Provenance Design and Evolution in a Production ML Library

Adam Pocock [1] Joseph Wonsil [2] Romina Mahinpei [3] Jack Sullivan [1] Margo Seltzer [2]

Abstract

Data Provenance is a formal record documenting how a digital artifact came to be in its present state. In the context of a Machine Learning model, provenance includes the data sources, data transformations, and algorithmic hyperparameters that are used to create the model. We present the design of Tribuo[1], an open-source, production ML library with integrated data provenance. Tribuo collects provenance automatically requiring no user action or intervention. Using the provenance data, we developed systems for reproducing ML models and generating model cards. Like a type-system, integrated provenance collection constrains design choices and provides utility in other parts of the system. Our integrated provenance approach has allowed us to automatically fix bugs in old models, detect non-obvious platform dependencies and deeply understand and debug models built by other groups. Integrating provenance collection into the library influences the design and evolution of the system, which requires making trade-offs among provenance fidelity, provenance size, and developer flexibility.

1. Introduction

Machine Learning models are integrated into many large pieces of software, providing critical features and functionality. Regulatory agencies frequently require understanding and tracking the behaviour of ML models, which is also necessary for developers to debug their systems. Consequently, there is much interest in data provenance and reproducibility in ML (Schelter et al., 2017; Gundersen & Kjensmo, 2018; Semmelrock et al., 2025). Data provenance (or lineage) is a record of how an object (i.e., dataset, model, evaluation) came to be (Carata et al., 2014). For ML models, this provenance must contain information about the training data, the training algorithm, model hyperparameters, and machine state. Together, this information provides the story of how the model was created, which is necessary for auditing and is the first step towards reproducing the model.

Provenance can be captured at different granularities depending on the requirements of the task, from fine-grain provenance that captures system calls to coarse-grain provenance that logs only the executable and source repository information (Carata et al., 2014). However the source code of the training program may not be sufficient as it does not fully capture the data & runtime environment (e.g., CPU/GPU, number of threads, library versions) from model creation. In Machine Learning, we are interested in the story of the model creation, so fine-grained provenance, capturing each individual matrix multiply or file open, hides that story. Similarly, noting that the user executed a jupyter notebook may not provide enough information to recreate the story.

Automatically capturing the provenance information with minimal user intervention tends to provide the best experience; relying on manual logging can lead to gaps in the data. Many existing provenance collection frameworks wrap other ML libraries to automatically extract provenance information. For example MLFlow (Chen et al., 2020) has two modes, either manual logging by users (similar to Weights and Biases (Biewald, 2020)) or "autolog", which modifies the Python code for each tracked library to record provenance information when executed. Autologging relies on parameter reporting by the training library, along with a deep understanding of how that library behaves so it can patch it to track appropriate behaviour. If there is no single tracking point (e.g., in PyTorch where users write their own training loop), then the autologging has no central point to intercept and track the user's actions, requiring manual logging. Integrating provenance collection into the ML library itself provides the greatest likelihood of capturing the relevant information with sufficient fidelity. Such integration allows the library itself to decide what information is pertinent and does not require the users of the library to deeply understand the library internals to capture its state.

[1]Oracle Labs, Burlington, MA, USA [2]Department of Computer Science, University of British Columbia, Vancouver, BC, Canada [3]Department of Computer Science, Princeton University, Princeton, NJ, USA. Correspondence to: Adam Pocock <adam.pocock@oracle.com>.

Proceedings of the ICML 2025 Workshop on Championing Open-source Development in Machine Learning (CODEML '25). Copyright 2025 by the author(s).

[1]Website: `https://tribuo.org`, GitHub: `https://github.com/oracle/tribuo`.

We discuss the design of the integrated provenance system in our open source Machine Learning library Tribuo. Tribuo is used in multiple Oracle products supplying ML functionality in enterprise SaaS applications with millions of records and also in large open source projects such as OpenSearch. These environments require that provenance collection is low overhead, to reduce cost, and accurate, to allow auditing of the model and application. Our approach evolved in response to the reality of our early internal users treating Tribuo as they would any other software library; sending bug reports to the development team without any information other than the model file itself and a rough description of the error. We found a small, accurate provenance object contained within the model itself provides a substantial amount of insight into the behaviour of the system to help debug the library and the surrounding code. This requires making a trade-off between provenance fidelity, provenance size, and developer flexibility. Given Tribuo's position as a production (rather than research) ML library, we chose to maximize fidelity and minimize provenance size with some cost to developer flexibility. In general, we encapsulate structures within the library; providing a simple API surface and wrapping things such as training loops, model random state, and third-party model implementations.

2. Design of the provenance system

Tribuo is an open source Java ML library; development started internally in 2016, and the first open source release followed in 2020. It's available under an Apache 2.0 license and developed in the open with contributions welcome from external users. It provides implementations of tabular ML algorithms such as trees, boosting, linear models, factorization machines, along with clustering algorithms, feature transformations, data ingestion pipelines, and evaluation methods for classification, regression, and clustering. In contrast to the fit/predict API popularised by scikit-learn (Pedregosa et al., 2011), which accepts ndarray inputs, we require strongly typed objects derived from our interfaces as inputs to the training and evaluation functions. This lets us begin provenance collection when data is loaded into our system.

The workflow to build and use a model consists of the following four steps. 1) Create a *Dataset* object from a data source (e.g., a file, database connection, in-memory representation). 2) Create a *Trainer* object based on a set of hyperparameters. 3) Create a *Model* by passing a Dataset to a Trainer's train method. 4) Evaluate the Model by first creating an *Evaluator* object for the prediction type (e.g., a LabelEvaluator for classification) and then invoking the Evaluator's evaluate method, passing in a Model and Dataset, to create an *Evaluation* object containing the relevant results.

Each of these objects contains a provenance function that returns a description of the object and its state. The provenance is a tree of immutable objects; for example Model provenance includes Dataset provenance and Trainer provenance. In turn the Trainer provenance can contain other Trainer provenances (e.g., a bagged ensemble trainer contains a provenance object for both the ensemble and the weak learner's trainer). Each Model and Evaluation contains an immutable provenance object, as they themselves are immutable. Datasets and Trainers generate provenance objects on request, as they contain mutable state such as the current list of dataset transformations and the state of the trainer's internal Pseudo-Random Number Generator (PRNG). Importantly, the Dataset provenance contains metadata only about the original data source, e.g., location, hash, timestamp, feature extraction information, but not the data itself. A Model provenance contains a Trainer provenance, a train Dataset provenance and system metadata such as the OS version. An Evaluation provenance contains both a Model provenance and a test Dataset provenance. A full model provenance is given in Appendix A. The types are extensible, so new algorithms can be added provided the developer implements a provenance object. This restriction ensures that everything is tracked.

Integrating the provenance collection in this way shifts the burden of collection from the user of the library to the developers of the library. The library developer needs to capture all the relevant user controlled state in an object, along with the object's type and any relevant environmental state such as the CPU architecture or OS. We capture library version information automatically as the implementations all refer to a core class which holds the version information. This doesn't capture source control information as we expect production use cases to use a tagged version number which is recorded in the Tribuo version class. Our library was developed on top of a configuration system that we use to automate provenance collection. Object fields annotated with `@Config` can be automatically inserted or read. At provenance collection time (i.e., when a model is trained or the data is modified), the library automatically reflectively iterates over the object's annotated fields to capture initial provenance, which the implementation can augment with more metadata such as timestamps and RNG state. Environmental state such as the CPU architecture, OS version, and Java version are captured during construction of the model provenance classes. This can be expanded to collect more information from the operating system process such as the environment variables, but we don't yet collect this information, as it is more likely to include sensitive information the user wouldn't expect to be captured. As Tribuo is designed for training models on a single node it doesn't have any hooks to collect cluster level information. However, there is an additional runtime hook for the user of the library to add additional provenance information into any given training run. We use this functionality to add external links to a model, such as to a bug tracking system, so the provenance object can direct users to other relevant places for information about the model. The runtime provenance hook also provides a location to insert version control information like a git hash, which is particularly useful during development when extending Tribuo with new classes outside the core library.

Tribuo is integrated into large enterprise applications, where privacy and security of user data is paramount. A model's provenance might contain private information, e.g., recognizable training data or details of the feature transformation pipeline. If the model is released publicly or deployed at another customer site, then the private information in the provenance must be removed. We can remove provenance from a model and replace it with a hash value computed from that provenance; this allows us to track the model while simultaneously keeping private information in a secure location. In practice many more models are created during development than are deployed into less secure production locations, so making provenance collection the default behavior proves useful.

PRNG state is an important facet of an ML system, which we need to track to accurately construct model provenance. Tracking the full PRNG state is expensive, so we do not expose any internal random states to users, but instead ensure that we can easily compute the current state given the PRNG's seed. We use splittable PRNGs (Steele Jr et al., 2014), which allow the construction of a new PRNG by splitting it from an existing one. Thus, we need only store the initial seed and the PRNG's split history in provenance, as that is sufficient to reconstruct its random state (Wonsil et al., 2023). This is similar to how JAX uses splittable functional PRNGs (Bradbury et al., 2018).

3. Consequences of designing for provenance

Our initial attempt at provenance in Tribuo consisted of text representing the dataset and trainer parameters. Incorporating provenance information made it much simpler to diagnose and debug issues without increasing the burden on users. The text-based system, while useful, was clearly deficient as it was not machine parsable without substantial effort, so in 2018 we started development of the provenance system described in Section 2. Our early focus on provenance substantially shaped the development of the library. The provenance acts as an ever-present stack trace, describing how a trained model came to be, which is helpful for debugging issues inside the library and in user code. This vastly improved the developer experience when addressing issues in model training or deployment, as the model can be introspected without access to the user's source code.

A single trainer object can be used concurrently to train multiple models (e.g., training each tree in a random forest). Each training job needs an independent source of randomness, so we sequentially split out PRNGs from the trainer's base PRNG. This implicitly gives an ordering to the collection of trained models, which needs to be respected when reproducing the ensemble. Similar considerations are necessary for internal parallelization within a single training run; otherwise, the provenance does not allow for perfect reproducibility. However our goal is accurate provenance, not necessarily perfect reproduction, so Tribuo has a mixture of models that reproduce exactly (on the same hardware/runtime) and those that produce extremely similar models but differ due to order-of-operation differences when using floating point arithmetic. The latter case is true for many other ML libraries, as it is difficult to produce deterministic behaviour from parallel accelerators such as GPUs.

Consistently available provenance information across library versions allows transparent bug fixing when users load old models. An older version of Tribuo had a bug where multi-dimensional regression outputs would have incorrect indices due to incorrect traversal order in the mapping. The output dimensions were permuted, though each individual output was computing the correct function. This bug affected the serialized forms of the models; they were corrupted on disk in a way that required rewriting the model on load. When fixing this bug, we found we could automatically correct older models during deserialization, because the provenance records the library version and the trainer hyperparameters (only some settings triggered the bug), so the fix could be selectively applied only to the corrupted models. Therefore all users had to do was load their model in a newer version of the library, then save it again to permanently fix the model. This does raise the question of what to record in the provenance for this kind of fix, as the model has changed; at the moment, we merely add a marker field denoting the fix.

Integrating with external libraries is more complicated when provenance capture is required. No single machine learning library can include all algorithms of interest to users, so many libraries either provide a mechanism for wrapping third-party libraries, like Amazon's DJL (djl, 2019), or expose a simple interface for others to implement like scikit-learn's fit/predict (Pedregosa et al., 2011). We chose the former method, as we do not expect our API to become universal in the way scikit-learn's has become, so we need to automatically capture provenance information from those third-party libraries without relying upon them collecting it for us. In practice this means the third-party library needs to be completely wrapped in our API, preventing users from directly calling its training methods. If the library exposes configuration objects (e.g., LibSVM (Chang & Lin, 2011)), then these need to be parseable by our code, so they can be decomposed into primitives suitable for provenance collection. In some cases, this can be quite restrictive. It is hard to automatically capture concise provenance from a library that requires users to write their own gradient descent loop like PyTorch does (Paszke et al., 2019). Consequently, our integration with deep learning libraries provides a simplified API, similar to Keras's fit method, with the restriction that users cannot write custom training loops. This is similar to the issues MLFlow has with automatically capturing provenance information, except we expose only those methods from which we can capture provenance rather than having a potentially lossy capture, if the user steps outside the supported API.

Refactoring code is a natural part of development, though many libraries take care to conform to some kind of semantic versioning for their public API, ensuring minimal changes across versions. Provenance information is another public API surface (as is a model's serialized form), so it is important to consider when a refactoring would modify provenance information. For example, our original termination criterion for SGD training was the step count. Transforming this to a more flexible system in which the user can supply different termination criterion functions leads to two provenances, one from before the switch with a simple integer field, and one afterwards with a provenance object representing the termination criterion (which may just contain a step count). That raises the question of how to treat old provenance. Should it be silently upgraded to new style provenance, containing a criterion object with a step count, or should the step count field be left to support old models? Never removing a provenance field guarantees that the hashes computed on it remain valid. However, the vestigial field requires keeping the old code path alive, both in the training code and the provenance code to enable reproducibility from old provenance, or it requires implementing special cases in the reproducibility framework. We've not found a clear answer to this kind of question, but it is representative of the things we need to consider when evolving the provenance.

We chose to build Tribuo in Java as our goal was to build a ML library for use in large enterprise applications, which are typically written in Java. We could have built the library in another language and exported the models for deployment in Java, but this would preclude usecases where the application wants to train a model, and the export process would need to preserve the provenance information. While the ML ecosystem in Java is smaller, in practice many algorithms and libraries have Java bindings, and in some cases we've written and open sourced new bindings for specific libraries that we required such as ONNX Runtime (ONNX Runtime developers, 2020). There have been several benefits of building a provenance system in a statically typed language like Java, principally in the way it allows the developer to control how the system is extended, and to enforce compile time constraints. It's not possible for a Tribuo developer to accidentally put a string in a timestamp field as that field is typed in the model's provenance. Furthermore the runtime type checking means that the provenance objects are validated for type errors on load. This makes it much simpler to build a provenance system, and also to rely upon the output in downstream tasks. Rapid iteration on a research idea is slightly slower in such a system, both due to the type system and also due to the requirement to think about provenance capture while implementing the idea, but as our focus is on production environments where users are unlikely to change the model implementation itself this is not a large problem for our use case.

4. Provenance Use Cases

We cannot validate the correctness of provenance systems without downstream usecases, preferably ones which use the provenance information in an automated process. We now discuss two provenance-based use cases that are built into the library.

4.1. Automated and Integrated Reproducibility

Reproducing ML models is a complicated problem, even with the detailed provenance captured by Tribuo. However the ability to reproduce a model is important, as it validates that no changes have happened to the data or the training pipeline, and it provides a way of verifying provenance information correctness for auditing purposes. It's also a starting point for systems that let users explore hyperparameter and modelling choices; without reliable re-execution, the effect of any given hyperparameter cannot be isolated from other uncaptured randomness in the system.

Building a reproducible system requires three steps. First, collect sufficient information about the system to allow reproduction, including data sources, transformations, hyperparameters, and training algorithms used. Second, reassemble and appropriately configure the training pipeline from the information in the initial training run. Third, execute that pipeline on the same data in a similar compute environment. The similarity of the compute environment is necessary to ensure that the reproduced model does not diverge too much from the original model (Zhuang et al., 2022).

All the information required for this is already collected by Tribuo's provenance system. The configuration system provides the mechanism for constructing individual components such as the Trainer, though the user needs to tie them together. Executing the rebuilt pipeline produces a model, which can then be evaluated alongside the original, and the two Model provenances can be compared directly. This system allowed us to diagnose that some training algorithms are sensitive to the Java version, due to differences in the implementation of the `exp` function. Tribuo's reproducibility system is described in more detail in Wonsil et al. (2023).

4.2. Model Card Generation

As machine learning becomes more integrated with other systems, it becomes more important to accurately and consistently describe each model. Model Cards (Mitchell et al., 2019) provide a well understood template for writing model descriptions and use cases. A model card contains the information about how a model was produced, data sources and their licenses, any relevant bias or fairness analysis, intended uses, and performance analysis. Tribuo's provenance captures model training details, evaluation performance, and the training and evaluation data. We automatically populate those parts of the model card, allowing users to focus on the parts external to ML software, such as licensing, intended uses, citation details, and contact information.

5. Conclusions

Building an ML library with integrated provenance changes how the library is written in much the same way a language's type system changes the way a program is written. It constrains some areas, while enabling new or more reliable functionality elsewhere (such as in the bug fixing example). As provenance is a core part of the library, it makes it easier to build functionality on top of it. The reproducibility and model card systems were added after the provenance system was completed. In theory, adding reproducibility would require no changes to the existing system. In practice, implementing these features uncovered bugs in the pre-existing infrastructure, reinforcing the necessity of having executable usecases for provenance as aids to validating its correctness.

There are several desirable features of a provenance system that are in tension: accurate provenance collection, small/efficient provenance objects, and the amount of user control. In Tribuo we've focused on the first two at the expense of the third, consequently, our provenance collection is extremely accurate allowing perfect reproducibility, and the provenance objects are small and quickly collected. The overhead of our provenance collection is minimal, taking less than a millisecond to capture provenance across a wide range of experiments. The price for this decision comes in user flexibility. Tribuo must remain in control of the training loop and cannot expose the training functions of third-party libraries, as this would allow users to construct models with inaccurate provenance. This choice imposes more development cost on the developers of Tribuo as integrating new algorithms is harder, but from a user perspective everything has reliable and consistent provenance with no additional effort on their part.

Some libraries have added provenance in pipeline systems on top of their core algorithmic functionality (e.g., TensorFlow (Abadi et al., 2016), with TensorFlow-Extended and ML MetaData (TFE)). Automatically capturing provenance in TensorFlow is tricky, as it allows users to build their own data pipeline and training loop without using TensorFlow's types, limiting the ability of the TensorFlow API to track how data flows through the system. Using TFE requires the user to build more of the system with TF's types, capturing the input and output locations as part of the pipeline. In this way, the approach is similar to Tribuo's, restricting user flexibility to allow provenance capture. Automatic provenance collection for more flexible systems such as scikit-learn is possible but requires either deep instrumentation of the language runtime (e.g., Phani et al. (2021)) or capturing user calls before they reach the library as in MLFlow. In the former case, the provenance information produced is too fine-grained for easy use (tracking individual matrix operations leads to large provenance traces missing the higher level semantic information), and the latter case relies upon the provenance library keeping pace with the ML library.

References

DJL, 2019. URL `http://github.com/deepjavalibrary/djl`.

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: a System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 265–283, USA, 2016. USENIX Association. ISBN 9781931971331.

Biewald, L. Experiment tracking with weights and biases, 2020. URL `https://www.wandb.com/`. Software available from wandb.com.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/jax-ml/jax`.

Carata, L., Akoush, S., Balakrishnan, N., Bytheway, T., Sohan, R., Seltzer, M., and Hopper, A. A primer on provenance. *Communications of the ACM*, 57(5):52–60, 2014.

Chang, C.-C. and Lin, C.-J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

Chen, A., Chow, A., Davidson, A., DCunha, A., Ghodsi, A., Hong, S. A., Konwinski, A., Mewald, C., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., Singh, A., Xie, F., Zaharia, M., Zang, R., Zheng, J., and Zumar, C. Developments in mlflow: A system to accelerate the machine learning lifecycle. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, DEEM '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380232. doi: 10.1145/3399579.3399867. URL https://doi.org/10.1145/3399579.3399867.

Gundersen, O. E. and Kjensmo, S. State of the art: Reproducibility in artificial intelligence. In *Thirty-second AAAI Conference on Artificial Intelligence*, 2018.

Mitchell, M., Wu, S., Zaldivar, A., Barnes, P., Vasserman, L., Hutchinson, B., Spitzer, E., Raji, I. D., and Gebru, T. Model cards for model reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency*, pp. 220–229, 2019.

ONNX Runtime developers. Onnx runtime, 2020. URL https://onnxruntime.ai/. Version: 1.4.0.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: an Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

Phani, A., Rath, B., and Boehm, M. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *Proceedings of the 2021 International Conference on Management of Data*, pp. 1426–1439, 2021.

Schelter, S., Böse, J.-H., Kirschnick, J., Klein, T., and Seufert, S. Automatically tracking metadata and provenance of machine learning experiments. In *NeurIPS workshop on ML Systems*, 2017.

Semmelrock, H., Ross-Hellauer, T., Kopeinik, S., Theiler, D., Haberl, A., Thalmann, S., and Kowald, D. Reproducibility in machine-learning-based research: Overview, barriers, and drivers. *AI Magazine*, 46(2):e70002, 2025. doi: https://doi.org/10.1002/aaai.70002. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/aaai.70002.

Steele Jr, G. L., Lea, D., and Flood, C. H. Fast splittable pseudorandom number generators. *ACM SIGPLAN Notices*, 49(10):453–472, 2014.

Wonsil, J., Sullivan, J., Seltzer, M., and Pocock, A. Integrated reproducibility with self-describing machine learning models. In *Proceedings of the 2023 ACM Conference on Reproducibility and Replicability*, pp. 1–14, 2023.

Zhuang, D., Zhang, X., Song, S., and Hooker, S. Randomness in neural network training: Characterizing the impact of tooling. *Proceedings of Machine Learning and Systems*, 4:316–336, 2022.

A. Appendix: Full Provenance Listing

We show the model provenance for a logistic regression trained using stochastic gradient descent with AdaGrad on the Iris dataset. The provenance is rendered as JSON for ease of reading, though it is natively a tree of objects. We've split it into two chunks, Listing 1 shows the model provenance excluding the data provenance, and Listing 2 shows the data provenance.

The top level JSON object is the model provenance, it contains metadata fields like the Java version, OS, CPU architecture, Tribuo version, and timestamps. Inside the model provenance are fields for the trainer and the dataset, which repeat some fields like the timestamps and library version to ensure they stand alone. The trainer provenance records the gradient optimizer, the optimizer parameters, the training objective, and other ancillary training parameters like the number of training epochs and the logging frequency. The dataset provenance records the number of features, the number of training examples, the number of output classes, the data source and any transformations applied to the dataset. The source provenance is from a class that generates train/test splits from another data source, and it records the seed of the RNG, the size of the initial data source, the fraction of data used for training, if this portion of data is the training or test split, and the original data source provenance (which is extracted and presented in Listing 2). The underlying source provenance records the SHA-256 hash of the source file, the data processors which were used to load the CSV file and featurize it, the file creation timestamp, the CSV separator and quote characters, and the file path on disk.

Listing 1. Model provenance for a logistic regression trained on Irises. The data provenance is in Listing 2, it would appear in the "source" key. Note for review that the library name has been removed from the class names.

```
{
  "class-name" : "org.tribuo.classification.sgd.linear.LinearSGDModel",
  "dataset" : {
    "class-name" : "org.tribuo.MutableDataset",
    "datasource" : {
      "class-name" : "org.tribuo.evaluation.TrainTestSplitter",
      "is-train" : "true",
      "seed" : "1",
      "size" : "150",
      "source" : { see Listing 2 },
      "train-proportion" : "0.7"
    },
    "is-dense" : "true",
    "is-sequence" : "false",
    "num-examples" : "105",
    "num-features" : "4",
    "num-outputs" : "3",
    "transformations" : [ ],
    "tribuo-version" : "4.3.2"
  },
  "instance-values" : { },
  "java-version" : "24",
  "os-arch" : "aarch64",
  "os-name" : "Mac OS X",
  "trained-at" : "2025-05-23T16:24:39.159986-04:00",
  "trainer" : {
    "class-name" : "org.tribuo.classification.sgd.linear.
        LogisticRegressionTrainer",
    "epochs" : "5",
    "host-short-name" : "Trainer",
    "is-sequence" : "false",
    "loggingInterval" : "1000",
    "minibatchSize" : "1",
    "objective" : {
      "class-name" : "org.tribuo.classification.sgd.objectives.LogMulticlass",
      "host-short-name" : "LabelObjective"
    },
    "optimiser" : {
      "class-name" : "org.tribuo.math.optimisers.AdaGrad",
      "epsilon" : "0.1",
      "host-short-name" : "StochasticGradientOptimiser",
      "initialLearningRate" : "1.0",
      "initialValue" : "0.0"
    },
    "seed" : "12345",
    "shuffle" : "true",
    "train-invocation-count" : "0",
    "tribuo-version" : "4.3.2"
  },
  "tribuo-version" : "4.3.2"
}
```

Listing 2. Data source provenance, extracted from Listing 1's "source" key in the model's provenance. Additionally, due to space concerns we removed three "DoubleFieldProcessors", that are the same as the "petalLength" processor given here but for the fields "petalWidth", "sepalWidth", and "sepalLength". This is due to the way provenance is captured in Tribuo as it occurs after the processors have been expanded and bound to each field name in the data file, even though the developer only specified they wanted to load float valued features and the file had 4 columns.

```
{
  "class-name" : "org.tribuo.data.csv.CSVDataSource",
  "dataPath" : "/Users/tribuo/development/bezdekIris.data",
  "datasource-creation-time" : "2025-05-23T16:24:38.883745-04:00",
  "file-modified-time" : "2025-05-23T16:24:30.283-04:00",
  "headers" : [ "sepalLength", "sepalWidth", "petalLength", "petalWidth", "
     species" ],
  "host-short-name" : "DataSource",
  "outputFactory" : {
    "class-name" : "org.tribuo.classification.LabelFactory"
  },
  "outputRequired" : "true",
  "quote" : "\"",
  "resource-hash" : "0FED2A99DB77EC533A62DC66894D3EC6DF3B58B6A8F3CF4A6B47E4086B7F
     97DC",
  "rowProcessor" : {
    "class-name" : "org.tribuo.data.columnar.RowProcessor",
    "featureProcessors" : [ ],
    "fieldProcessorList" : [ {
      "class-name" : "org.tribuo.data.columnar.processors.field.
         DoubleFieldProcessor",
      "fieldName" : "petalLength",
      "host-short-name" : "FieldProcessor",
      "onlyFieldName" : "true",
      "throwOnInvalid" : "true"
    }, ... ],
    "host-short-name" : "RowProcessor",
    "metadataExtractors" : [ ],
    "regexMappingProcessors" : { },
    "replaceNewlinesWithSpaces" : "true",
    "responseProcessor" : {
      "class-name" : "org.tribuo.data.columnar.processors.response.
         FieldResponseProcessor",
      "defaultValues" : [ "" ],
      "displayField" : "false",
      "fieldNames" : [ "species" ],
      "host-short-name" : "ResponseProcessor",
      "outputFactory" : {
        "class-name" : "org.tribuo.classification.LabelFactory"
      },
      "uppercase" : "false"
    },
    "weightExtractor" : {
      "class-name" : "org.tribuo.data.columnar.FieldExtractor"
    }
  },
  "separator" : ","
}
```