How to program machine learning in Java with the Tribuo library

Tribuo is an open source ML library designed for business applications—and for interoperability with many popular ML platforms.

by Adam Pocock

February 26, 2021

Machine learning (ML) is a process whereby software learns from data. ML can be an undirected process, such as finding clusters of similar data points, or a directed process, such as recognizing handwritten digits after being given examples of each digit. These two forms are called unsupervised learning and supervised learning, respectively.

With unsupervised learning tasks such as clustering or anomaly detection, the aim is to extract information from a data set without guidance from a user about what to look for.

With supervised learning tasks such as classification (predicting labels) or regression (predicting numerical values), the aim is to learn a function that maps the input data (also known as features) to the predicted output value.

One popular subfield of ML is known as deep learning, which is concerned with taking raw data inputs and automatically learning multiple layers of features, each more refined than the last, before making the final prediction. Deep learning powers the image recognition systems inside smartphone cameras and the speech recognition systems inside personal assistants such as Alexa, Siri, and Google Assistant, among many other use cases.
In ML, the function that maps inputs to outputs is called a \textit{model}, and the process of creating that model is called \textit{training}. A model can be as simple as an if-else statement with a learned threshold value or as complex as 100 million floating-point numbers across a chain of matrices 150 layers deep. Consequently, the training algorithms that produce these models can be quick to run on a single CPU core or take many thousands of GPU hours depending on the type of model, the amount of training data available, and the complexity of the problem being solved.

The work of ML, and more broadly of data science, is commonly associated with programming languages such as Python and with popular packages such as \textit{scikit-learn}, \textit{TensorFlow}, and \textit{PyTorch}. While much academic research and lots of exploratory analysis happens in those systems, in many cases, the final model needs to be deployed as part of a larger software system written in Java.

\textbf{How does software interact with models?}

Conceptually, you can think of most machine learning models as functions that accept an array (or a multidimensional array) of floats and produce a number (or an array of numbers).

The input arrays are the features: the pixels in an image, values from a database table, ID numbers representing words in a sentence, and so on. The output values can be cluster IDs, class label IDs (such as \(0 \rightarrow \text{"dog"}, 1 \rightarrow \text{"cat"}, \text{and} 2 \rightarrow \text{"horse"}\)), floating-point outputs for regression tasks, or other numeric types. This view of the world is mathematical and suitable for the \textit{MATLAB} programs that many ML algorithms were first implemented in.

Unfortunately, many modern ML packages haven’t moved on from this view of the world. They accept multidimensional arrays of floats and produce arrays of floats or integers, irrespective of the task that is being solved. In that world, it’s the developer’s problem to convert data into the appropriate features and to convert the outputs into a useful value for the application, mapping the numbers into useful objects.

This approach makes it tricky to track what each model is, because the inputs and outputs are the same types no matter what the task is. This burden increases for each additional model that needs to be incorporated into the system, because it provides yet another feature space with its IDs and another output space for whatever is being predicted.

The team at Oracle Labs thought it was unfair to leave all these tasks up to developers who want to use ML when the tasks really should be the job of the ML library that’s being used. This was part of the motivation for creating Tribuo.

\textbf{Introducing Tribuo}
Tribuo is a new open source library written in Java from Oracle Labs’ Machine Learning Research Group. It’s been used inside Oracle in production for several years, and Oracle open sourced it in September 2020 on GitHub under an Apache 2.0 license.

The team’s goal for Tribuo is to build an ML library for the Java platform that is more in line with the needs of large software systems. Tribuo operates on objects, not primitive arrays, Tribuo’s models are self-describing and reproducible, and it provides a uniform interface over many kinds of prediction tasks.

For Tribuo, the team decided to lean into the benefits of type systems. All Tribuo models produce typed outputs that know if the value they encode is a cluster ID, a regression output, a classification output, and so on. Not only does this make it simpler to consume the values that are produced, but it also makes it harder to use them incorrectly. It’s not possible to misinterpret a Tribuo classification prediction for a regression prediction, because they are two different types and the models themselves are typed using the prediction class they produce.

In addition to the type system enforcing things at the object level, each Tribuo model contains its own feature and output domains. You can inspect them to discover what features a model knows about, what values those features took at training time, and the number and kind of outputs the model can produce. Knowing these domains is important because it allows the model to automatically reject features that it has never seen before, rather than silently producing an unexpected or uncertain output.

**Tribuo’s enterprise-friendly focus and provenance system**

Building ML systems to solve enterprise problems gives Tribuo a different focus than that of many other ML libraries. Enterprises usually operate on database tables and text. Consequently, Tribuo assumes all the feature spaces it is operating in are sparse and any features that are missing are implicitly zero. This sparsity is useful both in natural language processing (NLP) tasks (because each document contains only a subset of the possible words/features) and in more-general enterprise tasks where features based on joins between different tables are conditionally extracted.

At Oracle Labs, the Tribuo team frequently found that most ML problems require a combination of the two types of data. For example, different customers may have been exposed to different kinds of marketing, and so incorporating the marketing features for all the kinds of marketing will dramatically increase the feature space even though each customer has used only a small subset of the marketing features. An implicitly sparse feature space means a huge number of zeros don’t need to be kept in memory to train a model, and there’s no need to worry about dealing with unexpected features at test time.
One final area where Tribuo is very different from other ML libraries is in its provenance system.

Commonly, ML models are built after multiple experimental runs with different parameters, data sets, and other modeling choices. Tracking all these runs is usually left to an external system such as MLflow or Weights & Biases. This means that the model itself doesn’t store any information about how it was created, and it relies upon external tracking to link things up.

When the Tribuo team built ML systems, they found that internally tracking which model went into production added an extra layer of complexity, because diagnosing ML issues is even trickier when the model training information lives in yet another system.

To mitigate this issue, the team built an extensive provenance system into Tribuo. Each data set tracks where the data was loaded from, how it was loaded in, and what transformations were applied after loading. Then the models record that data set provenance along with all the training parameters (such as the algorithm used and that algorithm’s parameters). Then the performance evaluations of models track the provenance of the model and the evaluation data. All of these provenances are serialized inside the models themselves, ensuring the information is always there when it’s needed.

The provenance isn’t just useful for tracking models; it also integrates with Tribuo’s configuration system. Tribuo’s training runs can be configured using a variety of configuration formats, using dependency injection to build the appropriate training algorithm.

The team added functionality to convert a model’s provenance into a configuration file, allowing users to repeat any model’s training procedure exactly. These configurations can be tweaked to use different training parameters or source data sets, as necessary, which the team can use to test tweaks to a model to address a customer’s need.

This coupling of provenance and configuration, along with Tribuo’s strong focus on experimental reproducibility, makes it a good platform for the experimental part of building an ML system—in addition to its obvious uses for tracking and replicating deployed models.

Why Tribuo was built in Java

Oracle has invested heavily in Java, and as part of Oracle Labs’ mission to transfer new technologies into Oracle’s products, the Tribuo team has been working on ML on the JVM for some time.

The team thought that instead of integrating dynamic languages into Java application stacks, it would be simpler to bring the power of ML to Java developers where they are, that is, on the JVM. That way, developers can take advantage of the large
library ecosystem for non-ML tasks such as loading data from databases or responding to a web request, leverage the high-performance JIT compilers available in the JVM to accelerate code without writing everything in a low-level language, and easily integrate with their applications. The languages with large ML library ecosystems tend to be harder to build large applications in due to the lack of static typing and compiler checking.

Working with teams building products in Java exposed a need for a good single-node ML library on the Java platform that could be used as a basis for building ML product features.

The Tribuo team started this effort in 2016 and, at the time, there were few good options available with enterprise-friendly licenses outside of Apache Spark, which is not particularly well suited to single-node problems. Tribuo moved into production use inside Oracle products in 2017, and the team has been building on it ever since.

Because Tribuo is written in Java, the team can leverage the type system to enforce the kind of correctness guarantees that software developers expect—and that are sadly lacking from much of the ML ecosystem. The team can also leverage Java’s strong library support for integrating with other file formats, such as CSV and JSON, and the wide variety of database and other connectors for integrating with other data sources.

While the team expects most users of Tribuo to integrate it into their applications as they would any other library, developers can also interactively develop models using popular notebook tools such as Jupyter. This workflow should be familiar to data scientists who work in the Python ecosystem, and the team hopes that Tribuo and Java won’t provide too big a learning curve. Tribuo’s tutorials are provided as Jupyter notebooks running on top of Java 11+ using a Jupyter jshell kernel (such as IJava).

Tribuo itself is compatible with Java 8+, and the code in the tutorials is easily translatable to Java 8 if necessary.

Using Tribuo

I shall run through an example of loading data, training a model, and evaluating its performance in Tribuo. I’ll also discuss the methods and use of each of the types introduced. Tribuo has extensive Javadoc coverage, which has more detail about all the classes I’ll discuss in this example.

It wouldn’t be an ML article without using the MNIST database of handwritten digit images. Using that, I’ll build a simple linear model that can predict what digit is represented by the supplied pixels. I’ll treat the grayscale pixel intensities as the input features, one per pixel.
Tribuo’s type system starts with the `Output` interface. The implementations of this class are the values that can be predicted by a Tribuo `model`, and many of Tribuo’s classes are parameterized by the type of `Output` they operate on.

Because I’m going to perform multiclass classification, I’ll use the `Label` subclass of `Output`, but there are subclasses for anomaly detection (`Event`), clustering (`ClusterID`), regression (`Regressor`), and multilabel classification (`MultiLabel`). There are tutorials for each prediction type on Tribuo’s website.

Each `Output` subclass has a family of support classes, each of which subclasses one of these interfaces: `OutputInfo` tracks the domain of the output (such as the possible values and the number of times those values occurred), `Evaluator` produces an `Evaluation` measuring the performance of the supplied `model` on the supplied data, `OutputFactory` manufactures `Outputs` of the right type from Strings, along with the sentinel `Unknown` output and the right versions of both `OutputInfo` and `Evaluator` for this `Output` subclass.

In Tribuo the data loading procedure starts with a `DataSource<T>`. This is typed based on the prediction type that’s going to be used, which in this case is `Label`. `DataSource<T>` is an `Iterable<Example<T>>` and also provides methods for computing the data provenance and an accessor for the `OutputFactory` instance. Tribuo’s `Examples` are tuples of an `Output` instance representing the ground truth for this data point (or the `Unknown` output if it’s not known) and an array of `Feature` objects.

Each `Feature` is a tuple of a feature name `String` and a double for the feature value. `DataSources` perform the initial ETL step of getting data off disk or out of the database and into examples and features (this step is also known as `featurization`). Tribuo provides plenty of `DataSources` for different data formats and also an extensive columnar input processing package, but below I’ll use the `IDXDataSource`, which can read an ML-specific format called IDX that is best known as the format that MNIST is provided in.

```javascript
var labelFactory = new LabelFactory();
var trainDataSource = new IDXDataSource<>(Pat
```

This code loads in the MNIST training set from the canonical files, creating an `Example` for each image that uses the nonzero pixels as features (because Tribuo’s representations are implicitly sparse) and the `Label` instance representing which digit it is.

`DataSources` get things into memory, but to train a model or transform the data further, it needs to be loaded into a `Dataset<T>`. Tribuo’s `Dataset` objects track the domains of
the features and outputs, canonicalize all the feature names so they all refer to the same underlying String instance (which is unnecessary on newer Java versions using the G1 garbage collector but is important for performance on Java 8), and provide methods for transforming feature values (such as rescaling features to be in the range of 0 to 1).

Tracking the domains is performed by the data set so this information could be shared across many different model training runs, each of which would otherwise have to compute it.

```java
var trainDataset = new MutableDataset<>(trainDataSource);
```

I could now apply transformations to this data set, subsample or oversample it to correct class imbalances or remove infrequent features. However, MNIST is pretty well behaved, so I'll move on to constructing a Trainer.

Tribuo's Trainers are implementations of a `Model<T> train(Dataset<T> data)` method, though there is an overload of `train` that accepts additional provenance information to be stored in the model. Trainers are mostly immutable; the only mutable state is a random number generator, which is used in a thread-safe manner and has its usages tracked (to allow the provenance system to replicate any given run).

Most of the trainers can be used concurrently by multiple threads. Trainers which are not concurrent have their train method marked with `synchronized` to ensure correct execution (this is usually due to the underlying native code not being threadsafe rather than any issue inside Tribuo itself).

I'm going to build a simple logistic regression that's trained using stochastic gradient descent (SGD), specifically an SGD algorithm called AdaGrad.

```java
var trainer = new LinearSGDTrainer(new LogMulticlass(), new AdaGrad(0.5), 5, 42);
```

This trainer builds a linear model using SGD, minimizing the log loss (so it's a logistic regression rather than a support vector machine or another kind of linear model), using an initial learning rate for AdaGrad of 0.5, training for five epochs (that is, five complete passes through the training set), and setting the random number generator (RNG) seed to 42.

This trainer can be used multiple times, either sequentially or simultaneously, and the provenance system records the state of the RNG used in each training run, ensuring it can be replicated. The RNG state is different across the runs to ensure ensemble algorithms that combine multiple independently trained models into a single model produce the expected output.

```java
var model = trainer.train(trainDataset);
```
Tribuo produces logging output during model training, allowing you to monitor the training procedure, along with displaying how many features and outputs it found in the data set. After the training algorithm has finished, it produces a `Model<Label>`, which is what I’m interested in because it allows the software to make predictions on new data.

ML wouldn’t be very useful if it made predictions only on things it already had seen. The model records the data set provenance (in this case, the hash and time stamp of each file, along with the paths on disk and the number of features and labels found), the trainer provenance (the SGD algorithm, the loss function, and hyperparameters such as the learning rate and the random seed), and some general model information such as the version of Tribuo used and a time stamp indicating when the model object was created.

Tribuo’s models expose four main methods, and the prediction centered ones have overloads for batch predictions accepting `Iterable<Example<T>>`. There are also accessors for the domains, provenance, and a few other model-specific fields, but those are fairly straightforward.

The first relevant method is `Prediction<T> predict(Example<T> example)` along with the overloads for various collections of examples. It takes an example, passes it through the model, and creates a `Prediction<T>` with the relevant generic type.

The second relevant method is `validate(Class<? extends Output> clazz)`, which checks to see if the model contains the appropriate output type. This is used when deserializing models, because the generic type is not available at runtime for deserialized objects. You can use `validate` as a guard before performing the unchecked cast to the desired model type.

Next come the explanatory methods, and first is `Map<String, List<Pair<String, Double>>> getTopFeatures(int n)`.

Some models allow the computation of the most important features used by the model, the `getTopFeatures` method exposes this, and the returned `Map` has either one key per output dimension (such as one key per possible class or using a regression dimension) or a single key ("ALL_OUTPUTS") if the model can’t express per-dimension feature scores.

Some models can’t score features at all, and those return an empty map.

Finally, you have Tribuo’s built-in model explanation system, which is called excuses because explanations are usually detailed and model-specific, whereas the excuses are an approximation to that (note Tribuo also has a separate package
for LIME-style explanations of classification models, which works with any native Tribuo model). This comes in the form of

```java
Optional<Excuse<T>> getExcuse(Example<T> example)
```

and overloads for collections of examples.

Excuses are available for linear models, trees, and ensembles; other model types return an empty `Optional`. An excuse provides the features used to make a prediction along with the contribution of those features to the outcome—either the path through the tree or the linear model weights.

### Making the run and evaluating the model

How does the model perform? First, I will load in the MNIST test set, and then I’ll show how to make predictions on single examples and collections of examples and how to use the evaluation classes to compute performance metrics such as accuracy.

```java
var testDataSource = new IDXDataSource<>(Path <Prediction<Label> prediction = model.predict(List<Prediction<Label>> batchPredictions = mo
```

A prediction contains three main elements: the predicted output, an optional set of other outputs and scores (if this model supplies scores for the other outputs), and a reference to the example that produced it.

Predictions also contain a boolean denoting whether the scores are probabilities, the number of features present in the example, and the number of features that the model used to make the prediction. The latter information is useful for evaluating how a model behaved and if it understood the supplied features. This information tends to be most helpful when you are working on NLP tasks, because it shows how many of the words in an example exist in the model’s vocabulary.

Each output type has its own `Evaluator` and `Evaluation` subclasses. An `Evaluator` is a factory that produces evaluations when supplied with models and test data or predictions. Each `Evaluator` computes a specific set of performance metrics and the appropriate averages. An `Evaluation` contains all the computed performance metrics, along with the sufficient statistics to compute other metrics, if desired.

One way to evaluate the model is to pass in the precomputed predictions along with the model, for example:

```java
var evaluator = new LabelEvaluator();
var evaluation = evaluator.evaluate(model, ba
```
You can also evaluate the model by passing the model and the test data directly, and the `Evaluator` internally calls the `predict` method to generate the necessary predictions. This is preferable because Tribuo will automatically track the test data provenance in the evaluation rather than requiring you to pass it in as an argument:

```java
var evaluation = evaluator.evaluate(model, testDataSource);
```

Both of the evaluations are equivalent, and you can now inspect an evaluation to see how accurate this model is:

```java
var accuracy = evaluation.accuracy();
```

Or you can generate a formatted string suitable for display in a notebook or on a terminal. Classification evaluations also include the confusion matrix (such as a table representing for a specific ground truth label what labels were predicted), which can be queried separately or formatted in a String for display in a notebook or on a terminal.

```java
System.out.println(evaluation.toFormattedString);
```

This String shows several metrics for each label in the test set, along with appropriate averages of those values across the labels, as follows:

- The number of times this label appears ($n$)
- The number of times this label was correctly predicted, that is, the “true positives” ($tp$)
- The number of times this label was missed, that is, the “false negatives” ($fn$)
- The number of times this label was incorrectly predicted, that is, the “false positives” ($fp$)
- The `recall`, which is $tp / (tp + fn)$ and indicates the number of times the model predicted this label correctly divided by the number of times that label appeared in the test set
Tribuo also provides evaluation aggregators for computing averages across multiple test evaluations or for using techniques such as cross-validation, a popular method for measuring model performance and providing an estimate of how uncertain a model is.

**Integrating Tribuo into the ML ecosystem**

While some might like to think that all computation happens in Java or on the JVM, at the moment, most ML models are trained on other platforms such as Python using the libraries mentioned earlier. It’s very common for data scientists to explore their problems and train models to solve them in Python, yet when it comes to putting those models into production, they need to integrate the models into a Java system.

This is the cause of many issues when ML systems are deployed because the Java versions of models may use different libraries and different input processing steps, and they may produce different outputs. The Tribuo team would like developers to be able to deploy their ML models easily inside Java systems; that’s part of the reason Tribuo was built, so Tribuo supports deploying models trained in external libraries alongside native Tribuo models.

Tribuo has a notion of external models, which currently has three implementations supporting models produced by different libraries. It supports loading XGBoost and TensorFlow models directly from the serialized formats those libraries produce (which are the same across Python, R, Java, and C++), and these models behave just like TensorFlow and XGBoost models trained inside Tribuo.

Also, Tribuo supports models in the **Open Neural Network Exchange (ONNX) format**, which is a common interchange format that many models can be saved in, including those trained using Python libraries such as scikit-learn and PyTorch.

Tribuo uses Microsoft’s **ONNX Runtime** to provide the ONNX model scoring support, using a Java API that Oracle contributed to that project. This runtime allows you to deploy Python models into a Java system without reimplementing the model itself. You can also bundle some of the input processing into the ONNX model file, ensuring that the model sees the same inputs whether it runs in Python or Java, which makes keeping development and production in sync far simpler.

Loading an external model is as simple as telling Tribuo where the model is located and giving the appropriate mapping from Tribuo’s feature and output names to the model’s feature and output names.
output ID numbers. This produces a `Model` object the same as any other Tribuo model, with provenance information recording the location and hash of the external model file that was wrapped. (Here’s a longer tutorial that covers loading external models in much more detail.)

**Tribuo and the Java platform**

Tribuo is written entirely in Java, though there are a couple of dependencies that contain native code. It currently targets Java 8, and it’s tested on long-term support versions of Java along with the latest release. Tribuo on Java 8 is already a good choice for building ML systems in Java, but there are new features previewed in Java 16 that promise to greatly speed up numerical computing on the JVM. Below, I’ll discuss a few of them and how they could improve the performance of Tribuo and other ML libraries.

The new JEP 338: Vector API (Incubator) is an incubating feature in Java 16, and it will allow fast numerics on the JVM by exposing a portable way to use SIMD (single instruction multiple data) operations in Java. SIMD instructions allow the use of special units on modern CPUs, which perform multiple floating-point or integer operations in parallel on a vector of numbers. This means you can multiply two vectors together up to eight times faster than using the standard floating-point multiply instruction on CPUs equipped with AXV-512 extensions, with similar boosts for other numeric operations that are amenable.

Idiomatic Vector API code will take advantage of whatever vector hardware is available and transparently drop back to scalar code. Java’s optimizing JIT compilers can already analyze loops and vectorize them if possible, but because Tribuo’s view of data is implicitly sparse, a layer of indirection and buffering is necessary with the use of vectorized operations and, unfortunately, the JIT compilers don’t always detect this. The autovectorization process is also limited to transformations that preserve Java’s floating-point semantics; this can prevent certain optimizations and is inherently conservative with others. With the Vector API, you can code up the inner loops of Tribuo’s Tensor operations and guarantee that they will be executed using fast SIMD operations, if the hardware supports them, on both ARM and x86_64 platforms.

The Vector API unlocks a lot of performance previously unavailable on the JVM, but sometimes you’ll still need to call native code, either to access accelerators such as FPGAs or GPUs, or because you need to do something tricky such as manually allocate registers in the middle of the matrix multiply loop to squeeze out all the CPU performance.

Historically, efficiently moving data between the JVM heap and native code has required the use of `sun.misc.Unsafe` or `ByteBuffer`. But with the Foreign Memory Access API previewed in Java 14, 15, and 16, there’s a new, simpler
abstraction for sharing memory between native code and Java code. This API allows users to allocate memory on the native heap and access it transparently from Java with appropriate safety. You can use this to allocate memory within Java, write out data to the memory, and call native functions passing a simple pointer to the data.

These functions are currently called via JNI, but because only a pointer is being passed rather than copying all the data, the overhead is much lower. Within Tribuo, the foreign memory allocation facilities allow you to create Tribuo Tensor types directly on the native heap and pass them to high-performance, native linear algebra routines such as those of the BLIS library without breaking Java’s memory safety guarantees or inducing a huge burden on Tribuo’s developers.

The Foreign Linker API, also incubating in Java 16, makes it significantly simpler to call native code without writing any JNI, making the integration of native code into Java programs much simpler.

These new features will greatly accelerate the performance of numerical code on the JVM, and the Tribuo team is excited to integrate them into Tribuo. The team expects them to be completely transparent to users; it should be as simple as upgrading the JVM and switching to the appropriately enabled version of Tribuo.

**Conclusion**

I’ve given an overview of the Tribuo library, talked about the design choices inherent in building an ML library that looks and feels like a Java library, and discussed integrating Tribuo into a wider ML workflow. I also briefly discussed how new Java features that are incubating or in preview could be used to accelerate the performance of numerical code such as Tribuo and other ML libraries running on the JVM.

For more information about Tribuo, along with tutorials and documentation, visit [tribuo.org](http://tribuo.org) or check out the code on [GitHub](https://github.com). Oracle develops Tribuo in the open and welcomes bug reports, discussions, and code contributions from the community.

**Dig deeper**

- Announcing Tribuo, a Java Machine Learning library
- Getting started with notebooks
- Oracle Machine Learning
- Machine learning—defined
- Oracle AI & data science blog

---

Adam Pocock
Adam is a Machine Learning researcher, who finished his PhD in Information Theory and feature selection in 2012. His thesis won the British Computer Society Distinguished Dissertation award in 2013. He's interested in distributed machine learning, Bayesian inference, and structure learning. And writing code that requires as many GPUs as possible, because he enjoys building shiny computers. He's the lead developer of Tribuo, a Java Machine Learning library.