Univ.-Prof. Dr.-Ing. Matthias Boehm
Technische Universität Berlin
Faculty IV - Electrical Engineering and Computer Science
Berlin Institute for the Foundations of Learning and Data (BIFOLD)
Big Data Engineering (DAMS Lab) Group

1 AMLS SoSe 2026: Exercise – AI Image Detection

Published: Apr 12, 2026 (last update: Apr 16)
Deadline: Jul 15, 2026; 11.59pm

This exercise is an alternative to the AMLS programming projects and aims to provide practical experience in the exploratory development of machine learning (ML) pipelines. The task is to create an ML pipeline to train a model for AI image detection. For this exercise, you should implement your solution in Python and may utilize existing open-source ML systems and libraries. The expected result is a zip archive named `AMLS_Exercise_<student_ID>.zip` of max 20 MB, containing:

- A self-contained solution folder containing the source code, a Docker container configuration, and all dependency manifests required to build and execute the solution without internet access at runtime. At minimum, this should include a `Dockerfile` to build a Docker image (the built image must **not** be included in the zip file). Use CPU-only versions of ML frameworks; GPU-enabled builds are not required and may lead to excessively large Docker images. The final image size should not exceed 4 GB. An example `Dockerfile` can be found in Appendix A.
- A PDF report of up to 8 pages (10pt), including the names of all team members and an explanation of the solutions to the individual sub-tasks.

Data: This exercise uses a labeled dataset of real and AI-generated images[1]. Figure 1 shows example images for each class. The main task is binary classification with labels `0: real` and `1: ai_generated`. For this exercise, labels `1` to `5` are merged into the single class `1: ai_generated`.

0: Real | 1: SD 2.1 | 2: SDXL | 3: SD 3 | 4: DALL-E 3 | 5: Midjourney

Figure 1: Example images of the dataset.

Grading: This exercise is be pursued in teams of 1 to 3 persons (one submission). The grading is a *pass/fail* for the entire team. Exercises with $\geq 50/100$ points are a pass, and the quality expectations increase with the team size. Exercises with ≥ 90 points receive 5 extra points in the exam.

[1]Derived from the Defactify dataset and the MS COCO dataset (CC BY 4.0). This modified version is provided for educational use in this course.

Dataset structure. The dataset can be downloaded from TU-Cloud and has the following structure:

```
data/
|-- train/
|-- calibration/
|-- calibration_augmented/
|-- validation/
|-- validation_augmented/
`-- predict/
```

The labeled splits `train/`, `calibration/`, `calibration_augmented/`, `validation/`, and `validation_augmented/` contain parquet files with columns `image: binary` and `source_class: int8`. The `predict/` split contains unlabeled parquet files with columns `row_id: int32` and `image: binary`.

Data splits. You are provided with a **train** split, a **calibration** split, and a **validation** split. A separate unlabeled holdout (**predict**) is given to test inference in your ML pipeline.

Submission Guidelines

Your submission must be designed such that it can be executed automatically inside a Docker container without internet access at runtime. You should therefore submit a self-contained `solution/` folder with your code, a `Dockerfile`, and the dependency manifests required to build the container. During grading, we will place the data directly into your submitted solution folder. Your submission should have the following structure:

```
report.pdf
<other-files> (e.g. code for task 1.4)
solution/
|-- Dockerfile
|-- clean.py
|-- prepare.py
|-- train.py
|-- train_augmented.py
|-- predict.py
|-- predict_augmented.py
|----------------------- AT RUNTIME (DO NOT SUBMIT) -----------------------
|-- artifacts/ <--- Write folder for cleaned/prepared data, trained models
|   |-- task02/predictions.csv
|   `-- task03/predictions.csv
`-- data/**    <--- Read-only mount during execution (see dataset structure)
```

The contents of `solution/data/` will be identical to the provided download except the contents of `solution/data/predict/`, which will be used to evaluate your models. During execution, the `solution/data/` directory is mounted into the container as read-only input. Your code must therefore treat this directory as read-only and write all derived files, caches, models, and predictions only under `solution/artifacts/`. The script execution order is:

```
1. python clean.py             --timeout_seconds 600
2. python prepare.py           --timeout_seconds 600
3. python train.py             --timeout_seconds 1800
4. python predict.py           --timeout_seconds 600
5. python train_augmented.py   --timeout_seconds 1800
6. python predict_augmented.py --timeout_seconds 600
```

Note that `prepare.py` should not prepare data from `solution/data/predict/` as it may change after training. Keep in mind that each script is terminated after a timeout that is given as CLI input. Hence, it makes sense to regularly write the best model checkpoint into `solution/artifacts/`.

1.1 Dataset Exploration and Cleaning (15/100 points)

As a starting point, analyze the provided training data. Summarize the class distribution, the image-size distribution, and basic descriptive statistics. In particular, report any characteristics that could be used to deduce which class an image belongs to. After analyzing the data characteristics, construct a deterministic cleaning pipeline for the training data. The resulting dataset should be suitable for downstream CPU-friendly modeling, but you must justify your preprocessing choices rather than simply applying a fixed recipe. This task is about exploration and cleaning, not augmentation.

Expected Results: Runnable code `solution/clean.py` for data exploration and cleaning, a cleaned training dataset or a script that regenerates it, and a short report summarizing and visualizing your findings as well as your chosen deterministic preprocessing pipeline.

1.2 Modeling and Tuning under Time Constraints (35/100 points)

Construct an ML pipeline—using the cleaned training data from Task 1 and the provided validation set—for classifying images as `real` or `ai_generated`. The number of false accusations should be limited, so your goal is to maximize `recall_ai` while ensuring that the false-positive rate on real images is at most 20% (which should be independently validated on the validation set). To calibrate your model for the target false-positive rate, you may use the calibration data in `data/calibration/`. It is advised to automatically calibrate your model as part of the pipeline and not to rely on manually determined thresholds. Make sure to hold out the validation data to verify the achieved false-positive rate. This task also emphasizes efficient CPU retraining: your solution should be designed such that it can be trained reproducibly within a strict time budget. To make local development less dependent on hardware differences, Appendix C provides a small in-memory reference CNN training script on synthetic data. Your own local training runtime should stay within at most 5× the reference runtime on your machine (evaluation is performed in a Docker container with `--cpus 8`).

A reasonable way to approach this task is to experiment with different model architectures, loss functions, optimizers, hyperparameters, as well as the operating threshold. You should report all meaningful evaluation metrics (e.g. false-positive rate, recall, ...) as well as the impact of individual modeling choices in the PDF. Also, report model performance on `data/validation_augmented`.

This task must be solved on CPU only, without internet access during runtime, and in a Docker-compatible, fully reproducible way. In particular, no pretrained models may be downloaded at runtime.

You should train and compare at least two different model families, for example a classical baseline on engineered features and a neural model trained from scratch on the cleaned images. The report should document these comparisons clearly. For the code submission, however, you only need to package the single best Task 2 pipeline in `prepare.py`, `train.py` and `predict.py`. Label prediction in `predict.py` should produce a file `artifacts/task02/predictions.csv` in the following format:

```
row_id,predicted_label
0,1
1,0
2,1
```

Appendix B provides a reference CNN implementation that you may use as a starting point, but you are free to construct your own models. You should tune hyperparameters and justify your choices in the report. Your final model should aim for at least 0.8 `recall_ai` while strictly respecting the 20% false-positive constraint on the validation set. Strong solutions are expected to push this value even higher while still remaining within the required false-positive limit. The model performance will be evaluated on a hidden holdout, so make sure not to overfit (also with regard to the false-positive rate).

Expected Results: Runnable preparation, training and inference code (`solution/prepare.py`, `solution/train.py` and `solution/predict.py`) and a documented validation protocol.

1.3 Data Augmentation and Feature Engineering (30/100 points)

The goal of this task is to improve the detector's robustness. In practice, models that perform well during training are often too fragile in real-world settings. Images may be scaled, compressed, blurred, or subjected to other transformations. Your task is to apply augmentation techniques to make your model more robust to such modifications. You may either train a new model from scratch or build upon a model checkpoint obtained in Task 2.

The same overall requirements from Task 2 still hold. In particular, you must stay within the same strict CPU retraining budget, and your final operating point must still satisfy the same threshold-based false-positive constraint on real images.

For this task, we provided additional calibration and validation data (`data/calibration_augmented`, `data/validation_augmented`) that represent slightly modified images. You should use these data to analyze the effect of your robustness strategy.

Report your overall approach, justify the chosen augmentation strategy, and compare model performance and robustness to your Task 2 model on both `validation/` and `validation_augmented/` in the PDF.

Expected Results: Code for the selected augmentation or feature-engineering techniques (`solution/train_augmented.py`, `solution/predict_augmented.py`), a robust model obtained by continuing from the Task 2 starting point, and a comparison against the Task 2 model on both `data/validation/` and `data/validation_augmented/` under the same CPU-time budget and threshold constraint. Your final Task 3 model should aim for at least 0.6 `recall_ai` on `data/validation_augmented/` while still respecting the 20% false-positive constraint on real images. Strong solutions are expected to improve robustness beyond this level and ideally maintain competitive performance on `data/validation/` as well. The model performance will be evaluated on a hidden holdout. The inference script `solution/predict_augmented.py` must produce a file `artifacts/task03/predictions.csv` in the same format as Task 2.

1.4 Explainability (20/100 points)

Finally, analyze and explain the behavior of your final model. The goal of this task is not only to report a score, but to reason about *why* the model makes its decisions and where it fails.

Reasonable directions include:

- saliency maps or gradient-based explanations,
- occlusion or perturbation analysis,
- analysis of false positives and false negatives,
- comparison of what the model attends to for real vs. AI-generated images.

You should use the explainability method in a critical way. Explanations should not be treated as automatically correct; rather, you should discuss whether the explanation appears plausible and whether it reveals remaining shortcut behavior or dataset bias.

Expected Results: Code for the selected explainability method, visual or quantitative examples of model explanations, and a discussion of what these explanations reveal about the model's behavior, strengths, and failure modes.

Appendix A - Dockerfile

An example Docker setup could look as follows:

```
FROM python:3.11-slim

WORKDIR /workspace/solution

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

COPY requirements.txt /workspace/solution/requirements.txt
RUN pip install --no-cache-dir -r /workspace/solution/requirements.txt
RUN pip install --no-cache-dir --index-url https://download.pytorch.org/whl/cpu \
    torch==2.5.1

COPY . /workspace/solution
```

Appendix B - Reference CNN

A minimal CNN baseline for binary image classification could look as follows. This implementation is intentionally simple and should be understood as a starting point rather than a recommended final solution.

```
import torch.nn as nn

features = nn.Sequential(
    nn.Conv2d(3, k, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    nn.Conv2d(k, 2 * k, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),
    nn.Conv2d(2 * k, 4 * k, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.AdaptiveAvgPool2d(1),
)

classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(4 * k, 2),
)
```

Appendix C - Train Time Reference

The following script provides a lightweight in-memory timing reference. It does *not* read parquet files and does *not* perform any realistic image preprocessing. Its purpose is only to give a rough local baseline for model-training throughput.

```
import os
import time
import torch
import torch.nn as nn

torch.manual_seed(0)
torch.set_num_threads(min(8, os.cpu_count() or 1)) # max 8 threads
torch.set_num_interop_threads(1)
k = 32
model = nn.Sequential(
    nn.Conv2d(3, k, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.Conv2d(k, k, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),

    nn.Conv2d(k, 2 * k, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.Conv2d(2 * k, 2 * k, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),

    nn.Conv2d(2 * k, 4 * k, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.Conv2d(4 * k, 4 * k, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.AdaptiveAvgPool2d(1),

    nn.Flatten(),
    nn.Linear(4 * k, 2),
)

x = torch.randn(128, 3, 224, 224)
y = torch.randint(0, 2, (128,))
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()

def train_steps(steps):
    for _ in range(steps):
        optimizer.zero_grad(set_to_none=True)
        logits = model(x)
        loss = criterion(logits, y)
        loss.backward()
        optimizer.step()

train_steps(10) # warmup
start = time.perf_counter()
train_steps(35)
elapsed = time.perf_counter() - start
print(f"elapsed_seconds={elapsed:.3f}")
```