Open source practices for music signal processing research

Brian McFee, Jong Wook Kim, Mark Cartwright, Justin Salamon, Rachel Bittner, and Juan Pablo Bello, *Senior Member, IEEE*

I. Introduction: reproducibility and complexity in MIR

In the early years of music information retrieval (MIR), research problems were often centered around conceptually simple tasks, and methods were evaluated on small, idealized datasets. A canonical example of this is genre recognition—*which one of N genres describes this song?*—which was often evaluated on the GTZAN dataset (1,000 musical excerpts balanced across 10 genres) [1]. As task definitions were simple, so too were signal analysis pipelines, which often derived from methods for speech processing and recognition, and typically consisted of simple methods for feature extraction, statistical modeling, and evaluation. When describing a research system, the expected level of detail was superficial: it was sufficient to state, *e.g.*, the number of Mel-frequency cepstral coefficients (MFCCs) used, the statistical model (*e.g.*, a Gaussian mixture model), the choice of dataset, and the evaluation criteria, without stating the underlying software dependencies or implementation details. Owing to the increased abundance of methods, proliferation of software toolkits, the explosion of machine learning, and a shift of focus toward more realistic problem settings, modern research systems are substantially more complex than their predecessors. Modern MIR research therefore requires careful attention to detail when processing meta-data, implementing evaluation criteria, and disseminating results.

The common practice in MIR research has been to publish findings when a novel variation of some component of the system (such as the feature representation or statistical model) led to an increase in performance. This approach is sensible when all relevant factors of an experiment can be enumerated and controlled, and when we have confidence in the correctness and stability of the underlying implementation. However, over time, researchers have discovered that confounding factors were prevalent and undetected in many research systems, which undermines previous

Music and Audio Research Lab
New York University

findings. Confounding factors can arise from quirks in data collection [2], subtle design choices in feature representations [3], or unstated assumptions in the evaluation criteria [4].

As it turns out, implementation details can have greater impacts on overall performance than many practitioners might expect. For example, Raffel et al. [4] reported that differences in evaluation implementation can produce deviations of 9–11% in commonly used metrics across diverse tasks including beat tracking, structural segmentation, and melody extraction. This results in a manifestation of the *reproducibility crisis* [5] within MIR: if implementation details can have such a profound effect on the reported performance of a method, it becomes difficult to trust or verify empirical results. Reproducibility is usually facilitated by access to common data sets, which would allow independent re-implementations of a proposed method to be evaluated and compared to published findings. However, MIR studies often rely on private or copyright data sets which cannot be shared openly. This shifts the burden of reproducibility from common data to common software: while data sets often cannot be shared, implementations usually can.

In this article, we share experiences and advice gained from developing open source software for MIR research, with the hope that practitioners in other related disciplines may benefit from our findings and become effective developers of open source scientific software.

Many of the issues we encounter in MIR applications are likely to recur in more general signal processing areas, as data sets increase in complexity, evaluation becomes more integrated and realistic, and traditionally small research components become integrated with larger systems.

A. Open source scientific software

We adopt the position previously voiced by numerous authors [6] that description of research systems is no longer sufficient, which follows from the position that scholarly publication serves primarily as advertisement for the scientific contributions embodied in the software and data [7]. Here, we specifically advocate for adopting modern open source software development practices in communicating scientific results.

The motivations for our position, while grounded in music analysis applications, apply broadly to any field in which systems reach a sufficiently high degree of complexity. Releasing software as “open source” requires more than posting code on a web site. We highlight several key ingredients of good research software practices:

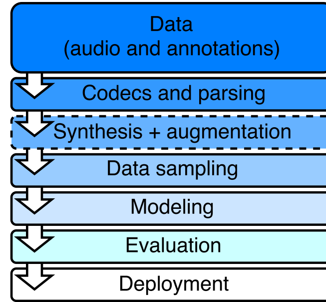

Fig. 1. A system block diagram of a typical MIR pipeline.

- *licensing*: to define the conditions under which the software can be used;
- *documentation*: so that users know how to operate the software, and what exactly it does;
- *testing*: so that the software is reliable;
- *packaging*: so that the software can be easily installed and managed in an environment;
- *application interface design*: so that the software can be easily integrated with other tools.

We discuss best practices for open source software development in the context of MIR applications, and propose future directions for incorporating open source and open science methodology in the creation of datasets.

II. System architecture and components

Figure 1 illustrates a generic, but representative MIR system pipeline, consisting of seven distinct stages. We describe each stage to provide a sense of scale involved in MIR research, document sources of software dependencies, and give pointers to common components.

The first stage is *data storage*, which is often implemented by organizing data on disk according to a file naming convention and directory structure. Storage may also be provided by relational databases (*e.g.*, SQLite [8]), key-value/document stores (*e.g.*, MongoDB https://www.mongodb.com or Redis https://redis.io), or structured numerical data formats (*e.g.*, HDF5 [9]). As datasets become larger and more richly structured, storage plays a critical role in the overall system.

The second stage is *input decoding*, which loosely captures the transformation of raw data (compressed audio or text data) into formats more convenient for modeling (typically vector representations). For audio, this consists primarily of compression codecs, which are provided by a small number of standard libraries (*e.g.*, ffmpeg [10] or libsndfile [11]). Although different (lossy) codec implementations are not guaranteed to produce numerically equivalent results, the

differences are usually small enough to be ignored for most practical applications. For annotations and metadata, the situation is less clear. Many datasets are provided in non-standard formats (*e.g.*, comma-separated values) which require custom parsers that can be difficult to correctly implement and validate. Although several formats have been proposed for encoding annotations and metadata (MusicXML [12], MEI [13], MPEG-7 [14], JAMS [15]), at this point none have emerged as a clear standard within the MIR community.

The third stage, *synthesis and augmentation*, is not universal, but it has seen rapid growth in recent years. This stage captures processes which automatically modify or expand datasets, usually with the aim of increasing the size or diversity of training sets for fitting statistical models. *Data augmentation* methods apply systematic perturbations to an annotated dataset, such as pitch-shifting or time-stretching, to induce these properties as invariants in the model [16]. Relatedly, *degradation* methods apply similar techniques to evaluation data as a means of diagnosing failure modes in a model once its parameters have been estimated [17]. *Synthesis* methods, like augmentation, seek to generate realistic examples either for training or evaluation, and though the results are synthetic, they are free of annotation errors [18]. Since these processes can have profound impact on the resulting model, it is important that augmentation and synthesis be fully documented and reproducible. Modern frameworks such as MUDA [16] and Scaper [18] achieve data provenance by embedding the generation/augmentation parameters within the generated objects, thereby facilitating reproducibility.

The fourth stage, *data sampling* refers to how a collection is partitioned and sampled when fitting statistical models. For statistical evaluation, data is usually partitioned into *training* and *testing* subsets, and this step is usually implemented within a machine learning toolkit (*e.g.*, SciKit-Learn [19]). We emphasize data partitioning because it can be notoriously difficult to implement correctly when dealing with related samples, such as multiple songs by a common artist [20]. *Stochastic sampling* is an increasingly important step, as it defines the sequences of examples used to estimate model parameters. Modern methods trained by stochastic gradient descent can be sensitive to initialization and sampling, so it is important that the entire process be codified and reproducible. Often, sampling is only specified implicitly, and provided by machine learning frameworks without explicit reproducibility guarantees. Sampling also becomes an engineering challenge when the training data exceeds the memory capacity of the system,

which is common when dealing with large data-sets.For problems involving large datasets, some framework-independent libraries have been developed to handle data sampling under resource constraints (*e.g.*, Pescador [21] and Fuel [22]).

The fifth stage, *modeling*, includes both feature extraction and statistical modeling, though the boundary between the two has blurred in recent years with the adoption of deep learning methods. Many open source libraries exist for audio feature extraction, such as Essentia [23], librosa [24], aubio [25], Madmom [26], or Marsyas [27]. Different libraries may produce different numerical representations for the same feature (*e.g.*, mel spectra), and even within a single library the robustness of different features to input encoding/decoding may vary [28]. While robustness is distinct from reproducibility, it highlights the importance of sharing specific software implementations. The statistical modeling component is most often provided by a machine learning framework, such as SciKit-Learn or Keras [29]. While the specific choice of framework is largely up to the practitioner's discretion, we emphasize that consideration should be given to how this choice interacts with the remaining two stages.

The sixth stage, *evaluation*, refers to measuring the performance of an entire developed system (not just the statistical model component). For simple classification problems, this functionality is typically provided by a machine learning framework (*e.g.*, SciKit-Learn). However, for domain-specific MIR problems, software packages have been developed to standardize evaluations, such as mir_eval for music description and source separation [4], sed_eval for sound event detection [30], and rival for recommender systems [31].

Finally, the last stage is *deployment*, by which we broadly mean dissemination of results (publication), packaging for reuse, or practical application in a real setting. This stage is perhaps the most overlooked in research, and possibly the most difficult to approach systematically, as the requirements vary substantially across projects. If we limit attention to reproducibility, *software packaging* emerges as an integral step to both internal reuse and scholarly dissemination. We therefore encourage researchers to take an active role in packaging their software components, and discuss in Section III specific tools for packaging and environment management.

A. *Example: onset detection*

Although we focus on large, integrated systems, it is instructive to see how system complexity plays out on a smaller scale, representative of earlier MIR work. As a conceptually simple example task, consider *onset detection*: the problem of estimating the timing of the beginning of musical notes in a recording. A method for solving this problem could be described as follows:

> Audio was converted to 22050 Hz (mono), and a 2048-point short-time Fourier transform (STFT) was computed with a 64-sample hop. The STFT was reduced to 128 Mel-frequency bands, and magnitudes were compressed by log scaling. An onset envelope was computed using thresholded spectral differencing, and peaks were selected using the method of Böck et al. [32].

This description is artificial, but the level of specificity given is representative of the literature.

While precise enough to be approximately reimplemented by a knowledgeable practitioner, the description above omits several details. To quantify the effect of these details, we conducted an experiment in which some unstated parameters are varied, and the resulting accuracy is measured on a standard dataset [33]. We varied the window function for STFT (*Hann* or *Hamming*), the log scaling (*bias-stabilized* $\log(1+X)$ or *clipped* 80dB below peak magnitude), and the differencing operator (*first order difference*, or a *Savitsky-Golay filter* as commonly used in *delta feature* implementations [34]). These 3 choices produce 8 configurations which are all consistent with the given description, any of which constitute a reasonable attempt at reconstructing the described method. There are of course many other parameters unstated above: the exact specification of the Mel filter-bank, how aggregation across frequency bands was computed, *etc.* For the sake of brevity, we limit the scope of this experiment to the three choices listed above.

Figure 2 shows the distribution of F-measure (harmonic mean of precision and recall) for each configuration. While the best-performing versions are approximately equivalent, the range of scores is quite large, spanning 0.43 to 0.76. Moreover, some decisions can appear to have a significant effect in some conditions (*e.g.*, the differencing filter when using Hamming windows) that vanishes in other conditions (*e.g.*, using a Hann window). This demonstrates that an incomplete system description can lead to incorrect conclusions about a particular design choice. Note that the interventions performed in this experiment are confined to a single stage of Figure 1 (modeling), but realistic systems are susceptible to variation at each stage of the pipeline.

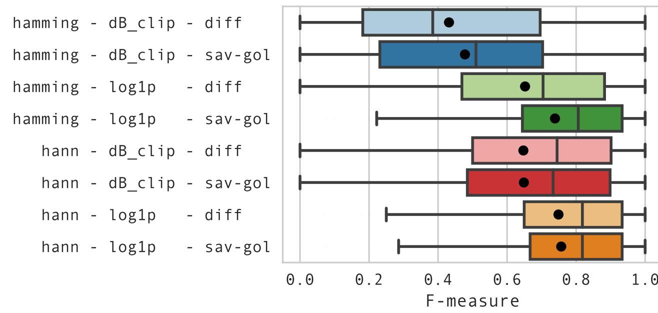

Fig. 2. Results of the onset detection experiment: each box corresponds to the inter-quartile range over test recordings, with the mean and median scores indicated by • and |, respectively. Each row corresponds to a system configuration which is consistent with the description given in section II-A, but differs in unstated parameters.

While this method is simple enough to be completely described in a short amount of text, full description quickly becomes impractical as methods become more complex. In modern research systems, the only practical means of fully characterizing the implementation is to provide the source code and data.

III. Best practices for OSS research

As described in the previous sections, modern research pipelines consist of many components with complex interactions. The engineering cost for developing and maintaining these components often exceeds that of implementing the core research method for a particular study. Sculley et al. discussed this cost as hidden technical debt, which is hard to notice and compounds silently [35]. In this section, we provide recommendations for open-source research software development, which can help improve code quality, reproducibility, and foster efficient long-term collaboration on large projects with distributed contributors. The suggestions we make here are broadly applicable outside MIR (or DSP), and we draw attention to these points specifically because domain experts are often not aware of their importance. Many of the recommendations given here are also implemented concretely in Shablona (https://github.com/uwescience/shablona), a template repository for starting scientific Python projects. Interested readers may wish to browse the Shablona repository while reading the following sections. Readers entirely new to software development and open source software may additionally benefit from the instructional materials provided by Software Carpentry (https://software-carpentry.org/) the Hitchhiker's Guide to Python (https://docs.python-guide.org/), and Wilson et al. [36].

A. Software Licensing

The defining characteristic of open source software is the *license*. Licenses dictate the terms under which software can be used, modified, or distributed. Note that if no license is explicitly stated, then no use, modification, or distribution is permitted [37], and to put it mildly, this significantly impedes adoption, reuse, and open science. It is therefore important to include a license agreement with any software intended for reuse and distribution.

There are many open source licenses to choose from, but four of the most popular licenses are MIT, BSD, Apache, and GPL. MIT and BSD-style are simple, permissive licenses with minimal requirements on how derivative works are distributed. Apache is also permissive, but it contains additional provisions, including a grant of patent rights from contributors to users. In contrast, GPL requires derivative works to be distributed under the same license terms.

Not all of these licenses will suit and individual or organization's needs. Therefore, it is common for particular communities to tend toward a specific license: the scientific Python community generally uses permissive, MIT or BSD-style licenses, while the R community mostly uses GPL-style licenses. A full discussion of the relative merits of different licensing options is far beyond the scope of this article, but we recommend https://choosealicense.com as a resource to help select and compare among the various options.

B. Documentation

Documentation is the primary source of information for users of a piece of software, and should be written and maintained with the most relevant and helpful content. A common practice for distributing documentation is to include it with the source code distribution, so that it is tightly coupled to the specific software version in use. It is recommended to use a documentation build tool that can automatically generate a website, using both explicit documentation files as well as the in-line comments in the source code for the application programming interface (API). Examples of such tools include Sphinx and MkDocs, and the generated website can be hosted on services such as Read The Docs (http://readthedocs.io). In addition to describing software functionality, documentation should also include relevant bibliographic references, and instructions for attribution.

To prevent the common problem where documentation falls out of sync with the software, it is important to document concurrently with programming. Similarly, before each new version of a package is released, a thorough audit of documentation should be conducted with respect to the changes introduced since the previous release. All changes should be summarized in a *CHANGELOG* or *release notes* section of the documentation, ideally with time-stamps, so that users can quickly discern changes introduced for each version. These simple steps, combined with semantic versioning and version control (both described below) require little effort, but substantially ease use and integration.

Finally, we emphasize the importance of providing *example* code in the documentation. While examples cannot replace a textual description of functionality, including self-contained example usage for each function or class (along with the expected output of the example code) can often be a more effective way of communicating the behavior of a component to a novice user.

C. Version Control Software

Version control software (VCS) is an essential tool for modern software development that 1) keeps track of who changed what, when, and for what reason, 2) supports creating and re-creating snapshots of everything in the project's history, and 3) enables a variety of tooling related to the software, such as test automation (Section III-D) and quality control (Section III-E). Git, currently the most popular VCS in OSS development, is a distributed VCS where the full history of the project is stored in every developer's computer. GitHub (https://github.com) is a service which offers free hosting for open-source projects, and leverages the decentralized nature of Git to provide a platform for collaboration of software developers. Bundled with the pull request feature (Section III-F) which allows users (internal and external to a project) to suggest changes, issue trackers, wikis, service integration, and website hosting, GitHub serves as the home for the majority of open source projects.

VCS is also important for managing releases, which are packaged versions of the software intended to be easily downloaded and used. Each release is marked with a version string (such as *1.5.3*), and semantic versioning (https://semver.org/) is a recommended practice of assigning software versions which can systematically inform the users about the incompatible changes to expect when updating the versions of the software. At a high level, semantic versioning states

that API-compatible revisions to a package retain the same major version index, which allows users (including other libraries) to loosely specify version requirements.

Unfortunately, there are no guarantees that a commercial hosting service like GitHub will persist indefinitely. Therefore, for software accompanying publications, we recommend using a funded research data repository such as Zenodo (see Section IV-D) in conjunction with Github. Large research data repositories typically guarantee multiple decades of longevity.

In short, we recommend using Git for efficient collaboration and sustainable development of software, with the help of GitHub for software distribution and issue tracking. GitLab is an alternative to GitHub which also offers a free hosting and issue tracking, but is can be locally installed and self-administered.

D. Automated Software Testing

It is beneficial for software projects to regularly perform automated tests to ensure the correctness of implementation. Automated software testing involves a set of specifications that precisely define the intended behaviors of the software, along with a testing framework that controls the execution of the tests and verifies that the software produces the expected outputs. The purpose of test automation is not only to verify that the current code works as intended, but also to quickly detect any regressions caused by changes to any part of the software.

Unit testing refers to automated testing of the smallest testable parts of the software—*units*—which are usually individual functions or classes. Specifying the behaviors of the individual units not only help find errors in the earliest stage of development, but also encourages a modular design composed of loosely coupled testable components. Other forms of testing include *integration testing*, where tests are designed to ensure that small components produce desired results when combined, and *regression testing* which compare current outputs to archived previous outputs, so that unexpected changes (*regressions*) can be easily and automatically detected.

By defining the guarantees of each part of the software and writing tests that can detect deviations from the guarantees, automated testing helps improve the stability and reliability of the software, and ultimately reduce the potential cost of undetected or late-detected errors. Test-driven development (TDD) is a software development process in which the specification is written prior to the actual development of features, and the implementations of features are then made to

pass the tests [38]. Although TDD is not often followed strictly, writing tests early in development can help both to clarify the intended behavior of a function, and to discover components that need to be simplified into smaller units.

E. Code Quality and Continuous Integration

Software developers should strive to maintain high quality code, meaning that it is well-formatted, well-organized, and clear to read. *Static analysis tools* are utilities which quantify various dimensions of code quality without executing the software. Many programming languages include static analyzers which test code adheres to a style (formatting and variable naming) guideline, such as Python's `pycodestyle` tool. Similarly, a *linter* is a static analysis tool that can suggest stylistic improvements to the *structure* of code, and identify possible sources of errors. Linters can also perform a measurement of code complexity and produce warnings if, for example, a function is too complex in its structure. A metric commonly used for measuring this is the cyclomatic complexity, which is the number of independent code paths in a unit of code.

Another important metric for code quality is *test coverage*, the proportion of code executed by the tests. Low code coverage implies that the software is not thoroughly tested and thus unlikely to be reliable. Having a low cyclomatic complexity is helpful in achieving high code coverage, because it determines the number of test cases required for achieve the full code coverage.

Integration is the task of putting the development outputs to a product, *i.e.*, ensuring quality by performing various automated tests and packaging the software for deployment. Continuous integration (CI) is a practice of performing integrations as frequently as possible by automating the process, so that the status of every change to the code is automatically verified. In addition to ensuring good software quality through automated tests, continuous integration provides a platform for automatic analysis of code quality. By using a version control system, continuous integration can be performed automatically at every registered change to the software, and services such as Travis CI (https://travis-ci.org), CircleCI (https://circleci.com) and AppVeyor (https://www.appveyor.com) provide free hosting for open source projects.

F. Project management, pull requests, and code review

While automated testing and static analysis are powerful tools, they must be used effectively to produce high-quality open source software. Ultimately, software is developed and maintained

by humans, and there is no total substitute for proper project management. A widely adopted practice in OSS development is to require that all changes to a code base be submitted via *pull requests*. A *pull request* combines one or more proposed revision to the software as a unit, which can either be accepted (*merged* into the main repository) or rejected. The benefit of this practice is that a pull request provides a convenient point for human intervention without having to manually track each individual change. Continuous integration systems typically execute all tests on a proposed pull request, which gives the project manager—who may be the same person as the pull request author—a quick way to determine whether the proposed changes conform to style requirements, are sufficiently tested and documented, and do not introduce test regressions.

Typically, a pull request should not be merged if any of the following conditions are not satisfied: 1) all tests pass, 2) test coverage has not decreased, 3) the code adheres to style requirements, 4) the proposed changes are properly documented. The first condition verifies that the proposed changes do not break existing behavior. The second condition requires that the proposed changes includes a minimum amount of corresponding tests. The third condition checks that the proposed change is stylistically consistent with the project's goals and existing code. The fourth condition ensures that the project's documentation does not fall out of sync with the source code. Of these, the first three conditions can be automated by continuous integration. However, that none of the above conditions ensure *correctness* of the proposed change, which ultimately should be determined (as best as possible) by a thorough *code review* by one or more parties beyond the author of the proposed changes. As a side effect, code review is also the ideal time to check the fourth condition, and request any modifications to the pull request. Adopting this work-flow early in a project's life cycle can provide structure to software development, and ease the burden of adhering to best practices (especially documentation and testing).

G. Interoperability and Interface Design

Publishing an OSS library means its functions and classes can be used by many users, who will benefit from a maintainable, extensible, and easy-to-understand API design. This includes programming practices such as descriptive function and variable naming, intuitive organization of functionality into sub-modules, and providing sensible default parameter values.

In addition to the importance of intuitive API design, we argue that function-oriented interfaces

are often better than object-oriented designs. In research settings, use cases are often procedural executions of steps in a pipeline, in which case using class hierarchies may entail unnecessary cognitive load. Functions have well-defined entry and exit points, making their lifespans explicit, but *objects* maintain state indefinitely, making it difficult to infer their scope. Moreover, classes do not easily traverse library boundaries, impeding inter-operability between components. If an API expects or produces an instance of a certain class, it forces every package depending on the API to conform to the specification of the class, and makes such packages sensitive to future changes in the class definition. For this reason, data containers can be better represented in the standardized, primitive collection types, such as dictionaries, lists, or Numpy's `ndarray` type.

In spite of the above arguments for function-oriented design, object-oriented interfaces can be useful when the primary goal explicitly requires persistent state. This is the case, for instance, when packaging statistical models, where the *state* encapsulates the model parameters.

H. Packaging and environments

Software is often organized into *packages* to facilitate maintainability and distribution, and it is a responsibility of a *package management system* to provide means to install specific versions of desired packages. Many programming languages provide package management systems that help organize installed libraries and applications, such as `pip` for Python and CRAN for R. These provide a way to specify dependency requirements, and user-interfaces to install and upgrade software. Because installing a software package becomes as simple as running a single-line command `[package-manager] install [package-name]`, it is often a good idea to distribute the software as a package for easier and wider adoption, even if the project is not primarily a library. Packages are constructed by *build tools*, which vary across languages, such as Python's `setuptools` or Java's `Gradle`. Working in conjunction with package management software, build tools allow a project to be packaged with its dependencies and their exact versions specified, along with the meta-data to help index the project within a repository.

Within Python, there are two dominant package systems: the Python Package Index (PyPI, or *pip* package manager) and Conda. The key distinction between these two systems is that *pip* can only package Python modules (and extensions written in C), but Conda packages can be written in any language. Conda packages thus allow dependency tracking across languages, so that a

package written in Python can have dependencies written in C (for example). This property is useful when developing large systems with heterogeneous components, as is common in MIR and likely to become more common in DSP more broadly in the future.

With all dependencies and their versions specified for a project, one can ensure the interoperability between components, and thus have an environment that provides reproducible results. However, libraries are known to change over time, and introduce incompatibilities across version upgrades. This can present a problem when reproducing an old experiment in a modern environment, or when working on multiple projects with conflicting dependencies. *Environment managers* (such as Conda or *virtualenv* in Python) resolve this by providing isolated environments in which packages can be installed. Virtual machines or containers like Docker can also provide isolated and reproducible environments which do not depend explicitly on the programming language in question. Container tools like ReproZip [39] can significantly ease reproducibility by automatically generating virtual machine images to reproduce a specific experiment.

I. Project structure

Figure 3 provides our recommended repository structure for MIR projects using Python, though the template could be easily adapted to other domains and languages. The top-level directory should at least include the license and a *readme* file which describes the project at a high level, and provides contact information for the authors. The file `env.yaml` (or *requirements.txt*) describes the software dependencies (and versions) necessary to reproduce the project's working environment; these should be automatically generated by a package or environment manager, *e.g.*, by executing `conda env export` or `pip freeze`.

The `data` sub-directory should contain any static data used in the experiment, such as a filename index of a dataset, or configuration files associated with various software components. Entire datasets need not be included in the repository here (to limit the size of the repository), but a script or instructions to procure the data should be provided.

The `scripts` sub-directory contains all the scripts needed to generate the results of the project. Here, we have taken inspiration from the UNIX System V *init* system, which organizes (system startup) scripts alpha-numerically to ensure a consistent order of execution. This simple convention eases reproducibility by eliminating any ambiguity in how the various components

```
project/                                      generated/
    LICENSE.txt                                   split01/
    README.txt                                        index-train.json
    env.yaml (or requirements.txt)                    index-test.json
    data/                                             model_parameters.h5
        index-all.json                                results.json
        ...                                       split02/
    scripts/                                          ...
        01-data-augmentation.py                   notebooks/
        02-pre-process.py                             01-analysis.ipynb
        03-model.py                                   ...
        04-evaluate.py
        ...
```

Fig. 3. An example file structure for an MIR research project.

should be executed. The exact subdivision of steps is not critical, but the four listed here—synthesis/augmentation, pre-processing, model estimation, and evaluation—apply broadly to many situations, and we have found this loose organization to be flexible and useful in our own projects.

The *pre-processing* step can entail a variety of processes that generate intermediate data, such as pre-computed feature transformations, or train-test splits of a dataset. For diagnostic purposes, we specifically advocate generating train-test index partitions independent of model estimation, and saving all index sets to disk as index files (*e.g.*, `splitNN/index_train.json`). This small amount of book-keeping can significantly ease debugging, reproducibility, and facilitate fair, paired comparisons between different methods over the same data partitions. All data produced automatically should be kept separate from the static `data` directory, *e.g.*, in a dedicated `generated` directory; and if there are multiple train-test splits, then all split-dependent data should be kept in its own sub-directory (or otherwise separated by filename) to prevent statistical contamination across partitions.

We recommend that any (interactive) post-hoc analysis of the results, including figure generation for publications, be stored separately under `notebooks`. Here, we suggest Jupyter notebooks (https://www.jupyter.org/), which are portable and support interactive execution in a variety of languages. If multiple steps are necessary, we again recommend ordering the files alpha-numerically to disambiguate execution order.

As a final note, we suggest that *all* (pseudo-) randomized computations throughout the process use a fixed seed, which can be easily set by a user. This ensures that the entire system is deterministic, and can significantly aid in debugging and reproducibility.

IV. PROPOSAL: TOOLS FOR DATA COLLECTION AND DISTRIBUTION

Just as complex systems often require multiple software components, they increasingly also require multiple datasets. Similar to software, datasets can also change over time, either from extension or correction [40]. In addition, even small changes in the data collection and processing pipeline can affect results. For example, previous studies have shown that even the visualization used in audio annotation can affect annotation quality [41]. Researchers also often "process" or "clean" annotations by removing outliers or aggregating annotations. These processes must be documented to appropriately use and extend annotations. While many open source principles can also be applied to data, there is much work to be done regarding tooling and infrastructure to support OSS practices for data collection. This section is both a position statement and a proposal to the community, in which we outline what has been done and propose what needs to be done to move forward regarding the tooling of data collection and distribution.

A. Data Annotation

First, we propose that the research community should develop and adopt standard, open source tools for audio annotation. This not only ensures that we are not replicating existing work with several ad hoc annotation solutions, but also that we are following best practices and can extend existing datasets developed by other research groups.

In addition to following the OSS principles outlined earlier in this article, these tools should also be configurable, extensible, web-based, so that they can be easily deployed without requiring users (annotators) to install software. Web-based solutions enable easy distribution of audio and crowd-sourced annotation, a now standard method for obtaining large numbers of annotations. While many of our data needs can be met using strong or weak labeling tasks, some of our data needs require more specialized, unforeseen tasks. Therefore, these tools should be *extensible*, with the capability to support new tasks and workflows. Lastly, the configuration of these tools—instructions, workflow definitions, task configurations, etc.—should also be stored in a single location in a human-readable format.

A number of open source, desktop applications have been already developed for annotation, such as Raven [42], Audacity [43] or Tony [44], but only recently have we seen the emergence of web-based tools for crowd-sourcing. The Audio Annotator is a simple web-based front-end

for strong-labeling of audio with standard audio visualizations [41]. While a good starting point, its functionality is limited, and it is not easily extensible. Freesound Datasets is a new web-based platform for crowd-sourcing weak labels of audio hosted on https://freesound.org [45]. However, it is currently limited to Freesound data, and also not extensible. Zooniverse is the most popular citizen science platform, with over a million registered users [46]. Zooniverse supports audio content and audio visualizations, but the available task types are limited to weak labeling and survey questions, and its extensibility is limited.

B. Dataset File Formats

As described in section II, standardized tools for reading and writing data file formats minimize the risk of parsing errors and eases distribution and use of data. There are several formats for encoding music annotations (MusicXML [12], MEI [13], MPEG-7 [14], JAMS [15]), but these formats are primarily for managing annotations for a single recording, rather than collections of annotated audio. To increase transparency and usability of datasets, we propose to develop a package to support collection management. Only the raw annotations and audio would be stored as data, and "views" could be defined to filter and process the data for a specific task. For example, if a dataset needs to be "cleaned" to remove erroneous annotations or outliers, then users could write a "clean" view of the data without discarding information. Additionally, pre-registered splits of the data could be implemented as a views on top of an existing view. Dataset files would also contain standardized meta-data and documentation of the dataset creation process.

C. Data Documentation

To understand the content of datasets, use them appropriately, and extend them when necessary, datasets must be thoroughly documented. This motivates standard reporting mechanisms and tools to facilitate the documentation of the data collection process. While standards should be developed and ratified by the community, the following are possible items to include for each annotation: 1) annotation software and version, 2) annotation software configuration, 3) description of all tasks, including: a) participant screening, b) training, c) annotation tasks, d) surveys, 4) description of annotator recruiting, 5) monetary (or other extrinsic) compensation mechanisms, 6) anonymized annotator IDs, 7) timestamps, 8) data "cleaning" or "processing" procedures, 9) data "synthesis" procedures description and code (if applicable).

Note that all such documentation should provide reasonable explanations and justifications for the choices made. This again helps the community understand the data and what it can be used for. We as a community should also determine screening and demographic survey procedures, and annotator quality metrics. Once these have been established, documentation tools in combination with standardized annotation file formats should be able to quickly aggregate and display this information about the population as a whole. Best practices for data documentation have been proposed before in MIR, though adoption by the community has been slow [47]. Recently, Gebru et al. proposed a standardized "datasheet" format for general machine learning datasets, inspired by the standardized datasheets that accompany electronic components [48].

D. Data Distribution

Lastly, we need tools to distribute, maintain, and index public datasets. While many of the requirements for data are similar to those of software, data typically requires more storage than software, rendering many existing services unsuitable. Data hosting should support versioning to support changes to datasets, provide Document Object Identifiers (DOIs), and guarantee longevity for several decades in order to prevent broken URLs and ephemeral data. These data requirements files would specify the datasets and versions required by software, and should be distributed along with the software requirements files. There are currently several hosting solutions that support large datasets, versioning, DOIs, and guarantee decades of longevity (e.g., Zenodo https://zenodo.org, Figshare https://figshare.org, Dryad https://datadryad.org, Dataverse https://dataverse.org). Unfortunately, these solutions have yet to develop a data management tool like we've described. However, it may be possible for a third party to build such a tool around the existing infrastructure.

In addition to hosting and distribution, we also need a platform for developing and maintaining data. At the minimum, this would include an issue tracker for reporting errors and proposing/discussing improvements to existing datasets. However, this could also double as a platform for proposing and discussing the creation of new datasets. While such functionality would ideally be integrated into hosting services, this could also be developed around existing infrastructure or supported with existing platforms such as GitHub.

V. Conclusion

While MIR has long been data-driven, and necessarily complex due to the long chain of steps involved in bridging audio signals and semantically meaningful representations, we expect the core issues of system complexity to eventually pervade all data-driven areas of signal processing. The general architecture outlined in section II is generic enough to capture most MIR use cases, and though different domains might exhibit slightly different work-flows, we expect that the overall system complexity issue will arise across domains. The recommendations put forward in section III should serve as a solid basis for improving the quality and reproducibility of scholarly research. Though we do not expect signal processing researchers to become experts in software engineering, we focus here on software precisely because it is often overlooked as a crucial component of research systems. Although most of our recommendations concern software, we see data management as the next frontier in improving data-driven research in general, and signal processing research specifically. The proposal put forward in section IV is intended to resolve certain short-comings in our current practices for dataset construction, but may be readily adapted to different application domains. We encourage future researchers to think carefully about data construction, preservation, and management issues moving forward.

References

[1] G. Tzanetakis and P. Cook, "Musical genre classification of audio signals," *IEEE Transactions on speech and audio processing*, vol. 10, no. 5, pp. 293–302, 2002.

[2] B. L. Sturm, "Revisiting priorities: Improving MIR evaluation practices," in *Proceedings of the 17th International Society for Music Information Retrieval Conference, ISMIR 2016, New York City, United States, August 7-11, 2016*, 2016, pp. 488–494.

[3] T. Cho, R. J. Weiss, and J. P. Bello, "Exploring common variations in state of the art chord recognition systems," in *Sound and Music Computing Conference*, 2010.

[4] C. Raffel, B. McFee, E. J. Humphrey, J. Salamon, O. Nieto, D. Liang, and D. P. W. Ellis, "mir_eval: A transparent implementation of common MIR metrics," in *Proceedings of the 15th International Society for Music Information Retrieval Conference, ISMIR 2014, Taipei, Taiwan, October 27-31, 2014*, 2014, pp. 367–372.

[5] H. Pashler and E. Wagenmakers, "Editors introduction to the special section on replicability in psychological science: A crisis of confidence?" *Perspectives on Psychological Science*, vol. 7, no. 6, pp. 528–530, 2012. [Online]. Available: https://doi.org/10.1177/1745691612465253

[6] P. Vandewalle, J. Kovacevic, and M. Vetterli, "Reproducible research in signal processing," *IEEE Signal Processing Magazine*, vol. 26, no. 3, 2009.

[7] J. B. Buckheit and D. L. Donoho, "Wavelab and reproducible research," in *Wavelets and statistics*. Springer, 1995.

[8] M. Owens and G. Allen, *SQLite*. Springer, 2010.

[9] M. Folk, A. Cheng, and K. Yates, "HDF5: A file format and I/O library for high performance computing applications," in *Proceedings of supercomputing*, vol. 99, 1999, pp. 5–33.

[10] F. Bellard, M. Niedermayer *et al.*, "FFmpeg," *Available from: http://ffmpeg.org*, vol. 3, 2012.

[11] E. de Castro Lopo, "Libsndfile," 2011.

[12] M. Good, "MusicXML for notation and analysis," *The virtual score: representation, retrieval, restoration*, vol. 12, 2001.

[13] P. Roland, "The music encoding initiative (MEI)," in *Proceedings of the First International Conference on Musical Applications Using*, vol. 1060, 2002, pp. 55–59.

[14] S.-F. Chang, T. Sikora, and A. Purl, "Overview of the MPEG-7 standard," *IEEE Transactions on circuits and systems for video technology*, vol. 11, no. 6, pp. 688–695, 2001.

[15] E. J. Humphrey, J. Salamon, O. Nieto, J. Forsyth, R. M. Bittner, and J. P. Bello, "JAMS: A JSON annotated music specification for reproducible MIR research," in *Proceedings of the 15th International Society for Music Information Retrieval Conference, ISMIR 2014, Taipei, Taiwan, October 27-31, 2014*, 2014, pp. 591–596.

[16] B. McFee, E. J. Humphrey, and J. P. Bello, "A software framework for musical data augmentation," in *Proceedings of the 16th International Society for Music Information Retrieval Conference, ISMIR 2015, Málaga, Spain, October 26-30, 2015*, 2015, pp. 248–254.

[17] M. Mauch and S. Ewert, "The audio degradation toolbox and its application to robustness evaluation," in *Proceedings of the 14th International Society for Music Information Retrieval Conference, ISMIR 2013, Curitiba, Brazil, November 4-8, 2013*, 2013, pp. 83–88.

[18] J. Salamon, D. MacConnell, M. Cartwright, P. Li, and J. P. Bello, "Scaper: A library for soundscape synthesis and augmentation," in *WASPAA*, New Paltz, NY, USA, Oct. 2017.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[20] B. Whitman, G. Flake, and S. Lawrence, "Artist detection in music with minnowmatch," in *Neural Networks for Signal Processing XI, 2001. Proceedings of the 2001 IEEE Signal Processing Society Workshop*. IEEE, 2001.

[21] B. McFee, C. Jacoby, E. J. Humphrey, and W. Pimenta, "pescadores/pescador: 2.0.0," Feb. 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1165998

[22] B. Van Merriënboer, D. Bahdanau, V. Dumoulin, D. Serdyuk, D. Warde-Farley, J. Chorowski, and Y. Bengio, "Blocks and fuel: Frameworks for deep learning," *arXiv preprint arXiv:1506.00619*, 2015.

[23] D. Bogdanov, N. Wack, E. Gómez, S. Gulati, P. Herrera, O. Mayor, G. Roma, J. Salamon, J. R. Zapata, and X. Serra, "Essentia: An audio analysis library for music information retrieval," in *Proceedings of the 14th International Society for Music Information Retrieval Conference, ISMIR 2013, Curitiba, Brazil, November 4-8, 2013*, 2013, pp. 493–498.

[24] B. McFee, C. Raffel, D. Liang, D. P. Ellis, M. McVicar, E. Battenberg, and O. Nieto, "librosa: Audio and music signal analysis in python," in *Proceedings of the 14th python in science conference*, 2015, pp. 18–25.

[25] P. Brossier, "Aubio, a library for audio labelling," *Computer program, Retrieved Aug*, vol. 27, 2009.

[26] S. Böck, F. Korzeniowski, J. Schlüter, F. Krebs, and G. Widmer, "Madmom: A new python audio and music signal processing library," in *Proceedings of the 2016 ACM on Multimedia Conference*. ACM, 2016, pp. 1174–1178.

[27] G. Tzanetakis and P. Cook, "Marsyas: A framework for audio analysis," *Organised sound*, vol. 4, no. 3, pp. 169–175, 2000.

[28] J. Urbano, D. Bogdanov, P. Herrera, E. Gómez, and X. Serra, "What is the effect of audio quality on the robustness of mfccs and chroma features?" in *Proceedings of the 15th International Society for Music Information Retrieval Conference, ISMIR 2014, Taipei, Taiwan, October 27-31, 2014*, 2014, pp. 573–578.

[29] F. Chollet *et al.*, "Keras," 2015.

[30] A. Mesaros, T. Heittola, and T. Virtanen, "Metrics for polyphonic sound event detection," *Applied Sciences*, vol. 6, no. 6, p. 162, 2016.

[31] A. Said and A. Bellogín, "Rival: a toolkit to foster reproducibility in recommender system evaluation," in *Proceedings of the 8th ACM Conference on Recommender systems*. ACM, 2014, pp. 371–372.

[32] S. Böck, F. Krebs, and M. Schedl, "Evaluating the online capabilities of onset detection methods," in *Proceedings of the 13th International Society for Music Information Retrieval Conference, ISMIR 2012, Mosteiro S.Bento Da Vitória, Porto, Portugal, October 8-12, 2012*, 2012, pp. 49–54.

[33] S. Böck, "onset_db," https://github.com/CPJKU/onset_db.

[34] D. P. Ellis, "PLP and RASTA (and MFCC, and inversion) in MATLAB using melfcc.m and invmelfcc.m," 2006. [Online]. Available: http://www.ee.columbia.edu/~dpwe/resources/matlab/rastamat

[35] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in Neural Information Processing Systems*, 2015, pp. 2503–2511.

[36] G. Wilson, J. Bryan, K. Cranston, J. Kitzes, L. Nederbragt, and T. K. Teal, "Good enough practices in scientific computing," *PLoS computational biology*, vol. 13, no. 6, p. e1005510, 2017.

[37] "Choosealicense - no license," https://choosealicense.com/no-permission/, accessed July 9th 2018.

[38] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[39] F. S. Chirigati, D. E. Shasha, and J. Freire, "ReproZip: Using provenance to support computational reproducibility." in *TaPP*, 2013.

[40] B. L. Sturm, "An analysis of the GTZAN music genre dataset," in *Proceedings of the second international ACM workshop on Music information retrieval with user-centered and multimodal strategies*. ACM, 2012, pp. 7–12.

[41] M. Cartwright, A. Seals, J. Salamon, A. Williams, S. Mikloska, D. MacConnell, E. Law, J. Bello, and O. Nov, "Seeing sound: Investigating the effects of visualizations and complexity on crowdsourced audio annotations," *Proceedings of the ACM on Human-Computer Interaction*, vol. 1, no. 1, 2017.

[42] Bioacoustics Research Program, “Raven pro: Interactive sound analysis software (version 1.5),” http://www.birds.cornell.edu/raven, accessed July 9th 2018, 2014.

[43] D. Mazzoni and R. Dannenberg, “Audacity [software]. pittsburg,” 2000.

[44] M. Mauch, C. Cannam, R. Bittner, G. Fazekas, J. Salamon, J. Dai, J. Bello, and S. Dixon, “Computer-aided melody note transcription using the tony software: Accuracy and efficiency,” 2015.

[45] E. Fonseca, J. Pons Puig, X. Favory, F. Font Corbera, D. Bogdanov, A. Ferraro, S. Oramas, A. Porter, and X. Serra, “Freesound datasets: a platform for the creation of open audio datasets,” in *Proc. 18th International Society for Music Information Retrieval Conference (ISMIR)*, Suzhou, China, Oct. 2017, pp. 486–493.

[46] K. Borne and Z. Team, “The zooniverse: A framework for knowledge discovery from citizen science data,” in *AGU Fall Meeting Abstracts*, 2011.

[47] G. Peeters and K. Fort, “Towards a (better) definition of the description of annotated MIR corpora,” in *Proceedings of the 13th International Society for Music Information Retrieval Conference, ISMIR 2012, Mosteiro S.Bento Da Vitória, Porto, Portugal, October 8-12, 2012*, 2012, pp. 25–30.

[48] T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. Daumeé III, and K. Crawford, “Datasheets for datasets,” *arXiv preprint arXiv:1803.09010*, 2018.