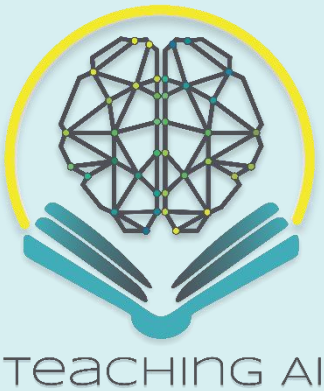

Report WP3-A4:

Implementing methods for data preparation (cleaning, conversion, validation, reduction)

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

Description of data cleaning, conversion, validation and reduction methods applied to video, image, audio/speech and numerical data.

Related to

WP3-A4: Implementing methods for data preparation (cleaning, conversion, validation, reduction)

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This report contains original unpublished work, except where clearly indicated otherwise.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. Organisation of data preparation 5

3. Common preparation principles 6

4. Video recordings 7

5. Still images 9

6. Audio/speech inputs 10

7. Numerical and industrial data 11

8. Preparation records and availability 13

9. Final conclusions 14

Annex 1. Python code used for data preparation 15

1. Introduction

Preparing data for AI exercises requires more than reading source files. Values, labels, media representations and partition definitions must be organised so that the information supplied to a model corresponds to the intended target. This report describes the preparation methods implemented in the available notebooks and supporting scripts.

The description follows video recordings, still images, audio/speech inputs and numerical and industrial data. The focus is the preparation of usable representations rather than model architectures, hyperparameter searches or performance comparisons. The final conclusions summarise the preparation methods, partition designs and information required to reuse the prepared resources.

The accompanying WP3-A3 report explains the organisation and execution of the notebooks. Here, the same examples are described through cleaning, conversion, validation and reduction. A category may involve no change where the relevant data already meet the required conditions.

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

2. Organisation of data preparation

A preparation workflow connects original observations to model-facing inputs and targets. It may retain source identifiers, timestamps, category definitions and transformation settings so that the conversion can be inspected and repeated. The specific operations depend on the unit of observation and the intended task.

Method category	Role in the available implementations	Examples
Cleaning	Handle missing, duplicated or unusable information without changing its meaning.	Missing table values, duplicate rows, blank clip labels and unavailable GPS targets.
Conversion	Represent source content as numerical inputs and appropriate targets.	Dummy columns, scaled arrays, RGB tensors, MFCCs, FFT features and optical flow.
Validation	Check whether files, values, labels and partitions satisfy the workflow requirements.	Filename matching, class checks, finite-value checks, timestamp alignment and source separation.
Reduction	Control the amount or dimensions of information presented to the model.	Column selection, duration limits, window sampling, image resizing and spectral limits.

These categories describe the function of an operation. They do not require every dataset to undergo all possible transformations. For example, an outlier warning does not automatically justify deleting a row, and an audio recording does not need missing-value imputation merely because a numerical table does.

3. Common preparation principles

3.1 Preserve the unit of observation

A source identifier links prepared observations back to the original material. Video clips retain their time intervals; Garmin windows retain the source video and drive; audio windows retain their source recording; numerical tables retain the selected feature and target definitions. This context helps explain what constitutes one training example.

3.2 Separate targets from predictors

Digit and tone identities, motion classes, GPS speeds and selected numerical outputs are targets. They must be distinguished from the representations used as predictors. Metadata can support grouping and inspection without being included in X. In particular, Garmin GPS readings supervise video-based speed estimation rather than form part of the model input.

3.3 Record the partition and transformation settings

The examples use different partition designs: temporal regions in one film, a held-out driving session, repetition-based or speaker-independent audio partitions, source-recording tone partitions and shuffled or ordered numerical partitions. These designs are recorded explicitly rather than treated as interchangeable. Transformation settings are saved where the code provides a reusable preparation object or manifest.

In the numerical notebooks, missing-value handling, optional row removal and dummy encoding occur before the 80/20 split. MinMax scalers are fitted on the resulting training partition and applied to the test partition. The report describes that implemented order; it does not imply that every transformation is fitted separately within a validation fold.

4. Video recordings

4.1 Popeye clip preparation

The notebook first checks that the source MP4 can be opened and has a valid frame rate, frame count and dimensions. Candidate clips are created from non-overlapping 2-second intervals. With the default configuration, 16 frames are sampled across each interval and converted into RGB images of 64 by 64 pixels.

Frames are resized without stretching the original proportions. Padding fills the remaining square area. Clips are stored as uint8 arrays for preparation and inspection; model input conversion rearranges the axes and maps pixel values to floating-point values in 0–1. This is a fixed pixel conversion, rather than a fitted statistical scaler.

A manual CSV provides the low_motion and high_motion labels. Duplicate or missing clip identifiers, unknown identifiers, invalid class values and mismatched timestamps or partitions are checked. Blank and skip labels are excluded from modelling, as are clips assigned to boundary gaps. Both classes are required in each evaluated partition.

Temporal cutoffs assign approximately the first 70% of the source timeline to training, the next 15% to validation and the last 15% to test, subject to the configured boundary gaps and the available labelled clips. These are regions of one film, not independently held-out films. The cache signature distinguishes the source video and extraction settings.

Category	Popeye implementation
Cleaning	Exclude blank/skip labels and unused boundary-gap clips; reject inconsistent annotation records.
Conversion	Decode RGB frames, preserve proportions through padding, construct clip tensors and scale pixels.
Validation	Check video metadata, identifiers, class values, interval alignment and class presence.
Reduction	Use 2-second clips, 16 sampled frames and 64 by 64 pixel images by default.

4.2 Garmin video and GPS preparation

Each video is matched to a speed CSV using its stem. Timestamp parsing accepts the supported second-based formats and checks whole-second values. The speed reader retains genuine numeric zero values, omits missing N/A readings, and rejects non-finite, implausible or duplicated valid measurements.

The metadata-export helper can create speed labels from supported Garmin metadata. Its configurable speed correction is specific to the supplied camera/dataset workflow. Existing

corrected CSV values are treated as km/h and are not converted again during preparation. Drive membership is supplied through the grouping CSV.

FFmpeg samples the visual stream at 5 FPS, centre-crops it, resizes it and converts it to grayscale. The default image size is 128 by 128 pixels. The lower 10% of the image is masked under the default setting, and the crop preview is used to inspect whether the visible speed/GPS overlay has been suppressed.

OpenCV computes Farneback optical flow between consecutive frames. Each 10-second window contains 50 two-channel flow maps and 10 speed targets. Flow values are clipped using the configured range and saved in a compact representation. Missing GPS readings are represented by a validity mask, so missing intervals are not converted into stationary-vehicle labels or closed by splicing unrelated video sections.

A nominal 60-second file may provide five full non-overlapping windows and an overlapping tail window. Already supervised seconds in the tail are masked. The split procedure keeps overlapping windows together. One complete drive is retained for test; clips from the remaining drives are randomly assigned to training and validation. Thus, training and validation can contain clips from the same drive.

Category	Garmin implementation
Cleaning	Distinguish zero from missing GPS; exclude unavailable targets using a mask; reject invalid label rows.
Conversion	Resample, crop, resize, mask and convert frames to optical flow; align ten per-second targets.
Validation	Check file pairing, timestamps, values, drive mapping, overlap groups and held-out test membership.
Reduction	Use 5 FPS, 10-second windows, a configurable square resolution and compact flow arrays.

5. Still images

5.1 Inspection and cleaning

The loader reads the separate local training and test folders, converts readable images to RGB and reports files that could not be opened. Hashes of decoded image arrays identify exact duplicates within the sampled training pool and exact overlap with the test pool. The applied workflow removes repeated training images and test images also present in the training pool. It does not perform a separate within-test duplicate-removal pass. In the supplied recorded execution, no unreadable files, sampled training duplicates or cross-pool matches were reported.

5.2 Conversion and visual inspection

Images of a different size are resized to 32 by 32 pixels. Grids, numerical pixel displays, RGB-channel views, grayscale and HSV conversions, and intensity histograms support inspection. The alternative colour representations are demonstrations; the saved model input remains RGB. Class selection is applied to both pools, with retained classes assigned consecutive integer labels. The one-hot table explains class encoding, while the saved target arrays contain integer labels.

5.3 Validation and partition settings

Brightness and contrast statistics flag images more than three standard deviations from the sampled training-pool averages. A widget allows the user to remove selected kinds; none is selected by default. Recorded outputs flag 13 dark images, 66 bright images and three low-contrast images, without removing any. The retained training pool is split with stratification, validation fraction 0.15 and seed 42. The recorded split contains 12,750 training and 2,250 validation images, with 10,000 test images kept separately. These are the supplied execution counts; optional selections or different input files can change them.

5.4 Reduction and prepared outputs

The 1,500-per-class training-pool cap, optional class selection, optional image removal and fixed spatial size control the prepared data volume. Default pixel scaling is the fixed conversion to floating-point values in 0–1; the optional standard method estimates channel means and standard deviations from training images only. The saved prepared.npz retains unscaled uint8 image arrays for all three partitions and their labels; prepared_meta.joblib retains the class mapping, image size, scaler and test-file paths. Augmentation previews demonstrate flips, rotation, zoom, translation and contrast changes. They do not add images to the saved arrays; augmentation is applied by the later neural-network training code according to its selected setting.

6. Audio/speech inputs

6.1 FSDD recording preparation

The recording index derives digit, speaker and repetition metadata from the filename. File and decoded-signal checks support the preparation workflow, including duplicate-content checks across partitions. The default repetition split assigns indices 0–4 to test, 5–9 to validation and 10–49 to training. An alternative configuration holds out selected speakers for validation and test.

Audio is decoded to floating-point samples, multichannel input is averaged to mono and recordings are resampled to 8 kHz where needed. Optional silence trimming is disabled by default. Audio longer than the configured 2.5-second cap is truncated; very short signals are padded to support spectral extraction. Peak normalisation is applied under the default configuration.

The feature extractor produces 20 MFCC coefficients, their time derivatives and 40-band log-mel representations. MFCC and delta means and standard deviations form an 80-value fixed-length vector. Sequence and spectrogram arrays are padded to a common frame capacity, and the original effective sequence lengths are retained.

The MFCC sequence scaler is fitted using real training frames only. It is applied before zero padding. Log-mel values use a fixed bounded conversion. Labels, recording indices, lengths and scaler information are retained with the prepared data, so the same representation can be applied to a new recording.

6.2 Sopele source and window preparation

The source index checks the expected 12 tone classes and sufficient original recordings for the source-level partition. One original file per class is assigned to validation and one to test; the remaining files train the models. Whole source files are assigned before windows are extracted, so windows from a source remain in the same partition.

Recordings are decoded, averaged to mono where needed and resampled to 22.05 kHz. The default window and hop are both 0.10 seconds. An energy threshold identifies active windows and per-window peak normalisation prepares their amplitudes. At most 120 active windows per file are selected under the default limit.

FFT features use a 4096-point transform and retain frequencies up to 6 kHz. The configured alternative retains waveform samples. A 40-band log-mel sequence is also produced for sequence-oriented inputs. The source/window index records the source identity and window start.

For the combined-tone task, the notebook constructs 36 classes by mixing m and v windows from the same partition. Relative gain can take the configured values −6, 0 or 6 dB. The default generated counts are 60 training, 30 validation and 30 test examples per combined class. These are generated examples, not counts of independent original recordings.

7. Numerical and industrial data

7.1 Inspection and cleaning

The numerical notebooks read their CSVs into pandas tables and strip extra spaces from column names. Overview functions display dimensions, variable types, initial rows and numerical summaries. Missing-value counts and percentages and the number of fully duplicated rows are reported before correction.

The default missing-value operation uses column means for numeric columns and the most frequent value for text columns. Median filling and dropping incomplete rows are alternative settings. Duplicate rows are removed while retaining the first occurrence. These operations apply where the relevant condition occurs; the report does not assume that every input file contains missing or duplicated data.

Outlier checks flag numeric values more than three standard deviations from their column mean, excluding binary or constant columns. They first display warnings and example rows. A user selector determines whether any flagged columns are used for row removal; none is selected by default. In the time-series example, the usual empty selection preserves the ordered sequence and real high-pollution or precipitation episodes.

7.2 Conversion and input selection

Interactive controls select input and output columns. Selected categorical predictors are expanded into n−1 dummy columns. The feature order and category information are retained with the prepared arrays. In the concrete example, Age remains numerical under the default cardinality threshold; a binary temperature-inversion variable in the time-series example can remain numeric.

Concrete and loan examples use shuffled 80/20 partitions with a fixed seed. The meteorological example keeps the existing row order and does not shuffle. Correct temporal order must therefore be present in the source table. MinMax scaling is fitted on training inputs and then applied to test inputs. The regression and time-series targets are scaled separately; loan_status remains a class label.

7.3 Dataset-specific outputs

Dataset	Preparation choices	Saved representation
Concrete compressive strength	Strength target; selected mix-design/age inputs; optional row filtering; shuffled 80/20 split; input and target scaling.	X/y train/test arrays, scalers, feature names and prediction-input metadata.
Meteorological and air-quality data	PM2.5 target; PM10 omitted from default inputs; ordered 80/20 split; selected columns and input/target scaling.	Ordered arrays, scalers, feature names and example historical inputs for the later sequence interface.
Loan applicant data	loan_status target; selected applicant attributes; categorical encoding; shuffled 80/20 split; input scaling.	Predictor arrays, class targets, encoded feature order and input metadata.

Each notebook saves prepared.joblib in its own output folder. The preparation object contains the arrays and transformations needed by subsequent notebook stages. Column selection and optional row filtering provide the implemented reduction controls; the notebooks do not automatically apply PCA or another general dimensionality-reduction method.

8. Preparation records and availability

8.1 Records produced by the workflows

Preparation records make the transformation from source data to model inputs inspectable. The relevant output depends on the example: a numerical preparation object, an audio source/window index or a video manifest. Raw observations and generated arrays should be distinguished when describing the final resource.

Example	Preparation record	Information to retain
Popeye	Clip table and annotation CSV	Clip identifiers, intervals, labels, partitions and extraction settings.
Garmin	Manifest, label audit and drive/split CSVs	Source videos, drive identity, window start, valid targets and flow settings.
CIFAR-10	prepared.npz and prepared_meta.joblib	Partition arrays, integer targets, class names, image size, scaler and test paths.
FSDD	Prepared object and recording index	Digit/speaker/repetition metadata, arrays, lengths and scaler.
Sopele	Prepared object, source index and window index	Source partitions, tone classes, window timing and mixture settings.
Numerical examples	prepared.joblib in each output folder	Inputs, target, category/feature order, arrays and scalers.

8.2 Access and reuse

The code and instructions can be shared as notebook packages with the associated helper scripts and environment information. Dataset redistribution depends on the relevant permissions. Where raw media or institutional tables are supplied separately, the resource description should explain how an authorised user can obtain the files and arrange the configured folders. The CIFAR-10 example uses local image folders; its notebook does not download or extract the dataset.

Availability field	Details to complete
Notebook/code location	[Project-platform or repository link to be added]
Prepared-resource access	[Download or authorised-access instructions to be added]
Final preparation summary	[Measured source/prepared counts and retained execution outputs to be added]

9. Final conclusions

9.1 Consolidated summary of preparation methods

The video, still-image, audio/speech and numerical examples illustrate cleaning, conversion, validation and reduction as dataset-specific preparation operations. They produce explicit predictors and targets while retaining identifiers, partitions and transformation settings. The workflows connect source observations to representations that can be inspected before training, rather than applying the same sequence of transformations to every modality.

9.2 Partition design and interpretation

Partition design determines how later model results should be interpreted. Popeye separates temporal regions within one film; Garmin retains a complete drive for testing; CIFAR-10 retains a separate test pool and draws a stratified validation subset from the training pool; FSDD offers repetition-based and speaker-independent partitions; and Sopele assigns source recordings before window extraction and mixing. Numerical examples use shuffled or ordered partitions according to the task. These choices define different evaluation conditions. In the numerical notebooks, imputation and dummy encoding precede partitioning, while scalers are fitted on training data; this order should be considered when interpreting later performance estimates.

9.3 Prepared resources and reuse

Prepared data should be distributed with feature and target definitions, label mappings, source indices, partition membership and fitted transformations where applicable. This information allows later prediction inputs to follow the same representation and makes preparation choices traceable. The included code provides the applied procedures; final source and prepared counts, retained execution outputs and access instructions should accompany the resources when the packages are run and shared. No model-performance conclusion follows from preparation alone.

Annex 1. Python code used for data preparation

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 application

Source file: TAI_Popeye_3D_CNN_classification.ipynb. Notebook cells 13, 14, 15, 17, 18, 20.

Cell 13

```
# Step 1: inspect video, create clip IDs and assign temporal partitions.
video_info = inspect_video(VIDEO_FILE)
clip_table = make_clip_table(video_info, CLIP_SECONDS, STRIDE_SECONDS, MAX_CLIPS)
clip_table = temporal_split(clip_table, video_info['duration_s'], SPLIT_GAP_SECONDS)
print('Candidate clips:', len(clip_table))
display(clip_table.head())
print('Clips by temporal partition:')
display(clip_table['split'].value_counts().reindex(['train', 'val', 'test', 'gap'],
fill_value=0).to_frame('clips'))
```

Cell 14

```
# Step 2: extract samples (or reuse a cache for identical video/clip settings).
clips = extract_video_clips(VIDEO_FILE, video_info, clip_table, NUM_FRAMES,
                            IMAGE_SIZE, CLIP_SECONDS, OUT_DIR, cache=True)
inspect_clips(clips, clip_table)
show_clip(clips, 0, label='Clip 0: uniformly sampled RGB frames',
          indices=(0, NUM_FRAMES // 3, 2 * NUM_FRAMES // 3, NUM_FRAMES - 1))
```

Cell 15

```
# Step 3: visualize WHICH time ranges belong to each partition.
fig, ax = plt.subplots(figsize=(10, 3.5))
parts = ['train', 'val', 'test', 'gap']
colors = {'train': 'tab:blue', 'val': 'tab:orange',
          'test': 'tab:green', 'gap': 'tab:gray'}
for row_index, part in enumerate(parts):
    subset = clip_table[clip_table['split'] == part]
    ax.scatter(subset['start_s'] / 60, np.full(len(subset), row_index),
               color=colors[part], s=24, label=part)
ax.set_yticks(range(len(parts)), parts)
ax.set(xlabel='Time in film (minutes)', title='Time-based video clip partitions')
ax.grid(axis='x', alpha=0.2)
plt.tight_layout()
plt.show()
```

Cell 17

```
# Generate the CSV file once.
ensure_annotation_csv(clip_table, ANNOTATION_CSV)

# Contact sheet for manual review. Change REVIEW_START to 8, 16, 24, ...
REVIEW_START = 0
REVIEW_COUNT = 8
sample_ids = np.arange(REVIEW_START, min(REVIEW_START + REVIEW_COUNT, len(clips)))
if len(sample_ids) == 0:
    raise ValueError('REVIEW_START is beyond the last clip.')
fig, axes = plt.subplots(len(sample_ids), 4, figsize=(10, 2.2 * len(sample_ids)), squeeze=False)
for row_i, idx in enumerate(sample_ids):
    for col_i, frame_i in enumerate(np.linspace(0, NUM_FRAMES - 1, 4).astype(int)):
        axes[row_i, col_i].imshow(clips[idx, frame_i])
        axes[row_i, col_i].axis('off')
        if col_i == 0:
            axes[row_i, col_i].set_title(f"{clip_table.loc[idx, 'clip_id']} | "
```

```
                                        f"{clip_table.loc[idx, 'start_s']:.1f}s", fontsize=9)
plt.tight_layout()
plt.show()

# To watch the 16 sampled frames move, uncomment this optional call:
# preview_clip_gif(clips, 0)
```

Cell 18

```
# Step 5: import MANUAL CSV labels. An unfilled template stops execution.
clip_table = load_manual_labels(clip_table, ANNOTATION_CSV, CLASS_NAMES)

# Exclude blank/skip rows and gap clips from modeling.
usable = clip_table[clip_table['label'].isin(CLASS_NAMES) &
                    clip_table['split'].isin(['train', 'val', 'test'])].copy()
counts = pd.crosstab(usable['split'], usable['label']).reindex(
    index=['train', 'val', 'test'], columns=list(CLASS_NAMES), fill_value=0)
print('Manually annotated class counts in each temporal partition:')
display(counts)
if (counts == 0).any().any():
    missing = [(part, name) for part in counts.index for name in counts.columns
               if counts.loc[part, name] == 0]
    raise ValueError(f'Missing manually labeled examples for: {missing}. '
                     'Add labels for both classes in train, val and test, then rerun this cell.
'
                     'A binary experiment needs examples of both classes in all splits.')

class_to_idx = {name: i for i, name in enumerate(CLASS_NAMES)}
clip_table['y'] = clip_table['label'].map(class_to_idx)  # calm=0, action=1
clip_table.to_csv(OUT_DIR / 'clip_table_labeled.csv', index=False)
print('Labels:', class_to_idx)
print('Saved joined metadata:', OUT_DIR / 'clip_table_labeled.csv')
```

Cell 20

```
# Dataset and loader definitions. Materialize float32 only for each mini-batch.
class ClipDataset(Dataset):
    def __init__(self, all_clips, frame_table, split, augment=False):
        self.clips = all_clips
        selected = frame_table[(frame_table['split'] == split) & frame_table['y'].notna()]
        self.indices = selected.index.to_numpy(dtype=np.int64)
        self.labels = selected['y'].to_numpy(dtype=np.int64)
        self.augment = bool(augment)

    def __len__(self):
        return len(self.indices)

    def __getitem__(self, pos):
        arr = self.clips[self.indices[pos]].transpose(3, 0, 1, 2).copy()  # C,T,H,W
        x = torch.from_numpy(arr).float() / 255.0
        if self.augment and random.random() < 0.5:
            x = x.flip(-1)  # horizontal flip only in training
        y = int(self.labels[pos])
        return x, torch.tensor(y, dtype=torch.long)

train_ds = ClipDataset(clips, clip_table, 'train', augment=True)
val_ds = ClipDataset(clips, clip_table, 'val')
test_ds = ClipDataset(clips, clip_table, 'test')
train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True, num_workers=0)
val_loader = DataLoader(val_ds, batch_size=BATCH_SIZE, shuffle=False, num_workers=0)
test_loader = DataLoader(test_ds, batch_size=BATCH_SIZE, shuffle=False, num_workers=0)
```

```
example_x, example_y = train_ds[0]
print('One network input:', tuple(example_x.shape), '(C,T,H,W) | label:', int(example_y))
print('Dataset sizes:', len(train_ds), len(val_ds), len(test_ds))
assert tuple(example_x.shape) == (3, NUM_FRAMES, IMAGE_SIZE, IMAGE_SIZE)
assert example_x.min() >= 0 and example_x.max() <= 1
```

A1.2 Video recordings — Garmin application

Source file: TAI_Video_Speed_Estimation.ipynb. Notebook cells 13, 14, 15, 16, 18, 20, 21, 23, 24, 25, 26.

Cell 13

```
# Discover new MP4s. No previous dataset or old run folder is required.
inventory = inventory_videos(VIDEO_DIR)
print(f"Found {len(inventory)} videos")
display(inventory.head(10))
print(f"Total input size: {inventory['size_mb'].sum()/1024:.2f} GB")
```

Cell 14

```
# OPTIONAL: extract Garmin Track3 CSVs from the MP4 metadata.
# The helper will not overwrite existing CSVs. Verify RAW_SPEED_DIVISOR first.
if RUN_METADATA_EXTRACTION:
    run_script(
        "export_garmin_labels.py",
        "--video-dir", VIDEO_DIR,
        "--out-dir", SPEED_DIR,
        "--speed-divisor", RAW_SPEED_DIVISOR,
    )
else:
    print("Metadata export skipped. Expecting verified *_speed.csv files in", SPEED_DIR)
```

Cell 15

```
# Check CSV coverage and genuine stationary speeds.
label_audit = audit_speed_csvs(inventory, SPEED_DIR)
print("Label-file status:")
display(label_audit["status"].value_counts())
print("Valid GPS labels:", int(label_audit["valid_seconds"].sum()))
print("Valid stationary (0 km/h) labels:", int(label_audit["stationary_zeros"].sum()))
display(label_audit.head(15))
label_audit.to_csv(DATA_DIR / "label_audit.csv", index=False)
if (label_audit["status"] == "NO_CSV").all():
    raise ValueError("No matching CSV files. Check SPEED_DIR, names or metadata extraction.")
```

Cell 16

```
# Plot one representative GPS trace before turning it into training targets.
example_rows = label_audit[label_audit["status"] == "OK"]
if example_rows.empty:
    raise ValueError("All CSVs have missing labels. No supervised training is possible.")
example_csv = Path(example_rows.iloc[0]["csv"])
plot_gps_timeline(example_csv, title=f"Raw ground truth: {example_csv.stem}")
```

Cell 18

```
# Preview the exact crop/mask used by the flow-preparation script.
preview_video = Path(inventory.iloc[0]["file_path"])
preview_path = DATA_DIR / "crop_preview.png"
run_script("run.py", "preview",
           "--video", preview_video,
           "--out", preview_path,
           "--image-size", IMAGE_SIZE,
           "--mask-bottom", MASK_BOTTOM,
```

```
          "--flow-clip", FLOW_CLIP)
display(Image(filename=str(preview_path)))
print("CHECK: Is every displayed speed/GPS value hidden from this frame?")
```

Cell 20

```
# Create an editable template, without overwriting an existing mapping.
if not GROUPS_CSV.exists():
    run_script("run.py", "groups-template",
               "--video-dir", VIDEO_DIR, "--out", GROUPS_CSV)
else:
    print("Existing drive mapping found:", GROUPS_CSV)
print("Open this CSV, fill each drive_id, save it, then run the next cell.")
display(pd.read_csv(GROUPS_CSV, dtype=str, keep_default_na=False).head(12))
```

Cell 21

```
# Validate that all videos are assigned to coherent driving sessions.
# This is intentionally strict: an incomplete manual mapping should stop the process.
drive_map = inspect_drive_groups(GROUPS_CSV, inventory)
print("Number of independent driving sessions:", drive_map["drive_id"].nunique())
display(drive_map.groupby("drive_id").size().rename("video_count").to_frame())
if drive_map["drive_id"].nunique() < 2:
    raise ValueError("At least two independent drives are needed: train/val pool and test.")
```

Cell 23

```
# Prepare a new dataset. This uses valid 0 km/h and excludes N/A from loss.
manifest_path = PROCESSED / "manifest.csv"
if not manifest_path.is_file():
    run_script("run.py", "prepare",
               "--video-dir", VIDEO_DIR,
               "--speed-dir", SPEED_DIR,
               "--groups", GROUPS_CSV,
               "--out", PROCESSED,
               "--image-size", IMAGE_SIZE,
               "--mask-bottom", MASK_BOTTOM,
               "--flow-clip", FLOW_CLIP,
               "--zero-label-policy", "valid",
               "--min-valid-seconds", 1)
else:
    print("Using existing optical-flow dataset:", manifest_path)
    print("If videos, labels or preprocessing settings changed, create a NEW PROCESSED folder.")
```

Cell 24

```
# Inspect the generated manifest.
manifest = pd.read_csv(manifest_path, dtype={"video_id": str, "drive_id": str})
required_columns = {"clip_file", "video_id", "drive_id", "start_second",
                    "mean_speed_kmh", "valid_seconds"}
if not required_columns.issubset(manifest):
    raise ValueError("Manifest is missing required columns")
if manifest.empty:
    raise ValueError("No clips were prepared")
print("Prepared clips:", len(manifest))
print("Source MP4s:", manifest["video_id"].nunique())
print("Independent drives:", manifest["drive_id"].nunique())
print("Valid labeled seconds:", int(manifest["valid_seconds"].sum()))
display(manifest.head())
```

Cell 25

```
# Plot the distribution of mean speed per 10-second clip BEFORE splitting.
plt.figure(figsize=(10, 4))
plt.hist(manifest["mean_speed_kmh"], bins=np.arange(0, max(130,
         int(np.ceil(manifest["mean_speed_kmh"].max()/10)*10)+10), 10),
         edgecolor="black", alpha=0.75)
plt.xlabel("Mean speed within a 10-second clip (km/h)")
plt.ylabel("Number of clips")
plt.title("New dataset: clip-average speed distribution")
plt.grid(axis="y", alpha=0.2)
plt.tight_layout()
plt.show()
```

Cell 26

```
# Inspect one saved example, including shape and label-validity mask.
example_flow_file = PROCESSED / manifest.iloc[0]["clip_file"]
with np.load(example_flow_file, allow_pickle=False) as item:
    flow = np.asarray(item["flow"], dtype=np.float32)      # (50,2,H,W)
    speed = np.asarray(item["speed"], dtype=np.float32)    # (10,)
    valid_mask = np.asarray(item["valid_mask"], dtype=bool)
print("Optical flow:", flow.shape, "(50 frames, 2 directions, height, width)")
print("GPS targets:", speed.shape)
print("Label validity:", valid_mask.astype(int).tolist(), "(1 = measured)")
print("Valid speeds:", speed[valid_mask].round(1).tolist())

# An optical-flow magnitude image visualizes movement, not GPS speed.
flow_magnitude = np.sqrt(flow[5, 0] ** 2 + flow[5, 1] ** 2)
plt.figure(figsize=(5, 4))
plt.imshow(flow_magnitude, cmap="magma")
plt.colorbar(label="Optical flow magnitude (pixels / frame)")
plt.title("Example Farneback flow magnitude")
plt.axis("off")
plt.show()
```

A1.3 Still images — CIFAR-10 application

Source file: TAI_cifar10_classification.ipynb. Notebook cells 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34.

The application uses the configuration and preparation definitions reproduced in the A3 annex. The original notebook retains the complete execution order.

Cell 21

```
# Load the training and test images and look at them.
X_all, y_all, class_names, files_all, broken_train = load_images(os.path.join(DATA_DIR,
'train'), per_class=TRAIN_PER_CLASS)
X_test, y_test, test_names, files_test, broken_test = load_images(os.path.join(DATA_DIR,
'test'), per_class=TEST_PER_CLASS)
overview(X_all, y_all, class_names, title='training images')
```

Cell 22

```
# A few examples of every class.
show_grid(X_all, y_all, class_names, n=8)
```

Cell 23

```
# An image is a table of numbers. Change the position (0, 1, 2, ...) to look at another image.
position = 0
print(f"Image {position}, class: {class_names[y_all[position]]}")
show_pixels(X_all[position], size=6)
```

Cell 24

```
# The same image in different colour representations, and its histograms.
show_color_spaces(X_all[position])
plot_histogram(X_all[position])
```

Cell 25

```
# Average histograms of two classes. Try other pairs, for example 'Ship' and 'Truck'.
compare_class_histograms(X_all, y_all, class_names, 'Airplane', 'Frog')
```

Cell 26

```
# Check for corrupted files and duplicated images, also between the training and the test set.
n_broken = check_corrupt(broken_train + broken_test)
dup_train = check_duplicates(X_all)
in_both = check_overlap(X_all, X_test)
```

Cell 27

```
# Fix what was found: remove duplicated training images, and test images that are also in the
training set.
X_all, y_all, files_all = remove_images(X_all, y_all, files_all, dup_train, 'duplicated training
image(s)')
X_test, y_test, files_test = remove_images(X_test, y_test, files_test, in_both, 'test image(s)
that are also training images')
```

Cell 28

```
# Check for unusual images (too dark, too bright, low contrast) and look at the most extreme
ones.
outliers = check_outliers(X_all, threshold=3)
show_outliers(X_all, y_all, class_names, outliers, n=8)
```

Cell 29

```
# Choose the kinds of unusual images that should be removed.
out_widget = select_outliers(outliers)
```

Cell 30

```
# Remove the images of the selected kinds.
X_all, y_all, files_all = fix_outliers(X_all, y_all, files_all, outliers,
list(out_widget.value))
```

Cell 31

```
# Choose the classes that take part.
class_widget = select_classes(class_names)
```

Cell 32

```
# Keep the chosen classes (the same classes in the training and the test set) and show how
labels are encoded.
all_class_names = list(class_names)
X_all, y_all, files_all, class_names = keep_classes(X_all, y_all, files_all, all_class_names,
class_widget.value)
X_test, y_test, files_test, _ = keep_classes(X_test, y_test, files_test, all_class_names,
class_widget.value)
show_label_encoding(class_names)
```

Cell 33

```
# Try data augmentation on one image. Nothing is saved here: the augmentation level is a
hyperparameter in WP4-A3.
preview_augmentation(X_all[position])
```

Cell 34

```
# Split into training and validation sets, fit the scaling and save everything for WP4.
# The images are saved unscaled (0-255); the scaling is applied when they are loaded in WP4.
X_train, X_val, y_train, y_val = split_data(X_all, y_all, val_size=0.15)
scaler = fit_scaler(X_train, method='0-1')
print(f"Scaling method: '{scaler['method']}' (fitted on the training set).")

np.savez_compressed(os.path.join(OUT_DIR, 'prepared.npz'),
                    X_train=X_train, y_train=y_train, X_val=X_val, y_val=y_val, X_test=X_test,
y_test=y_test)
meta = {'class_names': class_names, 'img_size': IMG_SIZE, 'scaler': scaler, 'files_test':
files_test}
joblib.dump(meta, os.path.join(OUT_DIR, 'prepared_meta.joblib'))
print(f"Prepared data saved to '{OUT_DIR}/prepared.npz' and '{OUT_DIR}/prepared_meta.joblib'.")
```

A1.4 Audio/speech inputs — FSDD application

Source file: TAI_fsdd_audio_classification.ipynb. Notebook cells 8, 9, 10, 11, 12.

Cell 8

```
RECORDINGS_DIR = find_recordings(RECORDINGS_DIR)
audio_index = index_audio(RECORDINGS_DIR)
overview_audio(audio_index)
```

Cell 9

```
# The complete source player is independent of the model-input preview.
def audio_explorer(index):
    digit=widgets.Dropdown(options=list(CLASSES),description='Digit')
    speaker=widgets.Dropdown(options=sorted(index.speaker.unique()),description='Speaker')
    repeat=widgets.IntSlider(value=0,min=0,max=49,description='Repeat',continuous_update=False)
    full_label=widgets.HTML()
    full_player=widgets.Audio(value=b'',format='wav',autoplay=False,loop=False)
    output=widgets.Output()
    def show(change=None):

rows=index[(index.digit==digit.value)&(index.speaker==speaker.value)&(index.repeat==repeat.value
)]
        with output:
            clear_output(wait=True)
            if rows.empty:
                full_player.value=b'';full_label.value='No matching recording.';return
            row=rows.iloc[0];source=Path(row.path)
            full_player.value=source.read_bytes()
            duration=sf.info(source).duration
            full_label.value=f'<h3>1. FULL RECORDING — {duration:.2f}
s</h3><p>{source.name}</p>'
            preview_audio(source,CFG,row.file,include_full=False)
    for control in [digit,speaker,repeat]:control.observe(show,names='value')
    show()
    return widgets.VBox([widgets.HBox([digit,speaker,repeat]),full_label,full_player,
                         widgets.HTML('<hr><h3>2. AUDIO USED BY THE MODEL</h3>'),output])

display(audio_explorer(audio_index))
```

Cell 10

```
split_index = make_audio_splits(audio_index, SPLIT_MODE, TEST_SPEAKER, VAL_SPEAKER)
display(pd.crosstab(split_index['split'], split_index.digit))
display(pd.crosstab(split_index.speaker, split_index['split']))
```

Cell 11

```
# Feature extraction is done once and cached with the configuration and scalers.
prepared = prepare_audio_dataset(split_index, CFG)
# A separate experiment folder prevents mixing models from different splits/settings.
signature = hashlib.sha256((json.dumps(CFG, sort_keys=True) + SPLIT_MODE +
                            split_index[['file', 'split',
'sha256']].to_csv(index=False)).encode()).hexdigest()[:12]
prepared['signature'] = signature
RUN_DIR = OUT_DIR / signature
RUN_DIR.mkdir(exist_ok=True)
joblib.dump(prepared, RUN_DIR/'prepared.joblib')
(OUT_DIR/'active_run.txt').write_text(signature, encoding='utf-8')
```

```
prepared['index'].to_csv(RUN_DIR/'audio_index.csv', index=False)
print('Experiment:', signature)
print('Pooled ML/MLP:', prepared['pooled'].shape)
print('MFCC sequences:', prepared['sequence'].shape)
print('CNN log-mel input:', prepared['cnn'].shape)
if prepared['notes']:
    display(pd.DataFrame(prepared['notes']))
```

Cell 12

```
# Inspect the tabular representation used by classical ML and the MLP.
feature_table = pd.DataFrame(prepared['pooled'], columns=prepared['feature_names'])
display(feature_table.head().round(3))
print('X has', feature_table.shape[1], 'features per recording. y is a digit from 0 to 9.')
```

A1.5 Audio/speech inputs — Sopele application

Source file: TAI_sopele_audio_classification.ipynb. Notebook cells 8, 9, 10, 11, 12.

Cell 8

```
DATA_DIR=resolve_sopele_data(DATA_SOURCE)
source_index=split_sopele_sources(index_sopele_sources(DATA_DIR))
overview_sopele_sources(source_index)
```

Cell 9

```
def source_audio_explorer(index):
    selector=widgets.Dropdown(options=[(r.label+' | '+r.file,i) for i,r in index.iterrows()],
                              description='Recording',layout=widgets.Layout(width='95%'))
    position=widgets.FloatSlider(value=0,min=0,max=0,step=CFG['hop_seconds'],description='Start
(s)',continuous_update=False)
    full_label=widgets.HTML()
    full_player=widgets.Audio(value=b'',format='wav',autoplay=False,loop=False)
    output=widgets.Output()
    def show(change=None):
        with output:
            clear_output(wait=True)
            row=index.loc[selector.value];wave=load_sopele_audio(row.path,CFG)
            windows,starts,rms,active=make_sopele_windows(wave,CFG)
            selected=np.flatnonzero(active)
            limit=CFG['max_windows_per_file']
            if limit and len(selected)>limit:
                selected=selected[np.linspace(0,len(selected)-1,limit,dtype=int)]
            if not len(selected):
                print('No active model windows in this recording.');return
            available=starts[selected]/CFG['sample_rate']
            chosen=int(np.argmin(np.abs(available-position.value)))
            i=selected[chosen];start_seconds=float(available[chosen])
            print(f'Source partition: {row.split} | window {chosen+1}/{len(selected)}')
            preview_sopele_wave(windows[i],CFG,row.label+' | '+row.file,
                                start_seconds=start_seconds)
    def select_recording(change=None):
        row=index.loc[selector.value];wave=load_sopele_audio(row.path,CFG)
        original=Path(row.path)
        full_player.value=original.read_bytes()
        duration=sf.info(original).duration
        full_label.value=f'<h3>1. FULL RECORDING — {duration:.2f} s</h3><p>{original.name} |
This player is independent of the slider.</p>'
        position.unobserve(show,names='value')
        position.value=0
        position.max=max(0.0,(len(wave)-
round(CFG['window_seconds']*CFG['sample_rate']))/CFG['sample_rate'])
        position.observe(show,names='value');show()

selector.observe(select_recording,names='value');position.observe(show,names='value');select_rec
ording()
    return widgets.VBox([selector,full_label,full_player,
                         widgets.HTML('<hr><h3>2. MODEL INPUT WINDOW</h3><p>The slider selects
only the short window below.</p>'),position,output])

display(source_audio_explorer(source_index))
```

Cell 10

```
print('Task:',TASK,'| ML/CNN representation:',FEATURE_MODE)
display(pd.crosstab(source_index['split'],source_index.label))
print('Splitting by original recording prevents related windows crossing partitions.')
```

Cell 11

```
prepared=prepare_sopele_dataset(source_index,CFG,TASK,FEATURE_MODE)
CLASSES=np.arange(len(prepared['labels']))
signature=hashlib.sha256((json.dumps(CFG,sort_keys=True)+TASK+FEATURE_MODE+str(SEED)+

source_index[['source_id','split','sha256']].to_csv(index=False)).encode()).hexdigest()[:12]
prepared['signature']=signature
RUN_DIR=OUT_DIR/signature;RUN_DIR.mkdir(exist_ok=True)
joblib.dump(prepared,RUN_DIR/'prepared.joblib')
(OUT_DIR/'active_run.txt').write_text(signature,encoding='utf-8')
prepared['index'].to_csv(RUN_DIR/'window_index.csv',index=False)
source_index.to_csv(RUN_DIR/'source_split.csv',index=False)
display(pd.crosstab(prepared['index']['split'],prepared['index'].label))
print('Experiment:',signature)
for name in ['pooled','cnn','sequence']:print(name,prepared[name].shape)
```

Cell 12

```
display(pd.DataFrame({'Class index':CLASSES,'Tone label':prepared['labels']}))
display(pd.DataFrame(prepared['pooled'][:5],columns=prepared['feature_names']).iloc[:,:15].round
(4))
print('Each example is a short fixed-length audio window. The class index encodes its tone
label.')
```

A1.6 Numerical and industrial data — Concrete compressive strength

Source file: TAI_concrete_regression.ipynb. Notebook cells 2, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29.

Cell 2

```
# Imports and basic settings. Nothing is computed here.
import os, warnings, itertools
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import joblib
from IPython.display import display
import ipywidgets as widgets
from ipywidgets import interact

warnings.filterwarnings('ignore')
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 between xgboost and
tensorflow on macOS
pd.set_option('display.width', 140)
pd.set_option('display.max_columns', 30)

SEED = 42
DATA_FILE = 'Concrete_Data.csv'
OUT_DIR = 'output_concrete'
MODEL_DIR = os.path.join(OUT_DIR, 'models')
os.makedirs(MODEL_DIR, exist_ok=True)
```

Cell 19

```
# Load the data and look at it.
df = load_data(DATA_FILE)
overview(df)
```

Cell 20

```
# Check for missing values and duplicates.
missing_table = check_missing(df)
n_dup = check_duplicates(df)
```

Cell 21

```
# Fix what was found. Missing values: 'mean', 'median' or 'drop'. Duplicated rows are removed.
df = fix_missing(df, method='mean')
df = fix_duplicates(df)
```

Cell 22

```
# Check for outliers (values outside +-3 standard deviations) and look at the most extreme rows.
outliers = check_outliers(df, threshold=3)
show_outliers(df, outliers, n=5)
```

Cell 23

```
# Choose the columns whose outlier rows should be removed.
out_widget = select_outliers(outliers)
```

Cell 24

```
# Remove the outlier rows of the selected columns.
df = fix_outliers(df, outliers, list(out_widget.value))
```

Cell 25

```
# Choose the input and output columns.
io = select_io(df, default_target='Strength')
```

Cell 26

```
# Read the choice.
input_cols, target_col = get_io(io)
X = df[input_cols].copy()
y = df[target_col].copy()
```

Cell 27

```
# Check which input columns are categorical. 'Age' has 14 distinct values and stays numeric.
cat_widget = select_categorical(X, max_unique=10)
```

Cell 28

```
# Replace categorical columns with n-1 dummy columns.
categorical_cols = list(cat_widget.value)
X = make_dummies(X, categorical_cols)
feature_names = list(X.columns)
```

Cell 29

```
# Split 80/20 with shuffle, scale inputs and output to 0-1, and save everything for WP4.
X_train, X_test, y_train, y_test = split_data(X, y, test_size=0.2, shuffle=True)
# Preserve original input names, category levels and training ranges for new predictions.
# Capture this BEFORE the arrays are scaled.
raw_inputs = df[input_cols].copy()
inference_metadata = {
    'input_cols': list(input_cols),
    'categorical_cols': list(categorical_cols),
    'category_levels': {c: sorted(raw_inputs[c].dropna().unique().tolist())
                        for c in categorical_cols},
    'input_reference': raw_inputs.loc[X_train.index].copy(),
    'target_reference': df.loc[X_train.index, target_col].copy(),
    'history_example': df[input_cols + [target_col]].tail(24).copy(),
}
X_train, X_test, y_train, y_test, x_scaler, y_scaler = scale_data(X_train, X_test, y_train,
y_test, scale_target=True)

prep = {'X_train': X_train, 'X_test': X_test, 'y_train': y_train, 'y_test': y_test,
        'x_scaler': x_scaler, 'y_scaler': y_scaler, 'feature_names': feature_names, 'target':
target_col}
prep.update(inference_metadata)
joblib.dump(prep, os.path.join(OUT_DIR, 'prepared.joblib'))
```

```
print(f"Prepared data saved to '{OUT_DIR}/prepared.joblib'.")
```

A1.7 Numerical and industrial data — Meteorological and air-quality time series

Source file: TAI_meteo_timeseries.ipynb. Notebook cells 2, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29.

Cell 2

```
# Imports and basic settings. Nothing is computed here.
import os, warnings, itertools
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import joblib
from IPython.display import display
import ipywidgets as widgets
from ipywidgets import interact

warnings.filterwarnings('ignore')
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 between xgboost and
tensorflow on macOS
pd.set_option('display.width', 140)
pd.set_option('display.max_columns', 30)

SEED = 42
DATA_FILE = 'data_thesis_4_1.csv'
OUT_DIR = 'output_meteo'
MODEL_DIR = os.path.join(OUT_DIR, 'models')
os.makedirs(MODEL_DIR, exist_ok=True)
```

Cell 19

```
# Load the data and look at it.
df = load_data(DATA_FILE)
overview(df)
```

Cell 20

```
# Check for missing values and duplicates.
missing_table = check_missing(df)
n_dup = check_duplicates(df)
```

Cell 21

```
# Fix what was found. Missing values: 'mean', 'median' or 'drop'. Duplicated rows are removed.
df = fix_missing(df, method='mean')
df = fix_duplicates(df)
```

Cell 22

```
# Check for outliers (values outside +-3 standard deviations) and look at the most extreme rows.
outliers = check_outliers(df, threshold=3)
show_outliers(df, outliers, n=5)
```

Cell 23

```
# Choose the columns whose outlier rows should be removed.
# For a time series removing rows breaks the hourly sequence, so the usual choice here is to
leave the selection empty.
# High PM and precipitation values are real episodes, not errors.
out_widget = select_outliers(outliers)
```

Cell 24

```
# Remove the outlier rows of the selected columns.
df = fix_outliers(df, outliers, list(out_widget.value))
```

Cell 25

```
# Choose the input and output columns. PM10 is left out of the inputs by default.
target = [c for c in df.columns if 'PM2.5' in c][0]
pm10 = [c for c in df.columns if 'PM10' in c][0]
io = select_io(df, default_target=target, default_inputs=[c for c in df.columns if c not in
(target, pm10)])
```

Cell 26

```
# Read the choice.
input_cols, target_col = get_io(io)
X = df[input_cols].copy()
y = df[target_col].copy()
```

Cell 27

```
# Check which input columns are categorical. 'Temperature inversion' is 0/1 and can stay as it
is.
cat_widget = select_categorical(X, max_unique=10)
```

Cell 28

```
# Replace categorical columns with n-1 dummy columns.
categorical_cols = list(cat_widget.value)
X = make_dummies(X, categorical_cols)
feature_names = list(X.columns)
```

Cell 29

```
# Chronological split 80/20 without shuffle, scale inputs and output to 0-1, and save everything
for WP4.
# Rows stay in time order, which is needed for building the lookback windows in WP4.
X_train, X_test, y_train, y_test = split_data(X, y, test_size=0.2, shuffle=False)
# Preserve original input names, category levels and training ranges for new predictions.
# Capture this BEFORE the arrays are scaled.
raw_inputs = df[input_cols].copy()
inference_metadata = {
    'input_cols': list(input_cols),
    'categorical_cols': list(categorical_cols),
    'category_levels': {c: sorted(raw_inputs[c].dropna().unique().tolist())
                        for c in categorical_cols},
    'input_reference': raw_inputs.loc[X_train.index].copy(),
    'target_reference': df.loc[X_train.index, target_col].copy(),
    'history_example': df[input_cols + [target_col]].tail(24).copy(),
```

```
}
X_train, X_test, y_train, y_test, x_scaler, y_scaler = scale_data(X_train, X_test, y_train,
y_test, scale_target=True)

prep = {'X_train': X_train, 'X_test': X_test, 'y_train': y_train, 'y_test': y_test,
        'x_scaler': x_scaler, 'y_scaler': y_scaler, 'feature_names': feature_names, 'target':
target_col}
prep.update(inference_metadata)
joblib.dump(prep, os.path.join(OUT_DIR, 'prepared.joblib'))
print(f"Prepared data saved to '{OUT_DIR}/prepared.joblib'.")
```

A1.8 Numerical and industrial data — Loan classification

Source file: TAI_loan_classification.ipynb. Notebook cells 2, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29.

Cell 2

```
# Imports and basic settings. Nothing is computed here.
import os, warnings, itertools
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import joblib
from IPython.display import display
import ipywidgets as widgets
from ipywidgets import interact

warnings.filterwarnings('ignore')
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 between xgboost and
tensorflow on macOS
pd.set_option('display.width', 140)
pd.set_option('display.max_columns', 30)

SEED = 42
DATA_FILE = 'loan_data.csv'
OUT_DIR = 'output_loan'
MODEL_DIR = os.path.join(OUT_DIR, 'models')
os.makedirs(MODEL_DIR, exist_ok=True)
```

Cell 19

```
# Load the data and look at it.
df = load_data(DATA_FILE)
overview(df)
```

Cell 20

```
# Check for missing values and duplicates.
missing_table = check_missing(df)
n_dup = check_duplicates(df)
```

Cell 21

```
# Fix what was found. Missing values: 'mean', 'median' or 'drop'. Duplicated rows are removed.
df = fix_missing(df, method='mean')
df = fix_duplicates(df)
```

Cell 22

```
# Check for outliers (values outside +-3 standard deviations) and look at the most extreme rows.
outliers = check_outliers(df, threshold=3)
show_outliers(df, outliers, n=5)
```

Cell 23

```
# Choose the columns whose outlier rows should be removed.
out_widget = select_outliers(outliers)
```

Cell 24

```
# Remove the outlier rows of the selected columns.
df = fix_outliers(df, outliers, list(out_widget.value))
```

Cell 25

```
# Choose the input and output columns.
io = select_io(df, default_target='loan_status')
```

Cell 26

```
# Read the choice and show the class balance.
input_cols, target_col = get_io(io)
X = df[input_cols].copy()
y = df[target_col].copy()
print(f"Class balance of '{target_col}':")
print(y.value_counts(normalize=True).round(3).to_string())
```

Cell 27

```
# Check which input columns are categorical. The five text columns are pre-selected.
cat_widget = select_categorical(X, max_unique=10)
```

Cell 28

```
# Replace categorical columns with n-1 dummy columns.
categorical_cols = list(cat_widget.value)
X = make_dummies(X, categorical_cols)
feature_names = list(X.columns)
```

Cell 29

```
# Split 80/20 with shuffle, scale inputs to 0-1 (the output is already 0/1), and save everything
for WP4.
X_train, X_test, y_train, y_test = split_data(X, y, test_size=0.2, shuffle=True)
# Preserve original input names, category levels and training ranges for new predictions.
# Capture this BEFORE the arrays are scaled.
raw_inputs = df[input_cols].copy()
inference_metadata = {
    'input_cols': list(input_cols),
    'categorical_cols': list(categorical_cols),
    'category_levels': {c: sorted(raw_inputs[c].dropna().unique().tolist())
                        for c in categorical_cols},
    'input_reference': raw_inputs.loc[X_train.index].copy(),
    'target_reference': df.loc[X_train.index, target_col].copy(),
    'history_example': df[input_cols + [target_col]].tail(24).copy(),
}
X_train, X_test, y_train, y_test, x_scaler, _ = scale_data(X_train, X_test, y_train, y_test,
scale_target=False)

prep = {'X_train': X_train, 'X_test': X_test, 'y_train': y_train, 'y_test': y_test,
        'x_scaler': x_scaler, 'feature_names': feature_names, 'target': target_col}
```

```
prep.update(inference_metadata)
joblib.dump(prep, os.path.join(OUT_DIR, 'prepared.joblib'))
print(f"Prepared data saved to '{OUT_DIR}/prepared.joblib'.")
```