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

Open-Source Practices for Music Signal Processing Research

Recommendations for transparent, sustainable, and reproducible audio research

Digital Object Identifier 10.1109/MSP.2018.2875349
Date of publication: 24 December 2018

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 data sets. A canonical example of this is genre recognition—i.e., Which one of *n* genres describes this song?—which was often evaluated on the GTZAN data set (1,000 musical excerpts balanced across ten 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 used, the statistical model (e.g., a Gaussian mixture model), the choice of data set, and the evaluation criteria, without stating the underlying software dependencies or implementation details. Because of an increased abundance of methods, the proliferation of software toolkits, the explosion of machine learning, and a focus shift toward more realistic problem settings, modern research systems are substantially more complex than their predecessors. Modern MIR researchers must pay careful attention to detail when processing metadata, implementing evaluation criteria, and disseminating results.

Reproducibility and Complexity in MIR

The common practice in MIR research has been to publish findings when a novel variation of some system component (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 the researchers 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 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 reimplementations of a proposed method to be evaluated and compared with published findings. However, MIR studies often rely on private or copyrighted data sets that cannot be shared openly. This shifts the burden of reproducibility from common data to common software: although data sets often cannot be shared, implementations usually can.

In this article, we share experiences and advice gained from developing open-source software (OSS) for MIR research with the hope that practitioners in other related disciplines will 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.

Open-source scientific software

We agree with 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 by the software and data [7]. Here, we specifically advocate for adopting modern OSS development practices when communicating scientific results.

The motivations for our position, although 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 website. We highlight several key ingredients of good research software practices:

- *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 OSS development in the context of MIR applications and propose future directions for incorporating open-source and open-science methodology in the creation of data sets.

System architecture and components

Figure 1 shows 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 a 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 at https://www.mongodb.com or Redis at https://redis.io), or structured numerical data formats (e.g., HDF5 [9]). As data sets become larger and more richly structured, storage plays a critical role in the overall system.

Input decoding, the second stage, 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 few 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 data sets are provided in nonstandard formats (e.g., comma-separated values) that 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], and JAMS [15]), at this point none have emerged as a clear standard in 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 that automatically modify or expand data sets, 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 data set, 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, although the results are synthetic, they are free of annotation errors [18]. Because these

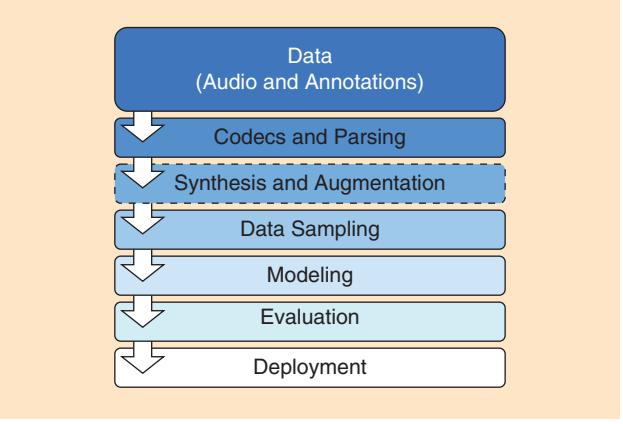

FIGURE 1. A system block diagram of a typical MIR pipeline.

processes can have a 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.

Data sampling, the fourth stage, refers to how a collection is partitioned and sampled when fitting statistical models. For statistical evaluation, data are 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, because 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 specified only implicitly and is provided by machine-learning frameworks without explicit reproducibility guarantees. Sampling also becomes an engineering challenge when the training data exceed the memory capacity of the system, which is common when dealing with large data sets. For problems involving large data sets, some framework-independent libraries have been developed to handle data sampling under resource constraints (e.g., Pescador [21] and Fuel [22]).

Modeling, as the fifth stage, includes both feature extraction and statistical modeling, although 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]. Although 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]. Although 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.

Referring to measuring the performance of an entire developed system (not just the statistical model component) is the sixth stage, evaluation. 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 is possibly the most difficult to approach systematically, because 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 in the "Best Practices for OSS Research" section we discuss specific tools for packaging and environment management.

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 next.

Audio was converted to 22,050 Hz (mono), and a 2,048-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.

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

Figure 2 shows the distribution of F-measure (harmonic mean of precision and recall) for each configuration. Although the best-performing versions are approximately equivalent, the range of scores is quite large, spanning 0.43 to 0.76. Moreover, some decisions can 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. 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.

Although this method is simple enough to be completely described in a short amount of text, a 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.

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. [35] discussed this cost as hidden technical debt, which is hard to notice and compounds silently. In this section, we provide recommendations for open-source research software development, which can help improve code quality and reproducibility and foster efficient long-term collaboration on large projects with distributed contributors. The suggestions we make here are broadly applicable outside MIR [or digital signal processing (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 OSS 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].

Software licensing

The defining characteristic of OSS is the license. Licenses dictate the terms under which software can be used, modified, or distributed. 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. Therefore, it is 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 the Massachusetts Institute of Technology (MIT), Berkeley Software Distribution (BSD), Apache, and General Public License (GPL). MIT and BSD 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 an individual's or organization's needs. Therefore, it is common for particular communities to tend toward a specific type of license: the scientific Python community generally uses the more permissive MIT- or BSD-style licenses, whereas the R programming language 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 the various options.

Documentation

Documentation is the primary source of information for users of a piece of software, and it 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. We recommend using a documentation build tool that can automatically generate a website using both explicit documentation files and 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 of documentation falling 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 (described in the following sections), require little effort, but they substantially ease use and integration.

Finally, we emphasize the importance of providing example code in the documentation. Although 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

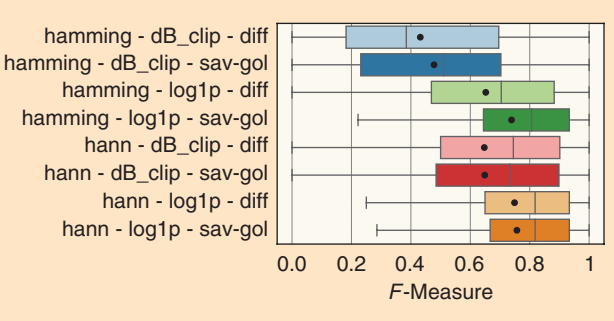

FIGURE 2. The results of the onset detection experiment: Each box corresponds to the interquartile range over test recordings, with the mean and median scores indicated by • and |, respectively. Each row corresponds to a system configuration that is consistent with the description given in the "Example: Onset Detection" section, but differs in unstated parameters.

effective way of communicating the behavior of a component to a novice user.

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 recreating snapshots of everything in the project's history; and 3) enables a variety of tooling related to the software, such as test automation (see the "Automated Software Testing" section) and quality control (see the "Code Quality and Continuous Integration" section). 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 that 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 (see the "Project Management, Pull Requests, and Code Review" section) that allows users (internal and external to a project) to suggest changes; issues trackers; and provides 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 that can systematically inform the users about the incompatible changes to expect when updating versions. 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 the "Data Distribution" section) 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 that also offers free hosting and issue tracking but can be locally installed and self-administered.

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 helps programmers 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 compares current outputs to archived previous outputs so that unexpected changes (regressions) can be easily and automatically detected.

Large research data repositories typically guarantee multiple decades of longevity.

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 reduces the potential cost of undetected or late-detected errors. Test-driven development (TDD) is a software development process in which the specification is written before the actual development of features, and the implementations of features are then made to pass the tests [38]. Although TDD protocol is not always strictly followed, writing tests early in development can help programmers clarify the intended behavior of a function and discover components that need to be simplified into smaller units.

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 that quantify various dimensions of code quality without executing the software. Many programming languages include static analyzers to test that 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, e.g., 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 is 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 to 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.

Project management, pull requests, and code review

Although automated testing and static analysis are powerful tools, they must be used effectively to produce high-quality OSS. 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 revisions to the software as a unit that 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 the need 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, and 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 include a minimum amount of corresponding tests. The third condition checks that the proposed changes are 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, none of the conditions ensures the 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. Incidentally, code review is also the ideal time to check the fourth condition and request any modifications to the pull request. Adopting this workflow 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).

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 submodules, and 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, and using class hierarchies may entail unnecessary cognitive load. Functions have well-defined entry and exit points, making their life spans explicit, but objects maintain state indefinitely, making it difficult to infer their scope. Moreover, classes do not easily traverse library boundaries, impeding interoperability 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 this 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 `ndarray` type.

Despite these 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.

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 with their exact versions specified, along with the metadata to help index the project in 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 package only 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, e.g., can have dependencies written in C. This property is useful when developing large systems with heterogeneous components, as is common in MIR and likely to become 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 that 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.

Project structure

Figure 3 provides our recommended repository structure for MIR projects using Python, although the template could be easily adapted to other domains and languages. The top-level directory should at least include the license and a readme.txt file that 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` subdirectory should contain any static data used in the experiment, such as a filename index of a data set or configuration files associated with various software components. Entire data sets 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` subdirectory contains all of 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 alphanumerically to ensure a consistent order of execution. This simple convention eases reproducibility by eliminating any ambiguity in how the various components should be executed. The exact subdivision of steps is not critical, but the four listed here—synthesis/augmentation, preprocessing, model estimation, and evaluation—apply broadly to many situations. We have found this loose organization to be flexible and useful in our own projects.

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

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

The preprocessing step can entail a variety of processes that generate intermediate data, such as precomputed feature transformations or train-test splits of a data set. 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 bookkeeping can significantly ease debugging and reproducibility and can 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; if there are multiple train-test splits, then all split-dependent data should be kept in their own subdirectory (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 and 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 alphanumerically 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.

Proposal: Tools for data collection and distribution

Just as complex systems often require multiple software components, they increasingly also require multiple data sets. Similar to software, data sets 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. Although 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.

Data annotation

First, we propose that the research community should develop and adopt standard, open-source tools for audio annotation. This ensures not only 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 data sets developed by other research groups.

In addition to following the OSS principles outlined earlier in this article, these tools should also be configurable, extensible, and 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 crowdsourced annotation, now a standard method for obtaining large numbers of annotations. Although 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, i.e., with the capability to support new tasks and workflows. Finally, the configuration of these tools—instructions, workflow definitions, task configurations, and so on—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 crowdsourcing. Audio Annotator is a simple web-based front end for strong labeling of audio with standard audio visualizations [41]. Although a good starting point, its functionality is limited, and it is not easily extensible. Freesound Datasets is a new web-based platform for crowdsourcing weak labels of audio, hosted on https://freesound.org [45]. However, it is currently limited to Freesound data and also is 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.

First, we propose that the research community should develop and adopt standard, open-source tools for audio annotation.

Data set file formats

As described in the "System Architecture and Component" section, standardized tools for reading and writing data file formats minimize the risk of parsing errors and ease distribution and use of data. There are several formats for encoding music annotations (MusicXML [12], MEI [13], MPEG-7 [14], and 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 data sets, 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 data set needs to be cleaned to remove erroneous annotations or outliers, then users could write a clean view of the data without discarding information. Additionally, preregistered splits of the data could be implemented as a view on top of an existing view. Data set files would also contain standardized metadata and documentation of the data set creation process.

Data documentation

To understand the content of data sets, use them appropriately, and extend them when necessary, data sets must be thoroughly documented. This motivates researchers to develop standard reporting mechanisms and tools to facilitate the documentation of the data collection process. Although standards should be developed and ratified by the community, the following are possible items to include for each annotation:

- annotation software and version
- annotation software configuration
- description of all tasks, including participant screening, training, annotation tasks, and surveys
- description of annotator recruiting
- monetary (or other extrinsic) compensation mechanisms
- anonymized annotator identifiers
- time stamps
- data cleaning or processing procedures
- data synthesis procedures description and code (if applicable).

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. As a community, we 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, although adoption by the community has been slow [47]. Recently, Gebru et al. [48] proposed a standardized data sheet format for general machine-learning data sets, inspired by the standardized data sheets that accompany electronic components.

Data distribution

Finally, we need tools to distribute, maintain, and index public data sets. Although many of the requirements for data are similar to those for software, data typically require more storage than software, rendering many existing services unsuitable. Data hosting should support versioning to support changes to data sets, provide digital object identifiers (DOIs), and guarantee longevity for several decades to prevent broken URLs and ephemeral data. These data requirements files would specify the data sets and versions required by software, and they should be distributed along with the software requirements files. There are currently several hosting solutions that support large data sets, versioning, and DOIs and guarantee decades of longevity (e.g., Zenodo at https://zenodo.org, Figshare at https://figshare.org, Dryad at https://datadryad.org, and Dataverse at https://dataverse.org). Unfortunately, these solutions have yet to develop a data management tool like we have 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 data sets. However, this could also double as a platform for proposing and discussing the creation of new data sets. Although 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.

Conclusions

Although MIR has long been data driven and necessarily complex because of 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 the "System Architecture and Components" section is generic enough to capture most MIR use cases, and, although different domains might exhibit slightly different workflows, we expect that the overall system complexity issue will arise across domains. The recommendations put forward in the "Best Practices for OSS Research" section should serve as a solid basis for improving the quality and reproducibility of scholarly research. While 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. Our proposal is intended to resolve certain shortcomings in our current practices for data set construction, but it may be readily adapted to different application domains. We encourage future researchers to think carefully about data construction, preservation, and management issues moving forward.

Authors

Brian McFee (brian.mcfee@nyu.edu) received his B.S. degree in computer science from the University of California, Santa Cruz, in 2003 and his M.S. and Ph.D. degrees in computer science and engineering from the University of California, San Diego, in 2008 and 2012, respectively. He is an assistant professor of music technology and data science at New York University. His work lies at the intersection of machine learning and audio analysis. He is an active open-source software developer and the principal maintainer of the librosa package for audio analysis.

Jong Wook Kim (jongwook@nyu.edu) received his B.S. degree in electrical engineering from the Korea Advanced Institute of Science and Technology, Daejeon, and his M.S. degree in computer science and engineering from University of Michigan in 2009 and 2011, respectively. From 2012 to 2015, he was a back-end software engineer at NCSOFT Corporation and Kakao Corporation, South Korea, and he was a research scientist intern at Pandora in 2017 and Spotify in 2018, focusing on music recommender systems and neural music synthesis. He is a Ph.D. candidate at New York University's Music and Audio Research Laboratory. His research interests include automatic music transcription and music language models.

Mark Cartwright (mark.cartwright@nyu.edu) received a B.M. degree in music technology from Northwestern University, Evanston, Illinois, in 2004. He received an M.A. degree in music science and technology in 2007 from Stanford University (CCRMA), California, and a Ph.D. degree in computer science in 2016 from Northwestern University, where his research focused on developing new interaction paradigms for audio production tools. Currently, he is a postdoctoral researcher at New York University's Music and Audio Research Laboratory. He was previously a visiting researcher at the Center for Digital Music at Queen Mary University of London and an intern at Adobe's Creative Technology Lab. His research lies at the intersection of human–computer interaction, audio signal processing, and machine learning.

Justin Salamon (justin.salamon@nyu.edu) received his B.A. degree in 2007 in computer science from the University of Cambridge, United Kingdom, and his M.Sc. and Ph.D. degrees in computer science from the Universitat Pompeu Fabra, Barcelona, Spain, in 2008 and 2013, respectively. In 2011, he was a visiting researcher at the Institut de Recherche et de Coordination Acoustique/Musique, Paris, France. In 2013, he joined New York University as a postdoctoral researcher, where he has been a senior research scientist since 2016. He is a senior research scientist at New York University's Music and Audio Research Laboratory and Center for Urban Science and Progress. His research focuses on the application of machine learning and signal processing to audio signals, with applications in machine listening, music information retrieval, bioacoustics, environmental sound analysis, and open-source software and data.

Rachel Bittner (rachelbittner@spotify.com) received her B.S. degrees in music performance and math at the University of California, Irvine, her B.M. degree in math at New York University's Courant Institute, and her Ph.D. degree in music technology in 2018 at the Music and Audio Research Lab at New York University under Dr. Juan Pablo Bello. She was a research assistant at NASA Ames Research Center, working with Durand Begault in the Advanced Controls and Displays Laboratory. She is a research scientist at Spotify in New York City. Her research interests are at the intersection of audio signal processing and machine learning, applied to musical audio. Her dissertation work applied machine learning to fundamental frequency estimation.

Juan Pablo Bello (jpbello@nyu.edu) received his B.Eng. degree in electronics in 1998 from the Universidad Simón Bolívar in Caracas, Venezuela, and in 2003 he received his Ph.D. degree in electronic engineering from Queen Mary University of London. He is a professor of music technology and computer science and engineering at New York University. His expertise is in digital signal processing, machine listening, and music information retrieval, topics that he teaches and on

which he has published more than 100 papers and articles in books, journals, and conference proceedings. He is the director of the Music and Audio Research Lab, where he leads research on music informatics. His work has been supported by public and private institutions in Venezuela, the United Kingdom, and the United States, including Frontier and CAREER Awards from the National Science Foundation and a Fulbright scholar grant for multidisciplinary studies in France. He is a Senior Member of the IEEE.

References

[1] G. Tzanetakis and P. Cook, "Musical genre classification of audio signals," *IEEE Trans. Speech and Audio Processing*, vol. 10, no. 5, pp. 293–302, 2002. doi: 10.1109/TSA.2002.800560.

[2] B. L. Sturm, "Revisiting priorities: Improving MIR evaluation practices," in *Proc. 17th Int. Society for Music Information Retrieval Conf., (ISMIR),* New York, 7–11 Aug. 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," presented at the Sound and Music Computing Conf., 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 *Proc. 15th Int. Society for Music Information Retrieval Conf., (ISMIR),* Taipei, Taiwan, 27–31 Oct. 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 Psychological Sci.*, vol. 7, no. 6, pp. 528–530, 2012.

[6] P. Vandewalle, J. Kovacevic, and M. Vetterli, "Reproducible research in signal processing," *IEEE Signal Processing Mag.*, vol. 26, no. 3, pp. 37–47, 2009.

[7] J. B. Buckheit and D. L. Donoho, "Wavelab and reproducible research," in *Wavelets and Statistics*, A. Antoniadis, G. Oppenheim, and B. McFee, Eds. New York: Springer, 1995, pp. 55–81.

[8] M. Owens and G. Allen, *SQLite*. New York: Springer-Verlag, 2010.

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

[10] F. Bellard, M. Niedermayer, et al. (2012). Ffmpeg. [Online]. Available: http://ffmpeg.org.

[11] E. de Castro Lopo. (2011). Libsndfile. [Online]. Available: http://www.mega-nerd.com/libsndfile/

[12] M. Good, "MusicXML for notation and analysis," *Virtual Score: Representation, Retrieval, Restoration*, vol. 12, pp. 113–124, 2001.

[13] P. Roland, "The music encoding initiative (MEI)," in *Proc. First Int. Conf. Musical Applications Using*, 2002, pp. 55–59.

[14] S.-F. Chang, T. Sikora, and A. Purl, "Overview of the MPEG-7 standard," *IEEE Trans. Circuits Syst. Video Technol.*, 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 *Proc. 15th Int. Society for Music Information Retrieval Conf., (ISMIR),* Taipei, Taiwan, 27–31 Oct. 2014, pp. 591–596.

[16] B. McFee, E. J. Humphrey, and J. P. Bello, "A software framework for musical data augmentation," in *Proc. 16th Int. Society for Music Information Retrieval Conf., (ISMIR),* Málaga, Spain, 26–30 Oct. 2015, pp. 248–254.

[17] M. Mauch and S. Ewert, "The audio degradation toolbox and its application to robustness evaluation," in *Proc. 14th Int. Society for Music Information Retrieval Conf., (ISMIR),* Curitiba, Brazil, 4–8 Nov. 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," presented at the Workshop on Applications of Signal Processing to Audio and Acoustics, New Paltz, NY, Oct. 2017.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, et al. "Scikit-learn: Machine learning in Python," *J. Mach. Learning Res.*, vol. 12, pp. 2825–2830, Oct. 2011.

[20] B. Whitman, G. Flake, and S. Lawrence, "Artist detection in music with minnowmatch," in *Proc. 2001 IEEE Signal Processing Society Workshop*, 2001, pp. 559–568.

[21] B. McFee, C. Jacoby, E. J. Humphrey, and W. Pimenta. (2018). Pescadores/pescador: 2.0.0. [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. (2015). Blocks and fuel: Frameworks for deep learning. arXiv. [Online]. Available: https://arxiv.org/abs/1506.00619

[23] D. Bogdanov, N. Wack, E. Gómez, S. Gulati, P. Herrera, O. Mayor, G. Roma, J. Salamon, et al., "Essentia: An audio analysis library for music information retrieval," in *Proc. 14th Int. Society for Music Information Retrieval Conf., (ISMIR),* Curitiba, Brazil, 4–8 Nov. 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 *Proc. 14th Python in Science Conf.*, 2015, pp. 18–25.

[25] P. Brossier. (2009). Aubio, a library for audio labelling. [Online]. Available: https://aubio.org/

[26] S. Böck, F. Korzeniowski, J. Schlüter, F. Krebs, and G. Widmer, "Madmom: A new Python audio and music signal processing library," in *Proc. 2016 ACM Multimedia Conf.*, 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. doi: 10.1017/S1355771800003071.

[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 *Proc. 15th Int. Society for Music Information Retrieval Conf., (ISMIR),* Taipei, Taiwan, 27–31 Oct. 2014, pp. 573–578.

[29] F. Chollet, et al. (2015). Keras. [Online]. Available: https://keras.io

[30] A. Mesaros, T. Heittola, and T. Virtanen, "Metrics for polyphonic sound event detection," *Appl. Sci.*, vol. 6, no. 6, p. 162, 2016. doi: 10.3390/app6060162.

[31] A. Said and A. Bellogín, "Rival: A toolkit to foster reproducibility in recommender system evaluation," in *Proc. 8th ACM Conf. Recommender Systems*, 2014, pp. 371–372.

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

[33] S. Böck, "oneset_db." Accessed on: Jan., 2018. [Online]. Available: https://github.com/CPJKU/onset_db

[34] D. P. Ellis. (2006). PLP and RASTA (and MFCC, and inversion) in MATLAB using melfcc.m and invmelfcc.m. [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, et al., "Hidden technical debt in machine learning systems," in *Proc. 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, 2017. doi: 10.1371/journal.pcbi.1005510.

[37] GitHub, Inc. No license. [Online]. Available: https://choosealicense.com/no-permission/

[38] K. Beck, *Test-Driven Development: By Example*. Reading, MA: Addison-Wesley, 2003.

[39] F. S. Chirigati, D. E. Shasha, and J. Freire, "ReproZip: Using provenance to support computational reproducibility," presented at the 5th USENIX Conf. Theory and Practice of Provenance (TAPP'13), 2013.

[40] B. L. Sturm, "An analysis of the GTZAN music genre dataset," in *Proc. 2nd Int. ACM Workshop on Music Information Retrieval With User-Centered Multimodal Strategies*, 2012, pp. 7–12.

[41] M. Cartwright, A. Seals, J. Salamon, A. Williams, S. Mikloska, D. MacCbonnell, E. Law, J. Bello, and O. Nov, "Seeing sound: Investigating the effects of visualizations and complexity on crowdsourced audio annotations," *Proc. ACM on Human-Computer Interaction*, vol. 1, no. 1, 2017. doi: 10.1145/3134664.

[42] Bioacoustics Research Program. (2014). Raven pro: Interactive sound analysis software (version 1.5). [Online]. Available: http://www.birds.cornell.edu/raven

[43] D. Mazzoni and R. Dannenberg. (2000). Audacity. Avaliable: https://www.audacityteam.org

[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," presented at the 1st Int. Conf. Technologies for Music Notation and Representation, 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 Int. Society for Music Information Retrieval Conf. (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 *Proc. AGU Fall Meeting Abstracts*, 2011.

[47] G. Peeters and K. Fort, "Towards a (better) definition of the description of annotated MIR corpora," in *Proc. 13th Int. Society for Music Information Retrieval Conf., (ISMIR),* Porto, Portugal, 8–12 Oct. 2012, pp. 25–30.

[48] T. Gebru, J. Morgenstern, B. Vecchione, J. W. Vaughan, H. Wallach, H. Daumeé III, and K. Crawford. (2018). Datasheets for datasets. arXiv. [Online]. Available: https://arxiv.org/abs/1803.09010

SP