YamNEt, vehicle model), logs sensor readings (GPS speed, noise) and append everything to a CSV file (LLC, n.d.c, JetBrains and Contributors, n.d.).

```
class LoggingService : LifecycleService() {
    private val serviceScope = CoroutineScope(SupervisorJob() + Dispatchers.
        Default)

    override fun onCreate() {
        super.onCreate()
        startForegroundServiceNotification()
        setupCamera() // ImageCapture + Places365 classification
        startYamNet() // Audio classification (ambient sounds)
        startVehicleModel()// Additional classification for vehicle sounds
        startPeriodicLogging()
    }

    private fun startPeriodicLogging() {
        serviceScope.launch {
            while (isActive) {
                // Collect sensor data: speed, noise, recognized scene & audio
                // Append everything to CSV
                viewModel.appendLog(...)
                delay(5000L)
            }
        }
    }
    ...
}
```

Listing 11: Excerpt from `LoggingService.kt`.

MainViewModel (Shared State and Logging)

`MainViewModel` (LLC, n.d.a) stores all real-time outputs (scene recognition, audio classification, etc), along with user inputs (`status_gt`). It also handles CSV logging and clearing

```
class MainViewModel : ViewModel() {
    val logList = mutableStateListOf<String>()
    val currentSpeed = mutableStateOf(0f)
    val currentStatusGt = mutableStateOf("Unknown")
    // ... other states (camera classification, audio classification, etc.)

    fun appendLog(...) {
        // Build CSV line with classification results, sensor data, status_gt
        // Add to logList, write to disk
    }

    fun clearAllLogs() {
        logList.clear()
        // Reset or overwrite CSV file
    }
}
```

Listing 12: Excerpt from `MainViewModel.kt`.

AirquixApplication (Global ViewModel)

`AirquixApplication` extends `Application` and creat a single `MainViewModel` instance that persists across configuration changes.

```
class AirquixApplication : Application(), ViewModelStoreOwner {
    private val appViewModelStore = ViewModelStore()
    private lateinit var mainViewModel: MainViewModel

    override fun onCreate() {
        super.onCreate()
        mainViewModel = ViewModelProvider(this)[MainViewModel::class.java]
    }

    override val viewModelStore: ViewModelStore
        get() = appViewModelStore

    fun getMainViewModel() = mainViewModel
}
```

Listing 13: Excerpt from `AirquixApplication.kt`.

Usage Flow

- **Permissions**: On first launch, camera, audio, location and other permissions are requested.
- **Start Logging**: Activates the foreground service for image/audio capture and classification.
- **Stop Logging**: Halts the service and stops periodic data capture.
- **Set Status (`status_gt`)**: Allows manual labeling of context (e.g., *vehicle in bus*, *indoor closed*).
- **Share Logs**: Exports the `all_in_one_logs.csv` file using Android's share sheet.

This design enables real-time collection and classification of environmental data, combining multiple sensor streams (camera, microphone, GPS) and machine-learning models (Places365, YamNet, vehicle classifier). For further details and the complete code, see the repository in the Electronic Appendix.

Electronic Appendix

Data, code, and figures are provided in electronic form at `https://github.com/LouAirquix/AirquixAppThesis`. This repo includes comprehenive 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 sometimes used for debugging and programming assistance. 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.

CSAILVision (n.d.a). convert_model.py, `https://github.com/CSAILVision/places365/blob/master/convert_model.py`. Accessed on 28.02.2025.

CSAILVision (n.d.b). demo_pytorch_cam.py, `https://github.com/CSAILVision/places365/blob/master/demo_pytorch_CAM.py`.

CSAILVision (n.d.c). run_placescnn_basic.py, `https://github.com/CSAILVision/places365/blob/master/run_placesCNN_basic.py`.

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

Developers, T. (n.d.a). Tensorflow lite model conversion, `https://www.tensorflow.org/lite/convert/python_api`. Accessed on 28.02.2025.

Developers, T. (n.d.b). Tensorflow tutorials, `https://www.tensorflow.org/tutorials`. Accessed on 28.02.2025.

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*.

JetBrains (n.d.). Kotlin language documentation, `https://kotlinlang.org/docs/home.html`. Accessed on 28.02.2025.

JetBrains and Contributors (n.d.). Kotlin coroutines documentation, `https://kotlinlang.org/docs/coroutines-overview.html`. Accessed on 21.02.2025.

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

LLC, G. (n.d.a). Guide to android architecture components (viewmodel), `https://developer.android.com/topic/libraries/architecture/viewmodel`. Accessed on 20.02.2025.

LLC, G. (n.d.b). Jetpack compose - android's modern toolkit for building native ui, `https://developer.android.com/jetpack/compose`. Accessed on 28.02.2025.

LLC, G. (n.d.c). Lifecycle-aware components, `https://developer.android.com/topic/libraries/architecture/lifecycle`. Accessed on 21.02.2025.

LLC, G. (n.d.d). Manifest documentation, `https://developer.android.com/guide/topics/manifest/manifest-intro`. Accessed on 28.02.2025.

MarkovML (n.d.). Ml model validation.
URL: *https://www.markovml.com/blog/ml-model-validation*

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.

Pedregosa, F., Varoquaux, G., Gramfort, A. et al. (n.d.). Scikit-learn: Machine learning in python, `https://scikit-learn.org/stable`. Accessed on 22.02.2025.

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*.

sriharsha1507 (n.d.). user_activity_detection, `https://github.com/sriharsha1507/user_activity_detection/`. Accessed on 18.02.2025.

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.

Waskom, M. L. and the seaborn development team (n.d.). Seaborn documentation, `https://seaborn.pydata.org/`. Accessed on 21.02.2025.

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

ws-dl (2021). Machine learning on mobile.
URL: *https://ws-dl.blogspot.com/2021/12/2021-12-20-machine-learning-on-mobile.html*

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