Université Paris Cité

DVC pipeline for phoneme ASR robustness to noise

Guillaume Wisniewski
guillaume.wisniewski@u-paris.fr

February 2026

The goal of this lab is to build a fully reproducible DVC pipeline that : (i) turns text references into phoneme references with `espeak-ng`, (ii) creates several noisy versions of the audio at different SNR levels, (iii) runs a pre-trained phoneme recogniser (`wav2vec2phonemes`), (iv) evaluates performance (e.g., PER), (v) plots performance vs noise for each language and the cross-language mean.

The aim of working with phonemic transcription is twofold : (i) to show you that there is real value in understanding what a phoneme is (something which, as a supposedly competent NLP practitioner, I only learned in a first approximation last week, and which I very much regret not having encountered much earlier) ; (ii) to ensure that you work with a single model that operates across languages, rather than one model per language, thereby making the results more directly comparable and your life easier.

Key concept : the manifest

In this lab, a **manifest** is a line-based JSON file (`.jsonl`) where each line describes one utterance. All stages *must* consume and produce manifests : scripts must not "scan folders" to discover data but rely solely on manifests.

Your manifest must contain (at least, you can of course add additional fields as needed) the following fields :

— `utt_id` : a stable, deterministic utterance identifier.
— `lang` : language code (e.g., `en`, `fr`).
— `wav_path` : relative path to the audio file.
— `ref_text` : reference transcript (raw text).
— `ref_phon` : reference phoneme sequence (produced by `espeak-ng`).
— `audio_md5` : checksum to ensure traceability

Your `utt_id` must be invariant across clean/noisy variants of the same utterance. One robust choice is :

— `utt_id = {lang}_{stem}` where `stem` is the wav filename without extension ;
— or `utt_id = {lang}_{md5(relative_wav_path)}` (stable as long as paths are stable).

Here is an example manifest (`data/manifests/en/clean.jsonl`) :

```
{"utt_id":"en_commonvoice_000001",\
 "lang":"en",\
 "wav_path":"data/raw/en/wav/commonvoice_000001.wav",\
 "ref_text":"I can hardly believe it.",\
 "ref_phon":"a  kæn h dli b li v  t",\
 "sr":16000,\
 "duration_s":1.92,\
 "snr_db":null}
{"utt_id":"en_commonvoice_000002",\
 "lang":"en",\
 "wav_path":"data/raw/en/wav/commonvoice_000002.wav",\
 "ref_text":"Please call me tomorrow.",\
 "ref_phon":"pli z k l mi tə m rə ",\
 "sr":16000,\
 "duration_s":2.14,\
 "snr_db":null}
```

Recall that the \ symbols indicate that the newline character is used only for readability and should not be considered. In fact, the previous example contains only two lines, each line being a (valid) JSON object. Using this line-based format instead of standard JSON allows the file to be streamed. Each recording can be read and processed one at a time, without loading the whole file into memory, as would be required if all recordings were stored in a single list.

The manifest plays a central role in ensuring the correctness and reproducibility of the pipeline. In particular, it should be created atomically : a manifest must only be written once all items it describes have been successfully produced and validated. In practice, this means generating the manifest in a temporary file and renaming it to its final location only after the whole process has completed without errors. This simple strategy makes the manifest an explicit indicator of successful execution : if a manifest exists, the corresponding stage is guaranteed to have run to completion ; if it does not, the stage must be considered incomplete or failed. Atomic manifest creation therefore provides a robust and transparent mechanism to detect partial runs and to prevent downstream stages from silently consuming corrupted or incomplete data.

Adding noise to a `wav` file

To help you with your arduous task, I generously provide the code for two functions that make it possible to add noise to an audio recording. Like any self-respecting

computer scientist, I have of course not tested this code, and it is therefore possible (understand : likely) that it does not do exactly what it claims to do. I nevertheless sincerely hope that you are convinced that it will save them more time than they will lose integrating it (cultural note : you are facing a classic problem in computer science often referred to as the "not invented here" syndrome, which suggests that only code one has written oneself is both correct and truly understandable).

```python
import numpy as np
import soundfile as sf


def add_noise(
    signal: np.ndarray,
    snr_db: float,
    rng: np.random.Generator,
) -> np.ndarray:
    signal_power = np.mean(signal ** 2)
    snr_linear = 10 ** (snr_db / 10)
    noise_power = signal_power / snr_linear

    noise = rng.normal(
        loc=0.0,
        scale=np.sqrt(noise_power),
        size=signal.shape,
    )
    return signal + noise


def add_noise_to_file(
    input_wav: str,
    output_wav: str,
    snr_db: float,
    seed: int | None = None,
) -> None:
    signal, sr = sf.read(input_wav)
    if signal.ndim != 1:
        raise ValueError("Only mono audio is supported")

    rng = np.random.default_rng(seed)
    noisy_signal = add_noise(signal, snr_db, rng)

    sf.write(output_wav, noisy_signal, sr)
```

The implementation is deliberately split into two functions with distinct roles. The

core signal processing is handled by a pure function (`add_noise`), which takes an input signal, an SNR value, and an explicit random number generator, and returns a noisy signal without performing any input/output operations or modifying external state. This separation is crucial for testing : a pure function can be unit-tested in isolation using synthetic signals, fixed random generators, and precise numerical assertions. The second function (`add_noise_to_file`) is responsible solely for file I/O and delegates all signal manipulation to the pure function. This design makes the code easier to test, reason about, and maintain, and illustrates a general principle that will be revisited in a later course when discussing testing, reproducibility, and functional design in data processing pipelines.

The `add_noise` function first reads the waveform and estimates its average power, then generates white Gaussian noise whose power is scaled so as to achieve the requested SNR (signal-to-noise ratio, i.e., the "amount" of noise relative to the signal). The noisy signal is obtained by adding this noise directly to the original waveform and returning the result. Crucially, the function optionally accepts a random seed, which is used to initialise the pseudo-random number generator. Fixing this seed is essential for reproducibility : it guarantees that the same input file and SNR value always produce exactly the same noisy output, ensuring that changes observed in downstream stages reflect genuine experimental modifications rather than stochastic variation.

Inference and evaluation

As you have all followed with great attention the magnificent (at least !) course on LLMs, using the HuggingFace 🤗 library for running inference with a pre-trained model should be child's play for you. In this lab, you must generate predictions using the model with signature `facebook/wav2vec2-lv-60-espeak-cv-ft`. The model takes raw audio waveforms as input and outputs sequences of phoneme tokens, which must then be compared to the reference phonemic transcriptions provided in the manifests. Be careful : all models that operate on audio impose strict constraints on the audio format, in particular with respect to the sampling rate and whether the signal is mono or multi-channel. As always, the prediction step must be implemented as a clearly identified stage of the pipeline, consuming manifests as input and producing prediction manifests as output.

The evaluation must be carried out using the Phoneme Error Rate (PER). PER is defined as the normalised edit distance between the predicted and reference phoneme sequences, counting substitutions (S), deletions (D), and insertions (I), and is given by :

$$\text{PER} = \frac{S + D + I}{N}, \tag{1}$$

where N denotes the number of phonemes in the reference sequence. Note that this measure is strictly equivalent to the Character Error Rate (CER) when each phoneme is

represented by a single character. This observation provides a useful intuition and allows you to reuse standard string edit-distance tools without any conceptual modification.

What you must do !

The (description of the) task is deliberately simple : you are expected to implement the experiment I have just described, with roughly the same level of precision as an Impressionist painter. In practical terms, this means being able to generate a performance curve as a function of noise (consider around ten noise levels) for a given language.

I am fully aware that, in my usual devious fashion, I only outline the main steps of what you are expected to do, without providing the exact commands to run or describing in detail what should happen at each stage. The aim (which I hope you will all find entirely commendable) is to encourage you to carry out a small amount of investigation yourselves, rather than merely copying and pasting commands, and thus to move beyond the stage of being a mere coding monkey.

One important point of vigilance is worth stressing. Each stage of the pipeline must be implemented in a fully independent manner, with manifests as the only inputs and outputs. Moreover, manifests must be created atomically, regardless of the operating system. If I insist so heavily on cross-platform behaviour, it is, as you may suspect, because this aspect is far from trivial to implement correctly.

You must initialise a Git repository for the project. The repository should track the code, the configuration files, as well as `dvc.yaml` and `params.yaml`. Audio files and large intermediate artefacts must not be versioned. In your final report, you must include a link to a public GitHub repository containing your pipeline, and you should submit only a PDF report. You know how highly I think of GitHub, but (i) it is a platform you are expected to be able to use, and (ii) it is by far the simplest way to ensure that I do not spend time I do not have setting up a GitLab instance and granting access.

Each step of the workflow must be declared as a DVC stage. For every stage, you must clearly specify its dependencies (scripts, manifests, and parameters) and its outputs (manifests, predictions, metrics, and figures). DVC should be used both to automatically re-run the pipeline when an input changes and to maintain a clear and interpretable history of experiments. Metrics and result figures must be explicitly declared.

You must first run the complete pipeline on a single language and verify that all stages execute correctly and that the expected metrics and figures are produced. You must then extend the pipeline by adding a new language. This extension must be carried out without modifying the code : only the pipeline parameters may be changed. After re-running the pipeline, you should verify that only the necessary stages are recomputed, that new language-specific curves appear, and that the mean curve is updated automatically.

If you manage to go from a performance curve for a single language to a curve covering all languages without changing your code (only the pipeline parameters) you can be

proud of yourselves : you will have met the objectives of the lab and taken another important step along the long and winding road that leads to becoming a proper NLP practitioner and/or data scientist.

All that remains is to write a short report describing the commands you used to build the pipeline, the difficulties you encountered, and the solutions you found. Do not forget to include a link to the GitHub repository of your project and to send me the report before the deadline.