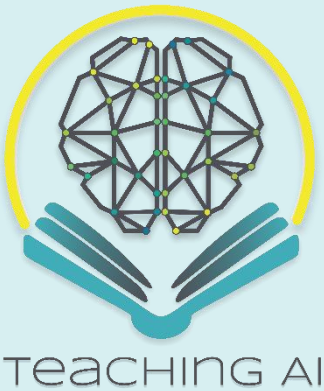

Report WP3-A3:

Implementation of the selected databases into the Python programming language (Jupyter notebook)

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Education and Culture Executive Agency. (EACEA). Neither the European Union nor EACEA can be held responsible for them.



Result

Implementation of selected video, image, audio/speech and numerical datasets in Python and Jupyter Notebooks.

Related to

WP3-A3: Implementation of the selected databases into the Python programming language (Jupyter notebook)

Statement of originality

This report contains original unpublished work, except where clearly indicated otherwise.

Disclaimer

This report contains material, which is the copyright of TAI Consortium Parties. All TAI Consortium Parties have agreed that the content of the report is licensed under a Creative Commons Attribution Non-Commercial Share Alike 4.0 International License. TAI Consortium Parties does not warrant that the information contained in the Report is capable of use, or that use of the information is free from risk and accept no liability for loss or damage suffered by any person or any entity using the information.

Copyright notice

© 2024-2027 TAI Consortium Parties

Note

For anyone interested in having more information about the project, please see the website at: https://tai-erasmus.eu/

This publication is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International Public License (CC BY-NC-SA 4.0).

Document version history

Version	Date	Author	Comment
0.1	20.11.2026.	Jelena Šaković Jovanović	First version

Table of contents

1. Introduction 4

2. Scope and organisation 5

3. Common notebook structure 6

4. Video recordings 7

5. Still images 8

6. Audio/speech inputs 9

7. Numerical and industrial data 10

8. Execution and supporting resources 11

9. Final conclusions 12

Annex 1. Python code used for implementation methods 13

1. Introduction

The implementation of selected datasets in Python makes the data accessible through executable examples that combine explanations, preparation functions and visible outputs. Jupyter Notebooks provide the working environment in which learners can load a dataset, inspect its structure, identify inputs and targets and obtain the representations required for subsequent AI exercises.

This report describes the Python organisation of the available implementations. Video recordings, still images, audio/speech inputs and numerical and industrial data are presented in that order. The final conclusions summarise the implementation structure, its educational use and the resources required for reuse.

WP3-A3 concerns the implementation structure and access to the data. The accompanying WP3-A4 report describes the cleaning, conversion, validation and reduction operations in greater detail. The notebooks also contain later model-training and interactive sections; this report focuses on the parts that load, inspect, represent and prepare the datasets.

Coordination field	Details
WP3 lead organisation	University of Montenegro (UoM)
Responsible person	Jelena Šaković Jovanović
Report author	Jelena Šaković Jovanović

2. Scope and organisation

The available material comprises two video examples, one still-image example, two audio/speech examples and three numerical examples. The examples differ in the unit of observation: a video clip, a second within a driving clip, a colour image, a complete spoken-digit recording, a short tone window or a row in a numerical table. The notebook configuration and input/target definition make these differences explicit.

Data type	Implemented example	Notebook or associated code
Video recordings	Popeye binary motion classification	TAI_Popeye_3D_CNN_classification.ipynb
Video recordings	Garmin vehicle speed estimation	TAI_Video_Speed_Estimation.ipynb and four helper scripts
Still images	CIFAR-10 object classification	TAI_cifar10_classification.ipynb
Audio/speech inputs	Free Spoken Digit Dataset (FSDD)	TAI_fsdd_audio_classification.ipynb
Audio/speech inputs	Sopele Sounds	TAI_sopele_audio_classification.ipynb
Numerical and industrial data	Concrete compressive strength	TAI_concrete_regression.ipynb
Numerical and industrial data	Meteorological and air-quality time series	TAI_meteo_timeseries.ipynb
Numerical and industrial data	Loan classification	TAI_loan_classification.ipynb

The inventory identifies the implementations covered by this report. It does not imply that every dataset considered during the earlier search has a separate notebook. Source data, annotation files and the notebook are treated as distinct resources, with access arranged according to the applicable conditions.

3. Common notebook structure

The notebooks follow a common progression while retaining the functions needed for each data type. Reusable methods are defined before they are applied to a dataset. The user can inspect intermediate tables, plots or media examples before proceeding to the model-facing representation.

Stage	Implementation role	Typical visible output
Configuration	Select data location, preparation settings, output folder and random seed.	Paths and active settings.
Data access	Read tables, index WAV files or match videos with annotations.	File inventory, dimensions and metadata.
Inspection	Review values, labels, sample media and data quality.	Summaries, plots, audio players and sampled frames.
Input and target definition	Separate model inputs from the value or class to be predicted.	Named columns, class mapping or time-aligned targets.
Preparation interface	Apply reusable functions and create model-facing arrays.	Prepared arrays, labels, partition records and transformation settings.
Persistence	Retain prepared data and metadata for the following notebook stages.	Joblib files, NumPy arrays, CSV manifests or configuration records.

A notebook can be rerun from its configuration and function-definition cells. The applicable source files and helper scripts must remain accessible at the configured paths. Output folders distinguish generated resources from the original data.

4. Video recordings

4.1 Popeye binary video classification

The Popeye notebook loads a local MP4 file and reads its frame rate, frame count, dimensions and duration. A clip table associates each candidate observation with an identifier and a start and end time. The default observation spans 2 seconds and contains 16 sampled RGB frames resized to 64 by 64 pixels with padding that preserves aspect ratio.

X is the sequence of frames within a clip. y is the manually assigned class low_motion or high_motion. The supplied clip_annotations.csv contains the clip identifier, time interval, split and label. The loader checks identifiers and their correspondence with the current clip configuration before labels are attached to the observations.

Representative-frame displays and an optional sampled-frame animation support inspection. Clips are assigned to train, validation and test according to their temporal position, with unused gaps around the boundaries. Unlabelled, skipped and gap observations are excluded from the model-facing subset. The resulting tensors retain the time dimension rather than treating individual frames as independent labelled observations.

4.2 Garmin vehicle speed estimation

The Garmin example combines a notebook with run.py, speednet.py, train_random_val.py and export_garmin_labels.py. The notebook presents the steps and calls the reusable algorithms through the selected Python interpreter. An environment diagnostic checks the libraries and external utilities required by the workflow.

The input resources are local video files and matching speed CSVs. Each video stem identifies the associated speed file. A separate drive_groups.csv associates files with continuous driving sessions, and fixed_test_drive.csv records the chosen held-out test session.

X is a 10-second optical-flow sequence extracted from the video pixels. y contains vehicle speeds at one-second intervals, with a mask specifying which target values are available. A default sample contains 50 two-channel flow maps derived at 5 frames per second and 10 speed targets. GPS speed supplies the supervised target; it is not an inference input.

Video example	Access and inspection functions	Prepared representation
Popeye	Video metadata, clip table, manual-label CSV, representative frames and optional GIF.	RGB clips, binary labels and temporal partition membership.
Garmin	Video inventory, matching CSV audit, drive mapping and crop preview.	Optical-flow windows, target vectors, validity masks and a source manifest.

5. Still images

5.1 CIFAR-10 dataset and notebook

TAI_cifar10_classification.ipynb implements object classification from local image folders. The notebook describes CIFAR-10 as 60,000 colour images of 32 by 32 pixels in ten classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. It expects data/cifar10/train/ and data/cifar10/test/, each with a subfolder for every class. Labels are assigned from the sorted class-directory names, so both roots must use the same class names and ordering.

5.2 Data loading and sample inspection

Reusable methods load files through Pillow, convert them to RGB and resize images whose dimensions differ from the configured 32 by 32 pixels. The default configuration samples 1,500 training-pool images and up to 1,000 test images per class using seed 42. Overview tables, class counts, example grids, pixel-value displays, colour-space panels and histograms make the image representation visible. Additional functions inspect unreadable files, exact pixel duplicates, overlap between the two pools and unusual brightness or contrast.

5.3 Input and target representation

One image is one observation. X is an RGB array with shape 32 by 32 by 3, initially stored as uint8 values from 0 to 255. y is the integer object-class label. A class selector keeps at least two classes and renumbers their labels consistently; a display illustrates labels and one-hot vectors. A stratified 85/15 split divides the retained training pool into training and validation data while the test pool remains separate. The default scaling divides pixels by 255; channel-wise standardisation fitted on training images is also implemented.

5.4 Implementation outputs

The notebook saves unscaled partition arrays and integer targets in output_cifar10/prepared.npz. The companion prepared_meta.joblib stores class names, image size, scaling settings and test-file paths. Scaling is applied when these resources are loaded for later model stages. The supplied notebook contains recorded outputs for 12,750 training images, 2,250 validation images and 10,000 test images with all ten classes retained. Its later training and interactive sections consume these resources; the annex reproduces the preparation definitions.

6. Audio/speech inputs

6.1 Free Spoken Digit Dataset

The FSDD notebook indexes WAV recordings and reads digit, speaker and repetition identifiers from their filenames. One recording is one observation. X is an audio representation and y is the spoken digit from 0 to 9. Speaker and repetition identifiers are retained as metadata rather than included in the predictors.

The notebook provides waveform, log-mel and MFCC inspection alongside playback of the complete source recording and the processed signal used by the model. Configuration controls the sampling rate, duration cap, spectral parameters and optional silence trimming. The default sampling rate is 8 kHz and the duration cap is 2.5 seconds.

The preparation interface returns an 80-value pooled MFCC/delta feature vector per recording, padded MFCC sequences and padded log-mel arrays. The standard repetition-based partition and the optional speaker-independent partition are explicit alternatives. Prepared arrays, sequence lengths, labels, scaler information and the recording index are saved for reuse.

6.2 Sopele Sounds

The Sopele notebook accepts the supplied archive, the dataset archive or an extracted data folder. It indexes the original single-tone files, checks the tone labels and assigns whole source recordings to partitions before creating short windows. The complete source recording can be played independently of the window selector.

For TASK=single, X is a short waveform window or its frequency representation and y is one of 12 tone labels, m0–m5 and v0–v5. For TASK=combined, source windows from the same partition are mixed to create 36 two-tone classes. The default window and hop duration is 0.10 seconds at 22.05 kHz.

The feature mode offers waveform or FFT-based vectors, with a log-mel sequence representation also prepared. The resulting object retains labels, arrays, source indices and window-level metadata. Experiment-specific output folders separate different task and preparation configurations.

Audio example	Observation and target	Reusable preparation output
FSDD	One recording; digit 0–9.	Pooled features, MFCC sequences, log-mel arrays, lengths and partition index.
Sopele	One short window; single-tone or generated two-tone label.	Waveforms/features, log-mel sequences, labels and source/window indices.

7. Numerical and industrial data

7.1 Shared table interface

The three numerical notebooks use CSV input and a common set of functions for loading, summary statistics, missing values, duplicates and outlier inspection. Interactive selectors identify input columns, the output column and any categorical columns to encode. The feature list and transformation objects are retained with the prepared arrays.

7.2 Concrete compressive strength

The concrete notebook reads Concrete_Data.csv. The default output is Strength, while the selected remaining columns form X. The notebook reports table dimensions and variable types, allows the user to inspect and optionally remove selected outlier rows and prepares a shuffled 80/20 train/test partition. Inputs and the continuous target are scaled to 0–1 using training-fitted scalers.

7.3 Meteorological and air-quality time series

The time-series notebook reads data_thesis_4_1.csv. A PM2.5 column is the default target; PM10 is excluded from the default input selection. The initial 80/20 partition preserves the existing row order without shuffling. The preparation output retains the input names, scalers and past-value examples used by the subsequent sequence-based workflow.

The later sequence interface constructs lookback windows from the selected inputs and measured past target values to predict the following target value. Preserving ordered observations in the data-preparation stage provides the basis for that interface; window construction and model training remain separate notebook steps.

7.4 Loan classification

The loan notebook reads loan_data.csv and selects loan_status as the default target. X consists of the chosen applicant attributes. The notebook displays the class balance, supports categorical-column selection and converts selected categories to n−1 dummy columns. A shuffled 80/20 split is followed by training-fitted input scaling. The target remains a class label.

Notebook	Default target	Preparation output
Concrete regression	Strength	Scaled input/target arrays, feature names and scalers.
Meteorological time series	PM2.5	Ordered input/target arrays, scalers and input-history metadata.
Loan classification	loan_status	Scaled input arrays, class labels, encoded feature names and scaler.

8. Execution and supporting resources

Each numerical notebook writes prepared.joblib in its own output folder. It includes the prepared arrays, target identity, feature names and the additional input metadata needed by later prediction controls. Dataset-specific output folders should be retained when the notebooks are distributed.

8.1 Files and execution environment

The notebook, its input files and any helper scripts should be arranged according to the configured paths. CIFAR-10 requires matching class subfolders under data/cifar10/train/ and data/cifar10/test/; numerical examples require their CSVs; FSDD uses recordings/; Sopele accepts a local archive or extracted folder; the video examples require the original videos and associated annotations. The Garmin notebook must remain beside its four Python scripts.

Audio packages include requirements files and starting instructions. TAI_export_environment.ipynb provides a separate environment-export utility for recording package versions. Video execution also depends on the media and metadata utilities used by the corresponding scripts. The environment specification distributed with each example should match the kernel used to run it.

8.2 Verification outputs and access records

The implementations expose checks and intermediate outputs that can be retained when the final package is run: loaded dimensions, recording or video inventories, annotation checks, sample representations and prepared-array shapes. Preparation summaries should be distinguished from training-performance results.

Resource	Location or access information
Notebook and helper-code package	[Repository or project-platform link to be added]
Dataset access instructions	[Dataset-specific locations and access conditions to be added]
Execution record	[Environment details and retained preparation outputs to be added]

9. Final conclusions

9.1 Consolidated implementation summary

The eight examples provide a common Python and Jupyter structure for accessing two video datasets, one still-image dataset, two audio/speech datasets and three numerical datasets. Each example makes the unit of observation, predictors and target explicit. Reusable functions are defined before their application, while tables, plots and media previews make the transition from source files to model-facing arrays visible to learners.

9.2 Educational use and implementation choices

A shared notebook progression supports teaching across the four modalities without forcing identical representations. Popeye uses labelled RGB clips, Garmin combines optical-flow inputs with aligned speed targets, CIFAR-10 uses colour images with directory-based object labels, FSDD represents complete spoken-digit recordings, and Sopele derives windows from source recordings. The numerical examples retain named predictors, selected targets and transformation information. These differences demonstrate why data access, observation boundaries and partition rules must be chosen for each dataset.

9.3 Resource and execution requirements

The implementation provides a basis for subsequent model training and interactive experiments. Reuse requires the original input files, matching annotations, helper scripts and compatible packages at the configured locations. Prepared arrays and metadata should be retained together with the active settings and execution records. The source listings document the implementation; confirmation of complete execution and resulting dataset counts must come from running the corresponding packages with their required resources.

Annex 1. Python code used for implementation methods

The listings below reproduce preparation code from the accompanying notebooks and helper scripts. Filenames and notebook cell numbers identify each listing. Definitions are collected in the A3 report and their application is shown in the A4 report. The original notebooks retain the complete execution order and dependencies. Model-training and prediction sections are outside these listings.

A1.1 Video recordings — Popeye methods

Source file: TAI_Popeye_3D_CNN_classification.ipynb. Notebook cells 3, 5, 6, 7, 8, 9, 10, 11.

Cell 3

```
# Uncomment once in a fresh environment if dependencies are unavailable:
# %pip install numpy pandas matplotlib opencv-python torch scikit-learn Pillow ipywidgets

from pathlib import Path
import copy
import hashlib
import json
import random

import cv2
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from IPython.display import display
from sklearn.metrics import (accuracy_score, f1_score, classification_report,
                             confusion_matrix, ConfusionMatrixDisplay,
                             roc_auc_score)

SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(SEED)
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
if DEVICE.type == 'cpu':
    torch.set_num_threads(min(4, max(1, torch.get_num_threads())))
print('PyTorch:', torch.__version__, '| device:', DEVICE)

VIDEO_FILE = Path('Popeye_the_Sailor_Meets_Sindbad_the_Sailor.mp4')
OUT_DIR = Path('output_popeye_binary_3dcnn')
MODEL_DIR = OUT_DIR / 'models'
MODEL_DIR.mkdir(parents=True, exist_ok=True)
ANNOTATION_CSV = OUT_DIR / 'clip_annotations.csv'

# Only TWO human-annotated classes; the order determines class IDs 0 and 1.
CLASS_NAMES = ('low_motion', 'high_motion')
assert len(CLASS_NAMES) == 2 and len(set(CLASS_NAMES)) == 2

# Non-overlapping clip extraction parameters.
CLIP_SECONDS = 2.0
STRIDE_SECONDS = 2.0
NUM_FRAMES = 16
IMAGE_SIZE = 64
MAX_CLIPS = 1000
SPLIT_GAP_SECONDS = 2.0   # temporal gaps limit near-duplicate leakage

# Default values
FILTER_GRID = [4]
BATCH_SIZE = 30
EPOCHS = 100
LEARNING_RATE = 5e-4
PATIENCE = 10
```

Cell 5

```
# Inspect basic video metadata. OpenCV reports FPS, number of frames and dimensions.
def inspect_video(path):
    path = Path(path)
    if not path.is_file():
        raise FileNotFoundError(
            f"Could not find {path.resolve()}. Put the MP4 next to the notebook "
            "or change VIDEO_FILE in the setup cell."
        )
    cap = cv2.VideoCapture(str(path))
    if not cap.isOpened():
        raise ValueError(f'OpenCV could not open {path}. Check the file and installed video
codecs.')
    fps = float(cap.get(cv2.CAP_PROP_FPS))
    n_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    cap.release()
    if not np.isfinite(fps) or fps <= 0 or n_frames <= 0 or width <= 0 or height <= 0:
        raise ValueError('Invalid video metadata (FPS, frame count or dimensions).')
    info = dict(path=str(path.resolve()), fps=fps, frame_count=n_frames,
                width=width, height=height, duration_s=n_frames / fps)
    print(f"Video: {path.name} | {width}x{height} | {fps:.2f} FPS | "
          f"{n_frames:,} frames | {info['duration_s'] / 60:.1f} min")
    return info
```

Cell 6

```
# Make a table of candidate clips. Each row is ONE observation (not one frame).
def make_clip_table(info, clip_seconds, stride_seconds, max_clips):
    if clip_seconds <= 0 or stride_seconds < clip_seconds or max_clips < 1:
        raise ValueError('Use positive clip duration, non-overlapping stride and max_clips >=
1.')
    latest_start = info['duration_s'] - clip_seconds - 1.0 / info['fps']
    if latest_start <= 0:
        raise ValueError('The video is shorter than one complete clip.')
    starts = np.arange(0.0, latest_start + 1e-8, stride_seconds)
    if len(starts) > max_clips:
        positions = np.linspace(0, len(starts) - 1, max_clips).round().astype(int)
        starts = starts[positions]
    return pd.DataFrame({
        'clip_id': [f'clip_{i:04d}' for i in range(len(starts))],
        'start_s': starts,
        'end_s': starts + clip_seconds,
    })
```

Cell 7

```
# Resize an RGB frame without stretching characters or changing the aspect ratio.
# Empty pixels are black padding around the image.
def letterbox_rgb(image_rgb, size):
    h, w = image_rgb.shape[:2]
    scale = size / max(h, w)
    nw, nh = max(1, round(w * scale)), max(1, round(h * scale))
    resized = cv2.resize(image_rgb, (nw, nh), interpolation=cv2.INTER_AREA)
    canvas = np.zeros((size, size, 3), dtype=np.uint8)
    y0, x0 = (size - nh) // 2, (size - nw) // 2
    canvas[y0:y0 + nh, x0:x0 + nw] = resized
```

```
    return canvas

# Read NUM_FRAMES evenly spaced across a clip. We seek once, then decode forwards.
# Output shape: (time, height, width, RGB) = (T, H, W, C), dtype uint8.
def read_clip(cap, fps, start_s, clip_seconds, num_frames, image_size):
    start_frame = max(0, int(round(start_s * fps)))
    available = max(1, int(round(clip_seconds * fps)))
    target_frames = np.rint(np.linspace(start_frame, start_frame + available - 1,
                                        num_frames)).astype(int)
    if not cap.set(cv2.CAP_PROP_POS_FRAMES, int(start_frame)):
        raise RuntimeError(f'Unable to seek to frame {start_frame}.')
    frames = []
    target_i = 0
    # Repeated target frame indices are supported for unusually low-FPS videos.
    for frame_i in range(start_frame, int(target_frames[-1]) + 1):
        ok, bgr = cap.read()
        if not ok:
            break
        while target_i < num_frames and target_frames[target_i] == frame_i:
            rgb = cv2.cvtColor(bgr, cv2.COLOR_BGR2RGB)
            frames.append(letterbox_rgb(rgb, image_size))
            target_i += 1
        if target_i == num_frames:
            break
    if len(frames) != num_frames:
        raise RuntimeError(f'Only decoded {len(frames)}/{num_frames} frames at {start_s:.2f}s.')
    return np.stack(frames, axis=0)
```

Cell 8

```
# Extract and optionally cache the clips. A signature prevents reusing a cache
# accidentally when the source file or extraction parameters change.
def extract_video_clips(video_path, info, clip_table, num_frames, image_size,
                        clip_seconds, output_dir, cache=True):
    video_path = Path(video_path)
    signature = dict(file=str(video_path.resolve()), bytes=video_path.stat().st_size,
                     mtime_ns=video_path.stat().st_mtime_ns,
                     starts=clip_table['start_s'].round(5).tolist(),
                     fps=info['fps'], num_frames=num_frames,
                     image_size=image_size, clip_seconds=clip_seconds)
    key = hashlib.sha256(json.dumps(signature, sort_keys=True).encode()).hexdigest()[:12]
    cache_file = Path(output_dir) / f'clips_{key}.npz'
    if cache and cache_file.is_file():
        with np.load(cache_file, allow_pickle=False) as stored:
            clips = stored['clips']
        print('Loaded cached clips:', cache_file)
        return clips
    cap = cv2.VideoCapture(str(video_path))
    if not cap.isOpened():
        raise RuntimeError(f'Cannot decode video: {video_path}')
    try:
        clips = []
        for i, row in enumerate(clip_table.itertuples(index=False)):
            clips.append(read_clip(cap, info['fps'], row.start_s, clip_seconds,
                                   num_frames, image_size))
            if (i + 1) % 25 == 0:
                print(f'Extracted {i + 1}/{len(clip_table)} clips')
    finally:
        cap.release()
    clips = np.stack(clips).astype(np.uint8, copy=False)
    if cache:
```

```
        np.savez_compressed(cache_file, clips=clips)
        print('Saved clip cache:', cache_file)
    return clips
```

Cell 9

```
# Check the extracted clip array: one sample has multiple RGB frames.
def inspect_clips(clips, clip_table):
    expected = (len(clip_table), NUM_FRAMES, IMAGE_SIZE, IMAGE_SIZE, 3)
    assert clips.shape == expected, f'Expected {expected}, got {clips.shape}'
    print(f'{len(clips)} clips; shape: {clips.shape}; dtype: {clips.dtype}')
    print(f'RGB intensity range: {clips.min()} to {clips.max()}')
    print(f'Raw memory: {clips.nbytes / (1024**2):.1f} MiB')
    assert clip_table['clip_id'].is_unique, 'Duplicate clip IDs!'
    assert (clip_table['end_s'] <= video_info['duration_s']).all(), 'Out-of-range clip!'

# Show representative frames. This is useful for review, but watch the moving
# video segment before assigning a manual label whenever movement is ambiguous.
def show_clip(clips, clip_index, label=None, indices=(0, 5, 10, 15)):
    clip = clips[int(clip_index)]
    select = [min(k, len(clip) - 1) for k in indices]
    fig, axes = plt.subplots(1, len(select), figsize=(12, 3))
    for ax, k in zip(axes, select):
        ax.imshow(clip[k])
        ax.set_title(f'Frame {k + 1}/{len(clip)}', fontsize=10)
        ax.axis('off')
    if label is not None:
        fig.suptitle(str(label))
    plt.tight_layout()
    plt.show()

# Optional animated GIF, useful for observing movement when annotating.
# This animates only the sampled 16 frames and does not contain audio.
def preview_clip_gif(clips, clip_index, out_dir=OUT_DIR):
    from PIL import Image as PILImage
    from IPython.display import Image as NotebookImage
    clip_index = int(clip_index)
    gif_dir = Path(out_dir) / 'preview_gifs'
    gif_dir.mkdir(parents=True, exist_ok=True)
    dest = gif_dir / f'clip_{clip_index:04d}.gif'
    images = [PILImage.fromarray(frame).resize((256, 256)) for frame in clips[clip_index]]
    frame_duration_ms = max(20, round(CLIP_SECONDS * 1000 / len(images)))
    images[0].save(dest, save_all=True, append_images=images[1:],
                   duration=frame_duration_ms, loop=0)
    display(NotebookImage(filename=str(dest)))
    print('Preview:', dest)
    return dest
```

Cell 10

```
# Assign whole clips to three temporal partitions, with unused gaps between them.
# No clip belongs to two partitions, and clips near cutoffs are excluded.
def temporal_split(clip_table, duration_s, gap_seconds):
    if gap_seconds < 0:
        raise ValueError('gap_seconds cannot be negative.')
    first_cut = 0.70 * duration_s
    second_cut = 0.85 * duration_s
    half_gap = gap_seconds / 2.0
    start = clip_table['start_s'].to_numpy()
    end = clip_table['end_s'].to_numpy()
    split = np.full(len(clip_table), 'gap', dtype=object)
```

```
    split[end <= first_cut - half_gap] = 'train'
    split[(start >= first_cut + half_gap) & (end <= second_cut - half_gap)] = 'val'
    split[start >= second_cut + half_gap] = 'test'
    result = clip_table.copy()
    result['split'] = split
    if (result['split'] == 'train').sum() == 0 or (result['split'] == 'val').sum() == 0 or
(result['split'] == 'test').sum() == 0:
        raise ValueError('One temporal split is empty. Use a longer video or reduce
SPLIT_GAP_SECONDS.')
    return result
```

Cell 11

```
# Export an empty manual-annotation CSV, never overwrite any existing labels.
# The 'split' column checks where more annotations are needed.
def ensure_annotation_csv(df, csv_path):
    csv_path = Path(csv_path)
    if not csv_path.exists():
        template = df[['clip_id', 'start_s', 'end_s', 'split']].copy()
        template['label'] = ''
        template.to_csv(csv_path, index=False)
        print('Created annotation template:', csv_path.resolve())
    else:
        print('Using existing annotations (file not overwritten):', csv_path.resolve())
    return csv_path


def load_manual_labels(df, csv_path, class_names):
    csv_path = Path(csv_path)
    if not csv_path.is_file():
        raise FileNotFoundError(f'Annotation CSV not found: {csv_path}')
    annotation = pd.read_csv(csv_path, dtype={'clip_id': 'string', 'label': 'string'})
    if not {'clip_id', 'label'}.issubset(annotation.columns):
        raise ValueError("CSV must contain columns 'clip_id' and 'label'.")
    if annotation['clip_id'].isna().any() or annotation['clip_id'].duplicated().any():
        raise ValueError('CSV contains missing or duplicate clip_id values.')
    unknown = set(annotation['clip_id']) - set(df['clip_id'])
    if unknown:
        raise ValueError(f'CSV has unknown clip IDs: {sorted(unknown)[:5]}')

    # If timestamps/splits are provided (as in our generated template), ensure
    # annotations are still aligned with the current video clipping settings.
    expected = df.set_index('clip_id')
    keyed = annotation.set_index('clip_id')
    for column in ('start_s', 'end_s'):
        if column in annotation:
            recorded = pd.to_numeric(keyed[column], errors='coerce')
            aligned = expected.loc[recorded.index, column]
            if not np.isfinite(recorded.to_numpy(dtype=float)).all() or not np.allclose(
                    recorded.to_numpy(dtype=float), aligned.to_numpy(dtype=float),
                    rtol=0, atol=0.025):
                raise ValueError(f'CSV {column} does not match extracted clips. '
                                 'Rebuild/re-annotate the CSV after changing clip settings.')
    if 'split' in annotation.columns:
        aligned_splits = expected.loc[keyed.index, 'split'].astype(str)
        if not np.array_equal(keyed['split'].astype(str).to_numpy(), aligned_splits.to_numpy()):
            raise ValueError('CSV split column no longer matches the current temporal split.')

    labels = keyed['label'].fillna('').str.strip().str.lower()
    allowed = set(class_names) | {'', 'skip'}
    invalid = set(labels.to_list()) - allowed
```

```
if invalid:
    raise ValueError(f'Invalid CSV labels: {sorted(invalid)}. '
                     f'Allowed: {list(class_names)}, skip, or blank.')
result = df.copy()
result['label'] = result['clip_id'].map(labels).fillna('')
n_labeled = int(result['label'].isin(class_names).sum())
if not n_labeled:
    raise ValueError(f'No manual labels found in {csv_path}. '
                     f'Enter {class_names[0]} or {class_names[1]} in the label column, '
                     'save the CSV, then rerun this cell.')
print(f'Loaded {n_labeled} manually labeled clips from CSV. '
      f'{len(result) - n_labeled} blank/skip/missing clips are excluded.')
return result
```

A1.2 Video recordings — Garmin inspection methods

Source file: TAI_Video_Speed_Estimation.ipynb. Notebook cells 3, 4, 6, 7, 8, 9, 10.

Cell 3

```
# Imports and configuration.
from pathlib import Path
from IPython.display import display, Image
import csv
import json
import math
import subprocess
import sys

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

pd.set_option("display.max_columns", 40)
SEED = 42
ROOT = Path.cwd().resolve()  # Open JupyterLab in the folder containing this notebook.
VIDEO_DIR = ROOT / "dataset" / "videos"
SPEED_DIR = ROOT / "dataset" / "speeds"
UNSEEN_DIR = ROOT / "dataset" / "unseen"
DATA_DIR = ROOT / "data"
PROCESSED = DATA_DIR / "processed_v1"
GROUPS_CSV = DATA_DIR / "drive_groups.csv"
FIXED_TEST_SPLIT = DATA_DIR / "fixed_test_drive.csv"
RUN_DIR = ROOT / "runs" / "stack5_random_val_01"

# Choose the settings before starting a new experiment.
IMAGE_SIZE = 128
MASK_BOTTOM = 0.10
FLOW_CLIP = 32.0
VAL_FRACTION = 0.15
MODEL_MODE = "stack5"
EPOCHS = 40
BATCH_SIZE = 4
LEARNING_RATE = 3e-4
WEIGHT_DECAY = 1e-4
PATIENCE = 8
BALANCE_SPEEDS = False     # Set True in a SEPARATE experiment folder to compare.
DEVICE = "auto"            # Uses CUDA if available, otherwise CPU. Can set 'cuda'.

# Input-data decisions. Verify these against the actual camera/dataset.
RAW_SPEED_DIVISOR = 1.609344  # Camera-specific correction verified for this Garmin dataset.
RUN_METADATA_EXTRACTION = True  # Enable only if matching CSVs are not yet present.

for folder in [DATA_DIR, ROOT / "runs", UNSEEN_DIR]:
    folder.mkdir(parents=True, exist_ok=True)
print("Project:", ROOT)
print("Video data:", VIDEO_DIR)
print("CSV labels:", SPEED_DIR)
print("Python:", sys.executable)
```

Cell 4

```
# This function runs a project script in a separate Python process and shows real errors.
# In particular, it avoids importing PyTorch into the notebook while plotting data.
def run_script(script, *arguments):
```

```
    path = ROOT / script
    if not path.is_file():
        raise FileNotFoundError(f"Required project script is missing: {path}")
    cmd = [sys.executable, str(path)] + [str(a) for a in arguments]
    print("Running:", subprocess.list2cmdline(cmd), flush=True)
    subprocess.run(cmd, cwd=ROOT, check=True)

required = ["run.py", "speednet.py", "train_random_val.py", "export_garmin_labels.py"]
missing = [name for name in required if not (ROOT / name).is_file()]
if missing:
    raise FileNotFoundError(f"Missing helper files next to notebook: {missing}")

# Verify all required video/model libraries in the correct environment.
run_script("run.py", "doctor")
```

Cell 6

```
# Return a table of input video files and flag potentially ambiguous names.
VIDEO_EXTENSIONS = {".mp4", ".mov", ".avi"}

def inventory_videos(folder):
    folder = Path(folder)
    if not folder.is_dir():
        raise FileNotFoundError(f"Video folder does not exist: {folder}")
    paths = sorted(p for p in folder.rglob("*")
                   if p.is_file() and p.suffix.lower() in VIDEO_EXTENSIONS)
    if not paths:
        raise ValueError(f"No videos found inside {folder}")
    table = pd.DataFrame([{
        "video_id": p.stem,
        "filename": p.name,
        "size_mb": round(p.stat().st_size / (1024 * 1024), 2),
        "file_path": str(p),
    } for p in paths])
    if table["video_id"].str.casefold().duplicated().any():
        raise ValueError("Duplicate video stems found. Rename video files uniquely.")
    return table
```

Cell 7

```
# Parse Garmin GPS labels, preserving a genuine zero and excluding missing values.

def parse_video_second(text):
    value = str(text).strip().lower()
    if ":" in value:
        parts = value.split(":")
        if len(parts) not in (2, 3):
            raise ValueError(f"Unsupported timestamp: {text!r}")
        seconds = 0.0
        for part in parts:
            seconds = 60.0 * seconds + float(part)
    else:
        seconds = float(value.removesuffix(" s").removesuffix("s").strip())
    rounded = round(seconds)
    if not math.isfinite(seconds) or rounded < 0 or abs(seconds-rounded) > 0.01:
        raise ValueError(f"Expected an integer-second timestamp, found {text!r}")
    return int(rounded)


def load_garmin_labels(csv_file):
    """Return {second: km/h} for valid speeds; N/A is omitted."""
```

```
    result = {}
    with Path(csv_file).open("r", encoding="utf-8-sig", newline="") as handle:
        for rownum, row in enumerate(csv.reader(handle), 1):
            if not row or all(not c.strip() for c in row):
                continue
            if rownum == 1 and row[0].strip().lower() in {
                "time", "timestamp", "sampletime", "second", "seconds"
            }:
                continue
            if len(row) < 2:
                raise ValueError(f"{csv_file}:{rownum}: Missing speed column")
            second = parse_video_second(row[0])
            raw = row[1].strip()
            if raw.lower() in {"", "n/a", "na", "-", "null", "none"}:
                continue
            speed = float(raw.replace(",", "."))
            if not math.isfinite(speed) or not 0 <= speed <= 400:
                raise ValueError(f"{csv_file}:{rownum}: Invalid speed {speed}")
            if second in result:
                raise ValueError(f"{csv_file}:{rownum}: Duplicate valid second {second}")
            result[second] = speed
    return result
```

Cell 8

```
# Inspect every matching label CSV.
def audit_speed_csvs(video_table, speed_folder):
    rows = []
    speed_folder = Path(speed_folder)
    for item in video_table.itertuples(index=False):
        candidates = list(speed_folder.rglob(f"{item.video_id}_speed.csv"))
        if len(candidates) > 1:
            raise ValueError(f"Several label files match {item.video_id}: {candidates}")
        if not candidates:
            rows.append({"video_id": item.video_id, "status": "NO_CSV",
                         "valid_seconds": 0, "stationary_zeros": 0, "csv": ""})
            continue
        labels = load_garmin_labels(candidates[0])
        rows.append({"video_id": item.video_id,
                     "status": "OK" if labels else "ALL_NA",
                     "valid_seconds": len(labels),
                     "stationary_zeros": sum(v == 0 for v in labels.values()),
                     "csv": str(candidates[0])})
    return pd.DataFrame(rows)
```

Cell 9

```
# Show the ground-truth speed timeline for a single video.
# Missing GPS seconds appear as gaps; valid 0 km/h values remain at zero.
def plot_gps_timeline(csv_file, title=None):
    labels = load_garmin_labels(csv_file)
    if not labels:
        print("No valid GPS speeds in", csv_file)
        return
    timeline = pd.Series(labels, name="speed_kmh", dtype=float)
    timeline = timeline.reindex(range(max(labels)+1))
    plt.figure(figsize=(10, 3.5))
    plt.plot(timeline.index, timeline.values, marker=".", markersize=4)
    plt.xlabel("Video time (s)")
    plt.ylabel("GPS speed (km/h)")
    plt.title(title or Path(csv_file).name)
```

```
    plt.grid(alpha=0.25)
    plt.tight_layout()
    plt.show()
```

Cell 10

```
# Match each MP4 to one real driving session. Do not infer sessions from filenames.
def inspect_drive_groups(groups_csv, inventory):
    mapping = pd.read_csv(groups_csv, dtype=str, keep_default_na=False)
    if not {"video_id", "drive_id"}.issubset(mapping.columns):
        raise ValueError("Expected video_id and drive_id columns.")
    mapping = mapping[["video_id", "drive_id"]].copy()
    mapping = mapping.apply(lambda col: col.str.strip())
    if mapping.duplicated("video_id").any() or (mapping == "").any().any():
        raise ValueError("Duplicated video_id or blank entries in drive_groups.csv")
    expected = set(inventory["video_id"])
    actual = set(mapping["video_id"])
    if expected != actual:
        raise ValueError(f"Video mapping mismatch. Missing: {sorted(expected-actual)}; extra:
{sorted(actual-expected)}")
    return mapping
```

A1.3 Video recordings — Garmin conversion functions

Source file: speednet.py. Selected preparation functions; imports and other dependencies remain in the original script.

```
def timestamp_seconds(value: str) -> int:
    """Handle ExifTool '0 s', '1.00 s', '0:00:30', or numeric seconds."""
    text = str(value).strip().lower()
    if ':' in text:
        parts = text.split(':')
        if not 2 <= len(parts) <= 3:
            raise ValueError(f"Unsupported timestamp: {value!r}")
        seconds = 0.0
        for part in parts:
            seconds = 60.0 * seconds + float(part)
    else:
        text = text.removesuffix(' s').removesuffix('s').strip()
        seconds = float(text)
    rounded = round(seconds)
    if not math.isfinite(seconds) or abs(seconds - rounded) > 0.01 or rounded < 0:
        raise ValueError(f"Expected an integer-second timestamp, got {value!r}")
    return int(rounded)


def read_speed_csv(path: Path) -> dict[int, float]:
    """CSV is already in km/h. No mph conversion is performed."""
    speeds = {}
    with Path(path).open('r', encoding='utf-8-sig', newline='') as f:
        for rowno, row in enumerate(csv.reader(f), start=1):
            if not row or all(not c.strip() for c in row):
                continue
            if len(row) < 2:
                raise ValueError(f"{path}:{rowno}: Expected timestamp,speed.")
            # Optional header row, without silently dropping malformed data rows.
            if rowno == 1 and row[0].strip().lower() in {'time', 'timestamp', 'sampletime',
'second', 'seconds'}:
                continue
            try:
                sec = timestamp_seconds(row[0])
            except ValueError as exc:
                raise ValueError(f"{path}:{rowno}: Cannot parse timestamp {row[0]!r}") from exc
            # A missing speed must remain missing; do not turn it into 0.
            # Its timestamp is left out of 'speeds', and the preprocessing
            # validity mask will exclude it without splicing the video.
            raw_speed = row[1].strip()
            if raw_speed.lower() in {'', 'n/a', 'na', '-', 'null', 'none'}:
                continue
            try:
                speed = float(raw_speed.replace(',', '.'))
            except ValueError as exc:
                raise ValueError(f"{path}:{rowno}: Cannot parse speed {raw_speed!r}") from exc
            if not math.isfinite(speed) or not (0 <= speed <= 400):
                raise ValueError(f"{path}:{rowno}: Implausible speed: {speed}")
            if sec in speeds:
                raise ValueError(f"{path}:{rowno}: Duplicate second {sec}")
            speeds[sec] = speed
    # A CSV containing only N/A values is not malformed. Such videos must
    # be skipped during preparation rather than stopping the entire batch.
    return speeds


def preprocess_config(image_size: int, flow_clip: float, mask_bottom: float) -> dict:
```

```
    if image_size not in (64, 96, 128, 192, 256):
        raise ValueError('image_size must be one of 64, 96, 128, 192, 256')
    if not 0 <= mask_bottom < 0.5:
        raise ValueError('mask_bottom must be >=0 and <0.5')
    if flow_clip <= 0:
        raise ValueError('flow_clip must be positive')
    return {'version': FORMAT_VERSION, 'fps': FPS, 'seconds': SECONDS,
            'image_size': image_size, 'flow_clip': float(flow_clip),
            'mask_bottom': float(mask_bottom)}


def decode_video_frames(video_path: Path, config: dict) -> np.ndarray:
    """FFmpeg timestamps are resampled at 5 FPS, then centre-cropped.

    The returned uint8 frames have shape [F, H, W]. No GPS is read.
    """
    sz = int(config['image_size'])
    filt = ("fps=5:round=near,"
            "crop='min(iw,ih)':'min(iw,ih)',"
            f"scale={sz}:{sz}:flags=area,format=gray")
    cmd = ['ffmpeg', '-hide_banner', '-loglevel', 'error', '-nostdin',
           '-i', str(video_path), '-map', '0:v:0', '-vf', filt,
           '-an', '-sn', '-dn', '-fps_mode', 'passthrough',
           '-f', 'rawvideo', '-pix_fmt', 'gray', 'pipe:1']
    try:
        result = subprocess.run(cmd, capture_output=True, check=False)
    except FileNotFoundError as exc:
        raise RuntimeError('FFmpeg not found. Install the Conda environment.') from exc
    if result.returncode:
        raise RuntimeError(f"FFmpeg failed for {video_path}:
{result.stderr.decode(errors='replace')[:700]}")
    frame_bytes = sz * sz
    if len(result.stdout) % frame_bytes:
        raise RuntimeError(f"Truncated FFmpeg frames for {video_path}")
    frames = np.frombuffer(result.stdout, dtype=np.uint8).reshape(-1, sz, sz).copy()
    if config['mask_bottom'] > 0:
        cut = int(round(sz * (1.0 - config['mask_bottom'])))
        frames[:, cut:, :] = 0
    return frames


def extract_flow_window(frames: np.ndarray, start_second: int, config: dict) -> np.ndarray:
    """Extract optical flow for exactly 10 seconds, starting on a whole second."""
    if frames.ndim != 3:
        raise ValueError(f"Expected [frame,height,width], got {frames.shape}")
    begin = int(start_second) * FPS
    if begin < 0 or begin + FLOW_FRAMES >= len(frames):
        raise ValueError('Video does not have the 51 frames required for this clip.')
    clip = []
    for idx in range(begin, begin + FLOW_FRAMES):
        flow = cv2.calcOpticalFlowFarneback(
            frames[idx], frames[idx + 1], None,
            pyr_scale=0.5, levels=3, winsize=15, iterations=3,
            poly_n=5, poly_sigma=1.2, flags=0)
        clip.append(np.moveaxis(np.clip(flow, -config['flow_clip'], config['flow_clip']), -1,
0))
    return np.stack(clip).astype(np.float16)


def inference_window_starts(frames: np.ndarray) -> list[int]:
    """Aligned 10-second windows plus an overlapping tail for training/inference.

    A window needs 51 source frames to calculate 50 optical-flow maps. For a
```

```
    nominal 60-second video decoded to 300 frames at 5 fps, windows are
    [0, 10, 20, 30, 40, 49], covering speed seconds 0 through 58. The
    final overlapping second is supervised only once in training.
    """
    full_seconds = (len(frames) - 1) // FPS
    if full_seconds < SECONDS:
        return []
    starts = list(range(0, full_seconds - SECONDS + 1, SECONDS))
    tail_start = full_seconds - SECONDS
    if tail_start not in starts:
        starts.append(tail_start)
    return starts


def window_speed_targets(labels: dict[int, float], start_second: int,
                         zero_label_policy: str = 'valid',
                         already_supervised: set[int] | None = None):
    """Return 10 speed targets and a boolean supervision mask in REAL time.

    Missing timestamp rows never count. With zero_label_policy='valid',
    genuine numeric zero speeds ARE supervised. Legacy 'missing' treats all
    zeros as unknown and should be used only for older unverified CSVs. Never splice
    separated parts of video together to close an invalid-label gap.
    """
    if zero_label_policy not in ('missing', 'valid'):
        raise ValueError('zero_label_policy must be missing or valid')
    already_supervised = already_supervised if already_supervised is not None else set()
    values = np.zeros(SECONDS, dtype=np.float32)
    mask = np.zeros(SECONDS, dtype=bool)
    for offset in range(SECONDS):
        sec = start_second + offset
        speed = labels.get(sec)
        if speed is None or sec in already_supervised:
            continue
        if speed < 0 or not math.isfinite(speed):
            raise ValueError(f'Invalid label at second {sec}: {speed}')
        if speed == 0 and zero_label_policy == 'missing':
            continue
        values[offset] = speed
        mask[offset] = True
    return values, mask
```

A1.4 Video recordings — Garmin preparation procedure

Source file: run.py. Selected preparation functions; imports and other dependencies remain in the original script.

```
def prepare(args):
    """Keep chronological video windows; mask invalid GPS labels per second."""
    output = Path(args.out).resolve()
    flow_dir = output / 'flows'
    flow_dir.mkdir(parents=True, exist_ok=True)
    if not 1 <= args.min_valid_seconds <= SECONDS:
        raise ValueError('--min-valid-seconds must be between 1 and 10')
    config = preprocess_config(args.image_size, args.flow_clip, args.mask_bottom)
    config['zero_label_policy'] = args.zero_label_policy
    config['min_valid_seconds'] = args.min_valid_seconds
    config_file = output / 'preprocessing.json'
    if config_file.exists():
        import json
        old = json.loads(config_file.read_text(encoding='utf-8'))
        if old != config:
            raise RuntimeError('Preprocessing or label settings changed. Use a NEW --out folder.
'
                               'Cached clips must not mix label policies or preprocessing
settings.')
    else:
        as_json(config_file, config)

    videos = sorted(p for p in Path(args.video_dir).rglob('*')
                    if p.is_file() and p.suffix.lower() in {'.mp4', '.mov', '.avi'})
    if not videos:
        raise RuntimeError(f'No videos found in {args.video_dir}')
    if args.max_videos is not None:
        videos = videos[:args.max_videos]
    csv_files = [p for p in Path(args.speed_dir).rglob('*')
                 if p.is_file() and p.suffix.lower() == '.csv']
    csv_map = {}
    for p in csv_files:
        stem = p.stem
        if stem.lower().endswith('_speed'):
            stem = stem[:-6]
        key = stem.casefold()
        if key in csv_map:
            raise ValueError(f'Multiple CSVs match {stem!r}: {csv_map[key]} and {p}')
        csv_map[key] = p
    groups = load_groups(Path(args.groups) if args.groups else None)
    stems = [v.stem.casefold() for v in videos]
    if len(set(stems)) != len(stems):
        raise RuntimeError('Two videos have the same base filename. Rename them to unique IDs.')

    manifest = []
    counts = Counter()
    skipped_empty = []
    for video in tqdm(videos, unit='video', desc='Farneback preprocessing'):
        video_id = video.stem
        csv_file = csv_map.get(video_id.casefold())
        if csv_file is None:
            print(f'No matching CSV, skipping: {video.name}')
            counts['no_csv'] += 1
            continue
        if args.groups and video_id not in groups:
            raise ValueError(f'No drive_id for {video_id} in {args.groups}; include all labelled
```

```
videos.')
        try:
            labels = read_speed_csv(csv_file)
            if not labels:
                print(f'All GPS speeds are N/A; skipping supervised training for: {video.name}')
                skipped_empty.append({'video_id': video_id, 'csv_file': str(csv_file.resolve()),
                                      'reason': 'all labels missing or N/A'})
                counts['all_na_videos'] += 1
                continue
            frames = decode_video_frames(video, config)
            starts = inference_window_starts(frames)
            hash_csv = sha256_file(csv_file)
            found = 0
            used_seconds = set()
            for start in starts:
                speeds, valid_mask = window_speed_targets(
                    labels, start, args.zero_label_policy, used_seconds)
                nvalid = int(valid_mask.sum())
                if nvalid < args.min_valid_seconds:
                    counts['windows_with_few_valid_labels'] += 1
                    continue
                target_name = f'{video_id}_start_{start:04d}.npz'
                target = flow_dir / target_name
                flow = None
                needs_save = True
                if target.is_file() and target.stat().st_mtime_ns >= video.stat().st_mtime_ns:
                    # Optical flow can be reused after a CSV label change.
                    with np.load(target, allow_pickle=False) as cached:
                        if (cached['flow'].shape == (FLOW_FRAMES, 2, config['image_size'],
config['image_size'])
                                and 'valid_mask' in cached):
                            flow = np.asarray(cached['flow'], dtype=np.float16)
                            needs_save = not (np.array_equal(cached['speed'], speeds)
                                              and np.array_equal(cached['valid_mask'],
valid_mask))
                if flow is None:
                    flow = extract_flow_window(frames, start, config)
                    needs_save = True
                if needs_save:
                    temp = target.with_name(target.name + '.tmp.npz')
                    np.savez_compressed(temp, flow=flow, speed=speeds,
                                        valid_mask=valid_mask.astype(np.uint8))
                    temp.replace(target)
                manifest.append({
                    'clip_file': f'flows/{target_name}', 'video_id': video_id,
                    'drive_id': groups.get(video_id, video_id),
                    'start_second': start, 'mean_speed_kmh': f'{speeds[valid_mask].mean():.5f}',
                    'valid_seconds': nvalid, 'invalid_seconds': SECONDS - nvalid,
                    'zero_label_policy': args.zero_label_policy,
                    'source_video': str(video.resolve()), 'source_csv': str(csv_file.resolve()),
                    'csv_sha256': hash_csv,
                })
                used_seconds.update(start + i for i in np.flatnonzero(valid_mask))
                found += 1
                counts['valid_seconds'] += nvalid
            counts['videos_with_clips'] += (found > 0)
            counts['clips'] += found
            if not found:
                print(f'No 10s window with enough valid speed labels: {video.name}')
        except Exception as e:
            if args.skip_errors:
                print(f'WARNING: Skipping {video.name}: {e}')
```

```
                counts['errors'] += 1
                continue
            raise RuntimeError(f'Preprocessing failed for {video}: {e}') from e
    if skipped_empty:
        write_csv(output / 'skipped_all_na.csv', skipped_empty, ['video_id', 'csv_file',
'reason'])
    if not manifest:
        raise RuntimeError('No usable clips produced. Check CSVs for N/A labels, filenames and
video duration.')
    manifest.sort(key=lambda x: (x['video_id'], x['start_second']))
    write_csv(output / 'manifest.csv', manifest, MANIFEST_COLUMNS)
    print(f'Wrote {len(manifest)} clips, {counts["valid_seconds"]} uniquely labelled seconds '
          f'from {counts["videos_with_clips"]} videos.')
    print(f'Manifest: {output / "manifest.csv"} '
          f'(groups: {len(set(r["drive_id"] for r in manifest))})')
    print(f'Skips: no_csv={counts["no_csv"]}; all_NA={counts["all_na_videos"]};
sparse_windows={counts["windows_with_few_valid_labels"]}; '
          f'errors={counts["errors"]}')
```

A1.5 Still images — CIFAR-10 methods

Source file: TAI_cifar10_classification.ipynb. Notebook cells 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19.

Cell 3

```
# Imports and basic settings. Nothing is computed here.
import warnings
warnings.filterwarnings('ignore')  # first, so that warnings from the imports below are hidden
too
import os, itertools, hashlib, time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import joblib
from PIL import Image
from IPython.display import display, clear_output
import ipywidgets as widgets
from ipywidgets import interact

import logging
logging.getLogger('tensorflow').setLevel(logging.ERROR)
# silence TensorFlow messages (must be set before tensorflow is imported)
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'
os.environ['KMP_DUPLICATE_LIB_OK'] = 'TRUE'  # avoids an OpenMP conflict on macOS
pd.set_option('display.width', 140)
pd.set_option('display.max_columns', 30)

SEED = 42
DATA_DIR = 'data/cifar10'
OUT_DIR = 'output_cifar10'
MODEL_DIR = os.path.join(OUT_DIR, 'models')
os.makedirs(MODEL_DIR, exist_ok=True)

IMG_SIZE = 32
TRAIN_PER_CLASS = 1500  # 5000 available; fewer images = faster training
TEST_PER_CLASS = 1000
```

Cell 5

```
# Load images from a folder with one subfolder per class. Files that cannot be opened are
skipped and reported.
def load_images(folder, per_class=None, size=IMG_SIZE):
    class_names = sorted(os.listdir(folder))
    clean_names = []
    for name in class_names:
        if os.path.isdir(os.path.join(folder, name)):
            clean_names.append(name)
    class_names = clean_names

    rng = np.random.default_rng(SEED)
    images, labels, files, broken = [], [], [], []
    resized = 0
    for label in range(len(class_names)):
        class_folder = os.path.join(folder, class_names[label])
        names = sorted(os.listdir(class_folder))
        # A random sample, because the files are sorted by sub-type and the first ones are not
representative.
        if per_class is not None and per_class < len(names):
            chosen = rng.choice(len(names), size=per_class, replace=False)
```

```
            chosen = sorted(chosen)
            subset = []
            for i in chosen:
                subset.append(names[i])
            names = subset
        for name in names:
            path = os.path.join(class_folder, name)
            try:
                im = Image.open(path)
                im = im.convert('RGB')
            except Exception:
                broken.append(path)
                continue
            if im.size != (size, size):
                im = im.resize((size, size))
                resized = resized + 1
            images.append(np.array(im))
            labels.append(label)
            files.append(path)

    X = np.array(images, dtype='uint8')
    y = np.array(labels)
    print(f"Loaded '{folder}': {len(X)} images in {len(class_names)} classes.")
    if resized > 0:
        print(f"{resized} image(s) had another size and were resized to {size} x {size}.")
    if broken:
        print(f"{len(broken)} file(s) could not be opened and were skipped (see
check_corrupt).")
    return X, y, class_names, files, broken
```

Cell 6

```
# Print a short overview: array shape, pixel range and the number of images per class.
def overview(X, y, class_names, title='data'):
    print(f"Images: {X.shape[0]}, height: {X.shape[1]}, width: {X.shape[2]}, channels:
{X.shape[3]}")
    print(f"Pixel type: {X.dtype}, smallest value: {X.min()}, largest value: {X.max()}")
    counts = []
    for label in range(len(class_names)):
        counts.append(int(np.sum(y == label)))
    table = pd.DataFrame({'class': class_names, 'images': counts}).set_index('class')
    display(table.T)
    plt.figure(figsize=(9, 3))
    plt.bar(class_names, counts, color='tab:blue')
    plt.title(f'Images per class ({title})')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()
```

Cell 7

```
# Show a grid of example images: one row per class, n images per row.
def show_grid(X, y, class_names, n=8):
    rng = np.random.default_rng(SEED)
    fig, ax = plt.subplots(len(class_names), n, figsize=(n * 1.1, len(class_names) * 1.15))
    for label in range(len(class_names)):
        idx = np.where(y == label)[0]
        pick = rng.choice(idx, size=min(n, len(idx)), replace=False)
        for j in range(n):
            ax[label, j].axis('off')
            if j < len(pick):
                ax[label, j].imshow(X[pick[j]])
        ax[label, 0].set_title(class_names[label], fontsize=8, loc='left')
    plt.tight_layout()
```

```
    plt.show()
```

Cell 8

```
# Show one image and the R,G,B values of a small square in its top-left corner.
def show_pixels(img, size=6):
    fig, ax = plt.subplots(1, 2, figsize=(7, 3.2))
    ax[0].imshow(img)
    ax[0].add_patch(plt.Rectangle((-0.5, -0.5), size, size, fill=False, edgecolor='yellow',
linewidth=2))
    ax[0].set_title('Image (yellow: the square below)')
    ax[1].imshow(img[:size, :size])
    ax[1].set_title(f'The square, enlarged ({size} x {size} pixels)')
    for a in ax:
        a.axis('off')
    plt.tight_layout()
    plt.show()
    rows = []
    for r in range(size):
        row = []
        for c in range(size):
            p = img[r, c]
            row.append(f"{p[0]},{p[1]},{p[2]}")
        rows.append(row)
    print(f"Pixel values (R,G,B) of the square. The whole image has shape {img.shape}.")
    display(pd.DataFrame(rows))
```

Cell 9

```
# Convert an RGB image to gray, HSV or one channel, and show all of them next to each other.
from skimage.color import rgb2gray, rgb2hsv

def convert_color(img, mode):
    if mode == 'gray':
        return rgb2gray(img)
    if mode == 'hsv':
        return rgb2hsv(img)
    if mode == 'R':
        return img[:, :, 0]
    if mode == 'G':
        return img[:, :, 1]
    if mode == 'B':
        return img[:, :, 2]
    raise ValueError(mode)

def show_color_spaces(img):
    hsv = convert_color(img, 'hsv')
    panels = [('original (RGB)', img, None),
              ('R channel', convert_color(img, 'R'), 'Reds'),
              ('G channel', convert_color(img, 'G'), 'Greens'),
              ('B channel', convert_color(img, 'B'), 'Blues'),
              ('gray', convert_color(img, 'gray'), 'gray'),
              ('H (hue)', hsv[:, :, 0], 'hsv'),
              ('S (saturation)', hsv[:, :, 1], 'gray'),
              ('V (value)', hsv[:, :, 2], 'gray')]
    fig, ax = plt.subplots(1, len(panels), figsize=(2 * len(panels), 2.4))
    for i in range(len(panels)):
        title, data, cmap = panels[i]
        ax[i].imshow(data, cmap=cmap)
        ax[i].set_title(title, fontsize=9)
        ax[i].axis('off')
    plt.tight_layout()
    plt.show()
    print("Gray = 0.2125 R + 0.7154 G + 0.0721 B. The gray image has 1 channel instead of 3, so
```

```
it has 3 times fewer numbers.")
```

Cell 10

```
# Histograms of one image, and the average histograms of two classes.
def plot_histogram(img):
    fig, ax = plt.subplots(1, 3, figsize=(13, 3))
    ax[0].imshow(img)
    ax[0].axis('off')
    ax[0].set_title('image')
    gray = (convert_color(img, 'gray') * 255).ravel()
    ax[1].hist(gray, bins=32, range=(0, 255), color='gray')
    ax[1].set_title('gray histogram')
    names = ['R', 'G', 'B']
    colors = ['red', 'green', 'blue']
    for i in range(3):
        ax[2].hist(img[:, :, i].ravel(), bins=32, range=(0, 255), color=colors[i], alpha=0.4,
label=names[i])
    ax[2].set_title('histogram per channel')
    ax[2].legend()
    for a in ax[1:]:
        a.set_xlabel('pixel value (0 = dark, 255 = bright)')
        a.set_ylabel('number of pixels')
    plt.tight_layout()
    plt.show()

def compare_class_histograms(X, y, class_names, class_a, class_b):
    fig, ax = plt.subplots(1, 2, figsize=(12, 3), sharey=True)
    pair = [class_a, class_b]
    colors = ['red', 'green', 'blue']
    for k in range(2):
        label = class_names.index(pair[k])
        images = X[y == label]
        for i in range(3):
            counts, edges = np.histogram(images[:, :, :, i], bins=32, range=(0, 255))
            counts = counts / len(images)
            ax[k].plot(edges[:-1], counts, color=colors[i], label='RGB'[i])
        ax[k].set_title(f"average histogram: {pair[k]} ({len(images)} images)")
        ax[k].set_xlabel('pixel value')
        ax[k].legend()
    plt.tight_layout()
    plt.show()
```

Cell 11

```
# Report the files that load_images could not open. They were already skipped.
def check_corrupt(broken):
    if not broken:
        print("No corrupted files found.")
    else:
        print(f"{len(broken)} file(s) could not be opened and were skipped:")
        for path in broken:
            print("  " + path)
    return len(broken)
```

Cell 12

```
# Find duplicated images and test images that are also training images, and remove them.
def image_hash(img):
    return hashlib.md5(img.tobytes()).hexdigest()

def check_duplicates(X):
    seen = {}
    copies = []
    for i in range(len(X)):
```

```
        h = image_hash(X[i])
        if h in seen:
            copies.append(i)
        else:
            seen[h] = i
    if not copies:
        print("No duplicated images found.")
    else:
        print(f"Found {len(copies)} duplicated image(s) ({100 * len(copies) / len(X):.2f}% of
the data).")
    return copies

def check_overlap(X_first, X_second):
    first = set()
    for i in range(len(X_first)):
        first.add(image_hash(X_first[i]))
    same = []
    for i in range(len(X_second)):
        if image_hash(X_second[i]) in first:
            same.append(i)
    if not same:
        print("No image of the second set appears in the first set.")
    else:
        print(f"{len(same)} image(s) of the second set also appear in the first set.")
    return same

def remove_images(X, y, files, positions, reason):
    if not positions:
        print(f"No {reason} to remove, nothing to do.")
        return X, y, files
    keep = np.ones(len(X), dtype=bool)
    keep[positions] = False
    new_files = []
    for i in range(len(files)):
        if keep[i]:
            new_files.append(files[i])
    print(f"Removed {len(positions)} {reason}. {int(keep.sum())} images remain.")
    return X[keep], y[keep], new_files
```

Cell 13

```
# Find and show unusual images: much darker, brighter or with lower contrast than the average
image.
# Brightness is the mean pixel value and contrast the standard deviation of the pixel values.
Nothing is removed here.
def check_outliers(X, threshold=3):
    flat = X.reshape(len(X), -1).astype('float32')
    brightness = flat.mean(axis=1)
    contrast = flat.std(axis=1)
    z_b = (brightness - brightness.mean()) / brightness.std()
    z_c = (contrast - contrast.mean()) / contrast.std()
    found = {'too dark': np.where(z_b < -threshold)[0],
             'too bright': np.where(z_b > threshold)[0],
             'low contrast': np.where(z_c < -threshold)[0]}
    order = {'too dark': z_b, 'too bright': -z_b, 'low contrast': z_c}
    rows = []
    outliers = {}
    for kind in found:
        idx = found[kind]
        if len(idx) > 0:
            idx = idx[np.argsort(order[kind][idx])]
            outliers[kind] = idx
            rows.append({'kind': kind, 'images': len(idx), 'percent': round(100 * len(idx) /
```

```
len(X), 2)})
            print(f"Warning: {len(idx)} image(s) are '{kind}' (more than {threshold} standard
deviations from the average).")
    if not outliers:
        print(f"No image is more than {threshold} standard deviations from the average.")
    else:
        print("Nothing is removed yet. Use show_outliers() to look at the images and
fix_outliers() to remove them.")
        display(pd.DataFrame(rows).set_index('kind'))
    return outliers

def show_outliers(X, y, class_names, outliers, n=8):
    if not outliers:
        print("No outliers to show.")
        return
    for kind in outliers:
        idx = outliers[kind][:n]
        fig, ax = plt.subplots(1, n, figsize=(n * 1.3, 1.7))
        for j in range(n):
            ax[j].axis('off')
            if j < len(idx):
                ax[j].imshow(X[idx[j]])
                ax[j].set_title(class_names[y[idx[j]]], fontsize=8)
        fig.suptitle(f"'{kind}': {len(outliers[kind])} image(s), showing the {len(idx)} most
extreme", fontsize=10)
        plt.tight_layout()
        plt.show()
```

Cell 14

```
# Widget for choosing the kinds of unusual images to remove (nothing selected by default), and
the removal.
def select_outliers(outliers):
    kinds = list(outliers.keys())
    w = widgets.SelectMultiple(options=kinds, value=[], description='Remove',
                               rows=max(2, len(kinds)), layout=widgets.Layout(width='40%'))
    if kinds:
        print("Select the kinds of images that should be removed (Ctrl/Cmd+click for several).
Leave empty to keep everything.")
    else:
        print("No outliers were found, nothing to select.")
    display(w)
    return w

def fix_outliers(X, y, files, outliers, kinds):
    positions = set()
    for kind in kinds:
        if kind in outliers:
            for i in outliers[kind]:
                positions.add(int(i))
    if not kinds:
        print("No kind selected, no images removed.")
        return X, y, files
    return remove_images(X, y, files, sorted(positions), 'unusual image(s)')
```

Cell 15

```
# Widget for choosing the classes that take part, and keeping only those classes (renumbered 0,
1, 2, ...).
def select_classes(class_names):
    w = widgets.SelectMultiple(options=class_names, value=tuple(class_names),
description='Classes',
                               rows=len(class_names), layout=widgets.Layout(width='40%'))
    print("Select the classes that take part (Ctrl/Cmd+click for several).")
```

```
    display(w)
    return w

def keep_classes(X, y, files, class_names, chosen):
    chosen = list(chosen)
    if len(chosen) < 2:
        raise ValueError("Choose at least two classes.")
    new_names = []
    for name in class_names:
        if name in chosen:
            new_names.append(name)
    keep = np.zeros(len(y), dtype=bool)
    new_y = np.zeros(len(y), dtype=int)
    for new_label in range(len(new_names)):
        old_label = class_names.index(new_names[new_label])
        keep[y == old_label] = True
        new_y[y == old_label] = new_label
    new_files = []
    for i in range(len(files)):
        if keep[i]:
            new_files.append(files[i])
    print(f"Classes ({len(new_names)}): {new_names}. {int(keep.sum())} images.")
    return X[keep], new_y[keep], new_files, new_names
```

Cell 16

```
# Show how class names become numbers (labels) and one-hot vectors.
def show_label_encoding(class_names):
    rows = []
    for label in range(len(class_names)):
        one_hot = [0] * len(class_names)
        one_hot[label] = 1
        rows.append({'class': class_names[label], 'label': label, 'one-hot vector':
str(one_hot)})
    display(pd.DataFrame(rows).set_index('class'))
    print("Unlike the dummy columns in the tabular notebooks (k-1 columns), here all k positions
are used,")
    print("because the network outputs one probability for every class.")
```

Cell 17

```
# Stratified split of the training images into training and validation sets. The test set stays
separate.
from sklearn.model_selection import train_test_split

def split_data(X, y, val_size=0.15):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=val_size, stratify=y,
random_state=SEED)
    print(f"Stratified split: {len(X_train)} training images, {len(X_val)} validation images.")
    return X_train, X_val, y_train, y_val
```

Cell 18

```
# Scale pixel values: '0-1' divides by 255, 'standard' uses the mean and deviation of every
channel of the training set.
def fit_scaler(X_train, method='0-1'):
    scaler = {'method': method}
    if method == 'standard':
        pixels = X_train.reshape(-1, 3).astype('float32') / 255.0
        scaler['mean'] = pixels.mean(axis=0)
        scaler['std'] = pixels.std(axis=0)
    elif method != '0-1':
        raise ValueError(method)
    return scaler
```

```
def apply_scaler(X, scaler):
    X = X.astype('float32') / 255.0
    if scaler['method'] == 'standard':
        X = (X - scaler['mean']) / scaler['std']
    return X
```

Cell 19

```
# Data augmentation: small random changes of the training images, used only during training, and
a preview with sliders.
AUGMENTATION_LEVELS = {
    'none': None,
    'light': {'flip': True, 'rotation': 0.03, 'zoom': 0.0, 'shift': 0.1, 'contrast': 0.0},
    'strong': {'flip': True, 'rotation': 0.08, 'zoom': 0.15, 'shift': 0.12, 'contrast': 0.2},
}

def make_augmenter(settings):
    if settings is None:
        return None
    from tensorflow import keras
    layers = []
    if settings['flip']:
        layers.append(keras.layers.RandomFlip('horizontal', seed=SEED))
    if settings['rotation'] > 0:
        layers.append(keras.layers.RandomRotation(settings['rotation'], seed=SEED))
    if settings['zoom'] > 0:
        layers.append(keras.layers.RandomZoom(settings['zoom'], seed=SEED))
    if settings['shift'] > 0:
        layers.append(keras.layers.RandomTranslation(settings['shift'], settings['shift'],
seed=SEED))
    if settings['contrast'] > 0:
        layers.append(keras.layers.RandomContrast(settings['contrast'], seed=SEED))
    if not layers:
        return None
    return keras.Sequential(layers, name='augmentation')

def preview_augmentation(img):
    flip = widgets.Checkbox(value=True, description='mirror (flip)')
    rotation = widgets.FloatSlider(value=0.08, min=0, max=0.5, step=0.01,
description='rotation')
    zoom = widgets.FloatSlider(value=0.15, min=0, max=0.5, step=0.05, description='zoom')
    shift = widgets.FloatSlider(value=0.12, min=0, max=0.4, step=0.02, description='shift')
    contrast = widgets.FloatSlider(value=0.2, min=0, max=0.8, step=0.05, description='contrast')
    def update(flip, rotation, zoom, shift, contrast):
        settings = {'flip': flip, 'rotation': rotation, 'zoom': zoom, 'shift': shift,
'contrast': contrast}
        aug = make_augmenter(settings)
        fig, ax = plt.subplots(1, 9, figsize=(15, 1.9))
        ax[0].imshow(img)
        ax[0].set_title('original', fontsize=9)
        batch = np.repeat(img[np.newaxis].astype('float32'), 8, axis=0)
        if aug is not None:
            batch = aug(batch, training=True).numpy()
        for j in range(8):
            ax[j + 1].imshow(np.clip(batch[j], 0, 255).astype('uint8'))
            ax[j + 1].set_title(f'version {j + 1}', fontsize=9)
        for a in ax:
            a.axis('off')
        plt.tight_layout()
        plt.show()
    interact(update, flip=flip, rotation=rotation, zoom=zoom, shift=shift, contrast=contrast)
```

A1.6 Audio/speech inputs — FSDD methods

Source file: TAI_fsdd_audio_classification.ipynb. Notebook cells 2, 4, 5, 6.

Cell 2

```
# Imports and shared configuration. Run after a kernel restart.
import os
os.environ.setdefault('TF_CPP_MIN_LOG_LEVEL', '2')
from pathlib import Path
import io, json, time, hashlib, uuid, copy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import soundfile as sf
import librosa
import librosa.display
import joblib
import ipywidgets as widgets
from IPython.display import display, Audio, clear_output
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score, f1_score,
                             log_loss, confusion_matrix, ConfusionMatrixDisplay,
classification_report)
from tqdm.auto import tqdm

SEED = 42
RECORDINGS_DIR = Path('recordings')
OUT_DIR = Path('output_fsdd')
OUT_DIR.mkdir(exist_ok=True)

CFG = dict(sample_rate=8000, n_fft=256, hop_length=128, n_mels=40, n_mfcc=20,
           max_seconds=2.5, normalize_peak=True, trim_silence=False, trim_top_db=30)
SPLIT_MODE = 'official'       # 'official' or 'speaker_independent'
TEST_SPEAKER = 'george'       # used only in speaker_independent mode
VAL_SPEAKER = 'jackson'
USE_GPU = False               # set True before TensorFlow is initialized to use a configured GPU
MAX_EPOCHS = 20
BATCH_SIZE = 64
RUN_BASELINES = True          # one configuration per method
RUN_GRID = True               # train the expanded grid; completed matching fits are reused
METHODS = ['KNN', 'SVM', 'Random forest', 'MLP', 'CNN', 'LSTM', 'GRU']
CLASSES = np.arange(10)
print('Working folder:', Path.cwd())
print('Output folder:', OUT_DIR.resolve())
```

Cell 4

```
def find_recordings(folder):
    folder = Path(folder)
    if folder.is_dir() and any(folder.glob('*.wav')):
        return folder
    candidates = [p for p in Path('.').glob('**/recordings') if any(p.glob('*.wav'))]
    if len(candidates) == 1:
        return candidates[0]
    raise FileNotFoundError('Set RECORDINGS_DIR to the folder containing the FSDD WAV files.')
```

```
def index_audio(folder):
    rows, errors = [], []
    for path in sorted(Path(folder).glob('*.wav')):
        try:
            digit, speaker, repeat = path.stem.split('_')
            digit, repeat = int(digit), int(repeat)
            if digit not in range(10) or repeat < 0:
                raise ValueError('Invalid digit or repetition index.')
            info = sf.info(path)
            values, sr = sf.read(path, dtype='float32', always_2d=True)
            if values.size == 0 or not np.isfinite(values).all():
                raise ValueError('Empty or non-finite signal.')
            rows.append(dict(file=path.name, path=str(path.resolve()), digit=digit,
speaker=speaker,
                             repeat=repeat, sample_rate=sr, channels=info.channels,
                             subtype=info.subtype, frames=len(values), seconds=len(values)/sr,
                             peak=float(np.max(np.abs(values))),
rms=float(np.sqrt(np.mean(values**2))),
                             sha256=hashlib.sha256(np.asarray(values).tobytes()).hexdigest()))
        except Exception as exc:
            errors.append({'file': path.name, 'error': str(exc)})
    if errors:
        display(pd.DataFrame(errors))
        raise ValueError('Fix the listed files before continuing; nothing was silently
omitted.')
    if not rows:
        raise ValueError('No valid WAV files found.')
    return pd.DataFrame(rows)

def overview_audio(index):
    print(f'{len(index)} recordings, {index.speaker.nunique()} speakers, {index.digit.nunique()}
classes.')
    display(index[['file', 'digit', 'speaker', 'seconds', 'sample_rate', 'channels']].head())
    display(pd.crosstab(index.speaker, index.digit))
    display(index[['seconds', 'rms', 'peak']].describe().round(4))
    print('Formats:')
    display(index.groupby(['sample_rate', 'channels',
'subtype']).size().rename('files').to_frame())
    print('Silent recordings:', int((index.peak == 0).sum()))
    print('Duplicate decoded signals:', int(index.sha256.duplicated().sum()))
    fig, ax = plt.subplots(1, 2, figsize=(11, 3))
    index.seconds.hist(bins=35, ax=ax[0]); ax[0].set(xlabel='Seconds', ylabel='Recordings',
title='Duration')
    index.groupby('digit').size().plot.bar(ax=ax[1]); ax[1].set(xlabel='Digit',
ylabel='Recordings', title='Class balance')
    plt.tight_layout(); plt.show()
```

Cell 5

```
def load_audio(source, config):
    # soundfile accepts a path or BytesIO. Stereo uploads are averaged to mono.
    data, sr = sf.read(source, dtype='float32', always_2d=True)
    if data.size == 0 or not np.isfinite(data).all():
        raise ValueError('The audio must contain finite samples.')
    wave = data.mean(axis=1)
    notes = []
    if data.shape[1] > 1:
        notes.append('Stereo converted to mono.')
    if sr != config['sample_rate']:
        wave = librosa.resample(wave, orig_sr=sr, target_sr=config['sample_rate'])
        notes.append(f"Resampled from {sr} Hz to {config['sample_rate']} Hz.")
```

```
    if np.max(np.abs(wave)) < 1e-8:
        raise ValueError('The recording is silent.')
    if config['trim_silence']:
        wave, _ = librosa.effects.trim(wave, top_db=config['trim_top_db'])
    maximum = int(round(config['sample_rate'] * config['max_seconds']))
    if len(wave) > maximum:
        wave = wave[:maximum]
        notes.append(f"Only the first {config['max_seconds']} seconds are used.")
    if len(wave) < config['n_fft']:
        wave = np.pad(wave, (0, config['n_fft']-len(wave)))
    if config['normalize_peak']:
        wave = wave / max(float(np.max(np.abs(wave))), 1e-8)
    return wave.astype('float32'), notes

def extract_audio_features(wave, config):
    mel = librosa.feature.melspectrogram(y=wave, sr=config['sample_rate'],
n_fft=config['n_fft'],
                                         hop_length=config['hop_length'],
n_mels=config['n_mels'],
                                         fmax=config['sample_rate']/2, power=2.0)
    db = librosa.power_to_db(mel, ref=np.max, top_db=80).astype('float32')
    mfcc = librosa.feature.mfcc(S=db, n_mfcc=config['n_mfcc']).astype('float32')
    delta = librosa.feature.delta(mfcc, width=3, mode='nearest')
    pooled = np.concatenate([mfcc.mean(axis=1), mfcc.std(axis=1),
                             delta.mean(axis=1), delta.std(axis=1)]).astype('float32')
    # These are numeric features, not a resized screenshot of a plot.
    return pooled, mfcc.T, db.T

def play_full_recording(source, label='Full recording'):
    # Preserve uploaded BytesIO positions for subsequent feature extraction.
    position = source.tell() if hasattr(source, 'tell') else None
    if hasattr(source, 'seek'): source.seek(0)
    try:
        original, original_sr = sf.read(source, dtype='float32', always_2d=True)
    finally:
        if position is not None: source.seek(position)
    print(f'{label}: {len(original) / original_sr:.3f} s')
    display(Audio(original.T, rate=original_sr))

def preview_audio(source, config, title='Audio example', include_full=True):
    wave, notes = load_audio(source, config)
    pooled, mfcc, db = extract_audio_features(wave, config)
    if include_full: play_full_recording(source)
    print(f"Audio used by the model after preprocessing: {len(wave) / config['sample_rate']:.3f}
s")
    display(Audio(wave, rate=config['sample_rate']))
    for note in notes:
        print(note)
    fig, ax = plt.subplots(1, 3, figsize=(15, 3.2))
    ax[0].plot(np.arange(len(wave))/config['sample_rate'], wave, lw=0.7)
    ax[0].set(xlabel='Seconds', ylabel='Amplitude', title='Waveform')
    img = librosa.display.specshow(db.T, sr=config['sample_rate'],
hop_length=config['hop_length'],
                                   x_axis='time', y_axis='mel', ax=ax[1])
    ax[1].set_title('Log-mel spectrogram'); fig.colorbar(img, ax=ax[1], label='dB')
    ax[2].imshow(mfcc.T, origin='lower', aspect='auto'); ax[2].set(xlabel='Frame',
ylabel='MFCC', title='MFCC sequence')
    fig.suptitle(title); plt.tight_layout(); plt.show()
    print('Pooled MFCC vector:', pooled.shape, '| MFCC sequence:', mfcc.shape, '| Log-mel:',
db.shape)

def make_audio_splits(index, mode='official', test_speaker='george', val_speaker='jackson'):
```

```
    split = pd.Series('', index=index.index, dtype='object')
    if mode == 'official':
        split.loc[index.repeat <= 4] = 'test'
        split.loc[index.repeat.between(5, 9)] = 'validation'
        split.loc[index.repeat >= 10] = 'train'
    elif mode == 'speaker_independent':
        if test_speaker == val_speaker or not {test_speaker,
val_speaker}.issubset(set(index.speaker)):
            raise ValueError('Choose two different speakers present in the dataset.')
        split[:] = 'train'
        split.loc[index.speaker == val_speaker] = 'validation'
        split.loc[index.speaker == test_speaker] = 'test'
    else:
        raise ValueError('Unknown SPLIT_MODE.')
    result = index.copy(); result['split'] = split
    for name in ['train', 'validation', 'test']:
        part = result[result.split == name]
        if set(part.digit) != set(CLASSES):
            raise ValueError(f'All 10 classes are required in {name}.')
    # Prevent identical decoded audio crossing partitions.
    if result.groupby('sha256')['split'].nunique().max() > 1:
        raise ValueError('Duplicate audio crosses data partitions.')
    return result
```

Cell 6

```
def prepare_audio_dataset(index, config):
    pooled, sequences, spectrograms, notes = [], [], [], []
    for row in tqdm(index.itertuples(), total=len(index), desc='Audio features'):
        wave, message = load_audio(row.path, config)
        vec, seq, spec = extract_audio_features(wave, config)
        pooled.append(vec); sequences.append(seq); spectrograms.append(spec)
        notes.extend([{'file': row.file, 'message': m} for m in message])
    train_ids = np.flatnonzero(index.split.to_numpy() == 'train')
    scaler = StandardScaler().fit(np.concatenate([sequences[i] for i in train_ids], axis=0))
    max_frames = 1 + int(round(config['sample_rate']*config['max_seconds'])) //
config['hop_length']
    seq_array = np.zeros((len(index), max_frames, config['n_mfcc']), dtype='float32')
    cnn_array = np.zeros((len(index), max_frames, config['n_mels'], 1), dtype='float32')
    lengths = []
    for i, (seq, spec) in enumerate(zip(sequences, spectrograms)):
        n = min(len(seq), max_frames); lengths.append(n)
        # Fit on real training frames, scale real frames, then pad with zeros for Masking.
        seq_array[i, :n] = scaler.transform(seq[:n]).astype('float32')
        cnn_array[i, :n, :, 0] = np.clip((spec[:n]+80)/80, 0, 1)
    return dict(config=dict(config), index=index.reset_index(drop=True),
                pooled=np.stack(pooled), sequence=seq_array, cnn=cnn_array,
                frame_lengths=np.array(lengths), sequence_scaler=scaler,
                y=index.digit.to_numpy(dtype='int32'), max_frames=max_frames,
                split_mode=SPLIT_MODE, notes=notes,
                feature_names=[f'{kind}_{j+1}' for kind in ['mfcc_mean', 'mfcc_std',
'delta_mean', 'delta_std']
                               for j in range(config['n_mfcc'])])

def transform_new_audio(source, prepared):
    wave, notes = load_audio(source, prepared['config'])
    vec, seq, spec = extract_audio_features(wave, prepared['config'])
    n = min(len(seq), prepared['max_frames'])
    seq_x = np.zeros((1, prepared['max_frames'], prepared['config']['n_mfcc']), dtype='float32')
    cnn_x = np.zeros((1, prepared['max_frames'], prepared['config']['n_mels'], 1),
dtype='float32')
```

```
    seq_x[0, :n] = prepared['sequence_scaler'].transform(seq[:n])
    cnn_x[0, :n, :, 0] = np.clip((spec[:n]+80)/80, 0, 1)
    return {'pooled': vec[None], 'sequence': seq_x, 'cnn': cnn_x}, wave, notes
```

A1.7 Audio/speech inputs — Sopele methods

Source file: TAI_sopele_audio_classification.ipynb. Notebook cells 2, 4, 5, 6.

Cell 2

```
# Imports and shared configuration. Run after a kernel restart.
import os
os.environ.setdefault('TF_CPP_MIN_LOG_LEVEL', '2')
from pathlib import Path
import io, json, time, hashlib, uuid, copy, re
from zipfile import ZipFile
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import soundfile as sf
import librosa
import librosa.display
import joblib
import ipywidgets as widgets
from IPython.display import display, Audio, clear_output
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (accuracy_score, precision_score, recall_score, f1_score,
                             log_loss, confusion_matrix, ConfusionMatrixDisplay,
classification_report)
from tqdm.auto import tqdm

SEED = 42
DATA_SOURCE = Path('2. Sopele Sounds.zip')
OUT_DIR = Path('output_sopele')
OUT_DIR.mkdir(exist_ok=True)
TASK = 'single'             # 'single' or 'combined'; change and rerun preparation for a
separate experiment
FEATURE_MODE = 'fft'        # 'fft' or 'waveform' for ML, MLP and CNN
CFG = dict(sample_rate=22050, window_seconds=0.10, hop_seconds=0.10,
           fft_size=4096, max_frequency=6000, n_fft=512, hop_length=128, n_mels=40,
           silence_db=-35.0, max_windows_per_file=120,
           mixed_train_per_class=60, mixed_validation_per_class=30, mixed_test_per_class=30,
           relative_mix_gain_db=[-6.0, 0.0, 6.0])
USE_GPU = False             # set True before TensorFlow is initialized to use a configured GPU
MAX_EPOCHS = 20
BATCH_SIZE = 64
RUN_BASELINES = True        # one configuration per method
RUN_GRID = True             # train the expanded grid; completed matching fits are reused
METHODS = ['KNN', 'SVM', 'Random forest', 'MLP', 'CNN', 'LSTM', 'GRU']
CLASSES = np.arange(12)     # updated from prepared labels for each task
print('Working folder:', Path.cwd())
print('Output folder:', OUT_DIR.resolve())
```

Cell 4

```
def resolve_sopele_data(source):
    source = Path(source)
    if not source.exists():
        candidates = [Path('sopele dataset.zip'), Path('data'), Path('sopele_data')]
        source = next((p for p in candidates if p.exists()), source)
    if source.is_dir():
```

```
        candidates = [source, source/'data'] + list(source.glob('*/data'))
        for folder in candidates:
            if (folder/'single_tones').is_dir():
                return folder
        raise FileNotFoundError('The selected folder must contain single_tones/ or
data/single_tones/.')
    if not source.is_file():
        raise FileNotFoundError('Place the Sopele ZIP beside the notebook or set DATA_SOURCE.')
    with ZipFile(source) as outer:
        names=outer.namelist()
        if any('/single_tones/' in name for name in names):
            payload=None
        else:
            nested=next((name for name in names if name.lower().endswith('sopele
dataset.zip')),None)
            if nested is None:
                raise ValueError('No Sopele dataset archive found in DATA_SOURCE.')
            payload=outer.read(nested)
        archive=ZipFile(io.BytesIO(payload)) if payload is not None else outer
        destination=Path('sopele_data')
        for info in archive.infolist():
            parts=Path(info.filename).parts
            anchor=next((i for i,p in enumerate(parts) if p in
('single_tones','real_pieces')),None)
            if info.is_dir() or anchor is None or Path(info.filename).suffix.lower() not in
('.wav','.pdf'):
                continue
            relative=Path(*parts[anchor:])
            if '..' in relative.parts:
                raise ValueError('Invalid archive path.')
            target=destination/relative;target.parent.mkdir(parents=True,exist_ok=True)
            target.write_bytes(archive.read(info))
        if payload is not None:archive.close()
    return destination

def index_sopele_sources(folder):
    rows=[]
    for path in sorted((Path(folder)/'single_tones').glob('*/*.wav')):
        label=path.parent.name
        if re.fullmatch(r'[mv][0-5]',label) is None:
            continue
        wave,sr=sf.read(path,dtype='float32',always_2d=True)
        info=sf.info(path)
        if wave.size==0 or not np.isfinite(wave).all():
            raise ValueError(f'Invalid audio in {path.name}.')
        rows.append(dict(file=path.name,path=str(path.resolve()),label=label,
                         source_id=path.relative_to(folder).as_posix(),sample_rate=sr,
                         channels=info.channels,subtype=info.subtype,seconds=len(wave)/sr,
                         sha256=hashlib.sha256(wave.tobytes()).hexdigest()))
    index=pd.DataFrame(rows)
    expected={f'{voice}{tone}' for voice in 'mv' for tone in range(6)}
    if index.empty or set(index.label)!=expected:
        raise ValueError('All twelve tone folders m0–m5 and v0–v5 are required.')
    if index.groupby('label').size().min()<3:
        raise ValueError('At least three independent source files per tone are required.')
    if index.sha256.duplicated().any():
        raise ValueError('Duplicate source recordings must be resolved before partitioning.')
    return index

def split_sopele_sources(index,seed=SEED):
    result=index.copy();result['split']='train';rng=np.random.default_rng(seed)
    for label,group in result.groupby('label',sort=True):
```

```
        ids=rng.permutation(group.index.to_numpy())
        result.loc[ids[0],'split']='test'
        result.loc[ids[1],'split']='validation'
    for name in ['train','validation','test']:
        assert set(result.loc[result.split==name,'label'])==set(result.label)
    return result

def overview_sopele_sources(index):
    print(f'{len(index)} original recordings; {index.label.nunique()} tone classes.')
    display(index[['file','label','seconds','sample_rate','channels']].head())

display(index.groupby('label').agg(files=('file','size'),seconds=('seconds','sum')).round(3))
    display(pd.crosstab(index.label,index['split']))
    print('All windows from a source recording stay in its assigned split.')
```

Cell 5

```
def load_sopele_audio(source,config):
    data,sr=sf.read(source,dtype='float32',always_2d=True)
    if data.size==0 or not np.isfinite(data).all():
        raise ValueError('Audio must contain finite samples.')
    wave=data.mean(axis=1)
    if sr!=config['sample_rate']:
        wave=librosa.resample(wave,orig_sr=sr,target_sr=config['sample_rate'])
    return wave.astype('float32')

def make_sopele_windows(wave,config):
    n=int(round(config['sample_rate']*config['window_seconds']))
    hop=int(round(config['sample_rate']*config['hop_seconds']))
    if n<config['n_fft'] or hop<1 or config['fft_size']<n:
        raise ValueError('Use a window at least n_fft samples long and an FFT size at least the
window length.')
    if len(wave)<n:
        wave=np.pad(wave,(0,n-len(wave)))
    starts=np.arange(0,len(wave)-n+1,hop,dtype=int)
    windows=np.stack([wave[start:start+n] for start in starts])
    rms=np.sqrt(np.mean(windows.astype(float)**2,axis=1))
    threshold=max(float(rms.max())*10**(config['silence_db']/20),1e-8)
    active=rms>threshold
    peak=np.max(np.abs(windows),axis=1,keepdims=True)
    windows=windows/np.maximum(peak,1e-8)
    return windows.astype('float32'),starts,rms,active

def sopele_features(window,config,mode):
    if mode=='fft':
        magnitude=np.abs(np.fft.rfft(window*np.hanning(len(window)),n=config['fft_size']))
        frequencies=np.fft.rfftfreq(config['fft_size'],1/config['sample_rate'])
        magnitude=magnitude[frequencies<=config['max_frequency']]
        magnitude=magnitude/max(float(magnitude.max()),1e-8)
        vector=(np.clip(20*np.log10(np.maximum(magnitude,1e-8)),-80,0)+80)/80
    elif mode=='waveform':
        vector=window.copy()
    else:
        raise ValueError("FEATURE_MODE must be 'fft' or 'waveform'.")
    mel=librosa.feature.melspectrogram(y=window,sr=config['sample_rate'],n_fft=config['n_fft'],

hop_length=config['hop_length'],n_mels=config['n_mels'],power=2.0)
    sequence=(librosa.power_to_db(mel,ref=np.max,top_db=80).T+80)/80
    vector=np.asarray(vector,dtype='float32')
    return vector,vector[:,None],sequence.astype('float32')
```

```
def play_full_recording(source, label='Full recording'):
    # Preserve uploaded BytesIO positions for subsequent feature extraction.
    position = source.tell() if hasattr(source, 'tell') else None
    if hasattr(source, 'seek'): source.seek(0)
    try:
        original, original_sr = sf.read(source, dtype='float32', always_2d=True)
    finally:
        if position is not None: source.seek(position)
    print(f'{label}: {len(original) / original_sr:.3f} s')
    display(Audio(original.T, rate=original_sr))

def preview_sopele_wave(wave,config,title='Audio
window',source=None,start_seconds=None,source_b=None):
    vector,_,seq=sopele_features(wave,config,FEATURE_MODE)
    if source is not None: play_full_recording(source, 'Full source recording')
    if source_b is not None: play_full_recording(source_b, 'Full second source recording')
    duration=len(wave)/config['sample_rate']
    interval='' if start_seconds is None else f' | source interval {start_seconds:.3f}–
{start_seconds+duration:.3f} s'
    print(f'Window used by the model: {duration:.3f} s{interval}')
    display(Audio(wave,rate=config['sample_rate']))
    fig,ax=plt.subplots(1,3,figsize=(14,3))

ax[0].plot(np.arange(len(wave))/config['sample_rate'],wave);ax[0].set(xlabel='Seconds',title='Wa
veform')
    ax[1].plot(vector);ax[1].set(xlabel='Feature index',title=FEATURE_MODE.upper())
    ax[2].imshow(seq.T,origin='lower',aspect='auto');ax[2].set(xlabel='Time frame',ylabel='Mel
bin',title='Log-mel sequence')
    fig.suptitle(title);plt.tight_layout();plt.show()
```

Cell 6

```
def prepare_sopele_dataset(source_index,config,task,feature_mode):
    rng=np.random.default_rng(SEED)
    source_pool={name:{} for name in ['train','validation','test']}
    for row in tqdm(source_index.itertuples(),total=len(source_index),desc='Source windows'):
        wave=load_sopele_audio(row.path,config)
        windows,starts,rms,active=make_sopele_windows(wave,config)
        selected=np.flatnonzero(active)
        if not len(selected):raise ValueError(f'No active windows in {row.file}.')
        limit=config['max_windows_per_file']
        if limit and len(selected)>limit:
            selected=selected[np.linspace(0,len(selected)-1,limit,dtype=int)]
        for i in selected:
            source_pool[row.split].setdefault(row.label,[]).append(

dict(wave=windows[i],source_id=row.source_id,start=float(starts[i]/config['sample_rate'])))
    labels=sorted(source_index.label.unique()) if task=='single' else [f'm{m}v{v}' for m in
range(6) for v in range(6)]
    if task not in ('single','combined'):raise ValueError('Choose TASK single or combined.')
    rows=[];waves=[]
    for split,pool in source_pool.items():
        if task=='single':
            for label in labels:
                for item in pool[label]:
                    rows.append(dict(file=f"{item['source_id']} @
{item['start']:.3f}s",label=label,
                                     split=split,source_a=item['source_id'],source_b='',
                                     start_a=item['start'],start_b=np.nan,gain_db=0.0))
                    waves.append(item['wave'])
        else:
```

```
            count=config[f'mixed_{split}_per_class']
            for label in labels:
                left,right=label[:2],label[2:]
                # A and B were already assigned to the same source split, before mixing.
                for _ in range(count):

a=pool[left][rng.integers(len(pool[left]))];b=pool[right][rng.integers(len(pool[right]))]
                    gain=float(rng.choice(config['relative_mix_gain_db']))
                    mixed=a['wave']+b['wave']*10**(gain/20)
                    mixed=mixed/max(float(np.max(np.abs(mixed))),1e-8)
                    rows.append(dict(file=f"{label} | {a['source_id']} +
{b['source_id']}",label=label,split=split,
                                     source_a=a['source_id'],source_b=b['source_id'],
                                     start_a=a['start'],start_b=b['start'],gain_db=gain))
                    waves.append(mixed.astype('float32'))
    index=pd.DataFrame(rows)

source_sets={part:set(index.loc[index.split==part,'source_a'])|set(index.loc[index.split==part,'
source_b'])- {''}
                 for part in ['train','validation','test']}
    for a,b in [('train','validation'),('train','test'),('validation','test')]:
        if source_sets[a]&source_sets[b]:raise ValueError('Source recordings cross partitions.')
    pooled=[];cnn=[];sequence=[]
    for wave in tqdm(waves,desc='Audio representations'):
        vector,c,s=sopele_features(wave,config,feature_mode)
        pooled.append(vector);cnn.append(c);sequence.append(s)
    label_to_index={label:i for i,label in enumerate(labels)}
    nfeatures=len(pooled[0])
    feature_names=([f'fft_{hz:.1f}Hz' for hz in
np.fft.rfftfreq(config['fft_size'],1/config['sample_rate'])
                    if hz<=config['max_frequency']] if feature_mode=='fft'
                  else [f'sample_{i}' for i in range(nfeatures)])
    return
dict(config=dict(config),task=task,feature_mode=feature_mode,labels=labels,index=index,

source_index=source_index.copy(),waves=np.stack(waves),pooled=np.stack(pooled),cnn=np.stack(cnn)
,

sequence=np.stack(sequence),y=index.label.map(label_to_index).to_numpy(dtype='int32'),
                feature_names=feature_names,frame_lengths=np.full(len(rows),len(sequence[0])),
                source_sets={k:sorted(v) for k,v in source_sets.items()})

def transform_sopele_recording(source,prepared):
    wave=load_sopele_audio(source,prepared['config'])
    windows,starts,rms,active=make_sopele_windows(wave,prepared['config'])
    selected=np.flatnonzero(active)
    if not len(selected):raise ValueError('No active audio windows found.')
    representations=[sopele_features(windows[i],prepared['config'],prepared['feature_mode']) for
i in selected]
    inputs={name:np.stack([r[j] for r in representations])
            for j,name in enumerate(['pooled','cnn','sequence'])}
    return inputs,wave,windows,starts,rms,active,selected
```

A1.8 Numerical and industrial data — shared methods

The following definitions are taken from TAI_concrete_regression.ipynb, cells 4–17. The executable method definitions in the corresponding sections of the meteorological and loan notebooks are identical; dataset-specific configuration and calls are listed separately.

```
# Read a CSV file into a DataFrame. Column names are stripped of extra spaces.
def load_data(path):
    df = pd.read_csv(path)
    df.columns = [c.strip() for c in df.columns]
    print(f"Loaded '{path}': {df.shape[0]} rows, {df.shape[1]} columns.")
    return df


# Print a short overview of the data: size, column types, first rows and basic statistics.
def overview(df):
    print(f"Rows: {df.shape[0]}, columns: {df.shape[1]}")
    print("\nColumn types:")
    print(df.dtypes.to_string())
    print("\nFirst 5 rows:")
    display(df.head())
    print("Basic statistics (numeric columns):")
    display(df.describe().T.round(3))


# Count missing values in each column. Returns a table with counts and percentages.
def check_missing(df):
    counts = df.isna().sum()
    table = pd.DataFrame({'missing': counts, 'percent': (100 * counts / len(df)).round(2)})
    table = table[table['missing'] > 0]
    if table.empty:
        print("No missing values found.")
    else:
        print(f"Missing values found in {len(table)} column(s):")
        display(table)
    return table


# Count fully duplicated rows.
def check_duplicates(df):
    n = int(df.duplicated().sum())
    if n == 0:
        print("No duplicated rows found.")
    else:
        print(f"Found {n} duplicated row(s) ({100 * n / len(df):.2f}% of the data).")
    return n


# Find values further than 'threshold' standard deviations from the column mean.
# Binary columns (0/1) are skipped. Nothing is changed here, only a warning and a summary table.
# Returns a dictionary column -> True/False mask of the outlier rows, used by show_outliers()
and fix_outliers().
def check_outliers(df, threshold=3):
    masks, rows = {}, []
    for col in df.select_dtypes(include='number').columns:
        s = df[col]
        if s.nunique() <= 2 or s.std() == 0:
            continue
        mean, std = s.mean(), s.std()
        mask = ((s - mean) / std).abs() > threshold
        n = int(mask.sum())
        if n > 0:
```

```
            masks[col] = mask.fillna(False)
            rows.append({'column': col, 'outliers': n, 'percent': round(100 * n / len(df), 2),
                         'lower_limit': mean - threshold * std, 'upper_limit': mean + threshold
* std,
                         'outlier_min': s[mask].min(), 'outlier_max': s[mask].max()})
            print(f"Warning: column '{col}' has {n} value(s) outside +-{threshold} standard
deviations.")
    if not masks:
        print(f"No values outside +-{threshold} standard deviations.")
    else:
        print("Nothing is removed yet. Use show_outliers() to look at the rows and
fix_outliers() to remove them.")
        display(pd.DataFrame(rows).set_index('column').round(3))
    return masks


# Show the most extreme outlier rows for every column found by check_outliers().
def show_outliers(df, outliers, n=5):
    if not outliers:
        print("No outliers to show.")
        return
    for col, mask in outliers.items():
        mask = mask.reindex(df.index, fill_value=False)
        rows = df[mask]
        order = (rows[col] - df[col].mean()).abs().sort_values(ascending=False).index
        print(f"Column '{col}': {len(rows)} outlier row(s), showing the {min(n, len(rows))} most
extreme:")
        display(rows.loc[order[:n]])


# Widget for choosing the columns whose outlier rows will be removed. Nothing is selected by
default.
def select_outliers(outliers):
    cols = list(outliers.keys())
    w = widgets.SelectMultiple(options=cols, value=[], description='Remove',
                               rows=max(2, min(len(cols), 10)),
layout=widgets.Layout(width='60%'))
    if cols:
        print("Select the columns whose outlier rows should be removed (Ctrl+click for several).
Leave empty to keep everything.")
    else:
        print("No outliers were found, nothing to select.")
    display(w)
    return w

# Remove the rows that are outliers in any of the selected columns.
def fix_outliers(df, outliers, columns):
    columns = [c for c in columns if c in outliers]
    if not columns:
        print("No columns selected, no rows removed.")
        return df
    remove = pd.Series(False, index=df.index)
    for col in columns:
        mask = outliers[col].reindex(df.index, fill_value=False)
        print(f"Column '{col}': {int(mask.sum())} outlier row(s) marked for removal.")
        remove |= mask
    df = df[~remove].reset_index(drop=True)
    print(f"Removed {int(remove.sum())} row(s) in total (some rows are outliers in more than one
column). {len(df)} rows remain.")
    return df


# Fill or remove missing values.
```

```
# method: 'mean', 'median' or 'drop'. Text columns are always filled with the most frequent
value.
def fix_missing(df, method='mean'):
    df = df.copy()
    total = int(df.isna().sum().sum())
    if total == 0:
        print("No missing values, nothing to do.")
        return df
    if method == 'drop':
        before = len(df)
        df = df.dropna()
        print(f"Removed {before - len(df)} row(s) with missing values.")
        return df
    for col in df.columns:
        n = int(df[col].isna().sum())
        if n == 0:
            continue
        if pd.api.types.is_numeric_dtype(df[col]):
            value = df[col].mean() if method == 'mean' else df[col].median()
            df[col] = df[col].fillna(value)
            print(f"Column '{col}': filled {n} missing value(s) with the {method}
({value:.4g}).")
        else:
            value = df[col].mode().iloc[0]
            df[col] = df[col].fillna(value)
            print(f"Column '{col}': filled {n} missing value(s) with the most frequent value
('{value}').")
    return df


# Remove duplicated rows. The first occurrence is kept.
def fix_duplicates(df):
    before = len(df)
    df = df.drop_duplicates().reset_index(drop=True)
    removed = before - len(df)
    if removed == 0:
        print("No duplicated rows, nothing to do.")
    else:
        print(f"Removed {removed} duplicated row(s). {len(df)} rows remain.")
    return df


# Show widgets for choosing input columns and the output column.
# Read the choice afterwards with get_io().
def select_io(df, default_target, default_inputs=None):
    cols = list(df.columns)
    if default_inputs is None:
        default_inputs = [c for c in cols if c != default_target]
    inputs = widgets.SelectMultiple(options=cols, value=default_inputs, description='Inputs',
                                    rows=min(len(cols), 15), layout=widgets.Layout(width='60%'))
    output = widgets.Dropdown(options=cols, value=default_target, description='Output')
    print("Select the input columns (Ctrl+click for several) and the output column.")
    display(inputs, output)
    return {'inputs': inputs, 'output': output}

# Read the current widget values and print them.
def get_io(io):
    input_cols = [c for c in io['inputs'].value if c != io['output'].value]
    target_col = io['output'].value
    print(f"Inputs ({len(input_cols)}): {input_cols}")
    print(f"Output: '{target_col}'")
    return input_cols, target_col
```

```
# Find columns that look categorical: text columns, and integer columns with few distinct values
(0, 1, 2, ...).
# Text columns are pre-selected. Integer candidates are listed but must be selected by the user.
def select_categorical(df, max_unique=10):
    text_cols = [c for c in df.columns if not pd.api.types.is_numeric_dtype(df[c])]
    int_cands = [c for c in df.columns if c not in text_cols
                 and pd.api.types.is_integer_dtype(df[c]) and df[c].nunique() <= max_unique]
    if text_cols:
        print("Text columns (treated as categorical):")
        for c in text_cols:
            print(f"  '{c}': {df[c].nunique()} categories -> {list(df[c].unique()[:6])}")
    else:
        print("No text columns found.")
    if int_cands:
        print(f"Integer columns with at most {max_unique} distinct values (possible categorical,
not selected by default):")
        for c in int_cands:
            print(f"  '{c}': values {sorted(int(v) for v in df[c].unique())}")
    else:
        print("No integer columns with few distinct values.")
    w = widgets.SelectMultiple(options=text_cols + int_cands, value=text_cols,
description='Categorical',
                               rows=max(2, min(len(text_cols) + len(int_cands), 10)),
layout=widgets.Layout(width='60%'))
    display(w)
    return w


# Replace categorical columns with dummy (0/1) columns. A column with k categories gives k-1
dummies.
def make_dummies(df, categorical_cols):
    if not categorical_cols:
        print("No categorical columns, nothing to do.")
        return df
    before = df.shape[1]
    out = pd.get_dummies(df, columns=list(categorical_cols), drop_first=True, dtype=int)
    print(f"Replaced {len(categorical_cols)} categorical column(s) with {out.shape[1] - before +
len(categorical_cols)} dummy column(s).")
    print(f"Columns before: {before}, after: {out.shape[1]}.")
    return out


# Split the data into training and test sets.
# shuffle=True: random split. shuffle=False: the first rows go to training (use this for time
series).
from sklearn.model_selection import train_test_split

def split_data(X, y, test_size=0.2, shuffle=True):
    if shuffle:
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size,
shuffle=True, random_state=SEED)
        print(f"Random split: {len(X_train)} training rows, {len(X_test)} test rows.")
    else:
        n_train = int(len(X) * (1 - test_size))
        X_train, X_test = X.iloc[:n_train], X.iloc[n_train:]
        y_train, y_test = y.iloc[:n_train], y.iloc[n_train:]
        print(f"Chronological split (no shuffle): first {len(X_train)} rows for training, last
{len(X_test)} rows for test.")
    return X_train, X_test, y_train, y_test
```

```
# Scale all columns to the range 0-1. The scaler is fitted on the training set only.
# Test values can fall slightly outside 0-1, this is expected.
from sklearn.preprocessing import MinMaxScaler

def scale_data(X_train, X_test, y_train=None, y_test=None, scale_target=False):
    x_scaler = MinMaxScaler().fit(X_train)
    X_train_s = x_scaler.transform(X_train)
    X_test_s = x_scaler.transform(X_test)
    print(f"Inputs scaled to 0-1 using the training set ({X_train.shape[1]} columns).")
    y_scaler = None
    if scale_target:
        y_scaler = MinMaxScaler().fit(np.asarray(y_train).reshape(-1, 1))
        y_train = y_scaler.transform(np.asarray(y_train).reshape(-1, 1)).ravel()
        y_test = y_scaler.transform(np.asarray(y_test).reshape(-1, 1)).ravel()
        print("Output scaled to 0-1 as well. Metrics and plots will be shown in original
units.")
    else:
        y_train, y_test = np.asarray(y_train), np.asarray(y_test)
    return X_train_s, X_test_s, y_train, y_test, x_scaler, y_scaler
```