Sandwood: Runtime Adaptable Probabilistic Programming for Java

Daniel Goodman
Oracle Labs
Daniel.Goodman@oracle.com

Adam Pocock
Oracle Labs

Natalia Kosilova
Oracle Labs

Abstract

This paper outlines Sandwood, a Java like probabilistic programming language for compiling models to Java objects.

Keywords: Probabilistic Programming, Java, Sandwood, MCMC, Parallel Programming

1 Sandwood

Inspired by our groups work on Augur [7], Sandwood[1], formally Vate [2], is a compiler, runtime, and Java like probabilistic programming language [3] for creating models for JVM-based applications. An example model and the Java code to use it is in Appendix A. Sandwood is open source and available under UPL. Inference is performed via MCMC style sampling and models are constructed to allow dynamic switching of execution environment. Current execution environments consists of single threaded, and Fork-Join [6] JVM environments, and the internal environment Callisto [4].

Models are compiled to JVM classes that encapsulate the complexity of the different model operations, providing the user with a clean Object-Oriented interface. Sandwood is single assignment, but designed to be familiar to Java programmers, making models more accessible to the Java developer community. Models can perform three classes of operation: *Value inference*, where the value of any variables that are not fixed are inferred; *Probability inference*, where the probabilities of individual parameters and the whole model are estimated; *Conventional execution*, which can generate outputs from a trained model in cases such as linear regression or can produce synthetic data sets. The values of variables within the model can be fixed at compile time or at run time.

Value inference is performed primarily through Gibbs sampling [1] using conjugate priors when possible and reverting to marginalizing out sampled variables for finite discrete distributions, and localized Metropolis–Hastings [5] inference for unbounded or continuous distributions. The results of this inference can be: All the sampled values less any requested burning in and thinning; The values when the model was in its most probable state (Maximum a Posteriori, MAP); Or no values saved. These policies are set for the whole model and can be modified on a per variable basis.

1.1 Contributions

The contributions of Sandwood are:

- A type safe language for writing probabilistic programming models that is familiar to Java developers.
- A compiler and runtime system for compiling models to Java class files, allowing efficient execution.
- An Object-Oriented design for compiled models that enforces the separation of the model implementation.
- An Object-Oriented design allowing the execution environment to be dynamically changed so models written once can take best advantage of the system they are used on.

2 Model Presentation

Models are declared via the keyword `model` followed by the model name, any arguments, and a block containing the body. The arguments and the body are valid java syntax, with the exception of syntactic sugar for initializing arrays of constants and setting loop bounds. An example is in Appendix A. In Java the body describes a sequence of operations to perform, in a probabilistic model it describes the relationship between variables so the value on the left of an assignment can affect the values on the right. Variables named in the model will become fields in the compiled model allowing properties of these variables to be queried or set.

Variables within the model consist of base types such as `int` and `double`, arrays, and random variables representing instances of the distributions supported by Sandwood. Random variables are constructed either via a call to a constructor, for example `new Beta(1.0, 1.0)`, or via statically imported factory methods. The method `observe` is used to tie the values generated by the model to data provided by the user for inference. One or more values can be drawn from a random variable by the `sample` method. If values are not observed they can be changed during the inference steps.

Sandwoods single assignment semantics necessitates additional support for operations such as summing the values in an array that would have used mutable state. To achieve this the language includes the construct `reduce` which takes as its arguments an array of values of type X, a unit value of type X, and an associative function taking two values of type X and returning one value of type X. These are used to build a binary tree with the function called at each node and either an array element or a unit value at each leaf. `reduce` returns the result of executing this tree.

Models can be split into reusable methods, which are not required to exist in the same files, but can be formed into

[1] sandwood.org

libraries, allowing families of models with a similar structure to be created without having to recreate the entire model. Recursive models are currently not supported because functions are inlined during the compilation process. This decision was taken to allow future GPU support. An example method using a reduce operation is in Appendix B.

2.1 Compiled Model

Compiled models are a collection of classes that work in conjunction with the Sandwood runtime. The runtime is used to separate out common code, preventing duplication between models, and improving maintainability. The compiled class that the user will interact with has the same name as the model. Named parameters in the model have a public field of the same name in this class. Each of these fields references an object that can be used to assign values to and query properties of the parameter.

Once the value of parameters in the model have been set by a user or inferred from training data, it is possible to fix them on a per value basis. Use cases for this include: Fitting a topic model, then fixing the topics and running inference against new documents to determine their topic; And running conventional execution on a trained model to generate a synthetic dataset. Inferred data can be saved as JSON files and used to instantiate new copies of the model.

3 Compilation

Sandwood model classes are split into two categories: Classes that provide the object-oriented aspects of the model; And core classes implementing a common interface, providing the current state of the model, and the numeric methods to manipulate this state. Only one of these core classes is instantiated at any given time, with each targeting a different hardware/software backend. This separation is particularly important in allowing models to target different hardware or runtime systems by constructing a different core class while maintaining the stability of the user code.

3.1 Compilation Process

The compilation of a model takes the following steps: 1. Translate the model to Java API calls. 2. Compile to intermediate Java class files. 3. Execute the intermediate code to construct a DAG representing the model code. This is a direct representation of the model code, not a Bayesian Network. 4. Explore the DAG to determine relationships between variables. 5. Construct Intermediate Representation (IR) trees of methods for inferring new values and calculating probabilities for each random variable. 6. Construct IR trees representing methods for combining these operations. 7. Apply optimisations to the trees. 8. Output the constructed methods to the target language.

The first step is source-to-source translation to Java code which contains static methods consisting of API calls. A root method represents the model, and each method defined in the model has a corresponding method defined. The Java code is then compiled producing classes to be executed in the next step. The Java compilation allows Java to type check the model. Any type errors are mapped back to where they occur in the model. The API also provides the potential to construct translators for other languages.

Executing the root method API calls constructs a DAG consisting of task nodes and variable nodes representing the model. Task nodes represent operations within the model such as arithmetic, sampling, constructing random variables, array operations, and control constructs such as *for* loops and conditionals. Variable nodes represent the data generated by each task, with each task producing a single variable. The DAG is then traversed to identify producer-consumer relationships between random variables, input values, observed values, and internal values. To construct the complete DAG for a given model would require runtime information describing how many iterations each *for* loop performs, this could also make the DAG prohibitively large. To overcome this, like in a Bayes net diagram, "for loop" nodes in the DAG represent multiple iterations, with each iteration having a different index value. In conjunction with the granularity of arrays dependencies can exist between loops iterations.

IR trees representing code for conventional execution of parts of the model are generated from the DAG via calls to the nodes in the DAG that will then recursively construct the required trees. Code for value inference via Gibbs Sampling and probability calculations requires the ability to invert the relationships between consuming and producing tasks. This is done by a combination of traces recording sequences of tasks, generated templates ensuring IR trees only executes if a relationship exists, and inverse functions included in tasks.

All generated methods are optimized, this allows the earlier steps to produce simple code that is easier to check for correctness. Examples of the optimizations include partial evaluation with know value and bounds tracking on variables, code reordering and loop and variable elimination. Many of the optimizations cannot be applied later by the target language compiler because domain specific information is lost.

4 Conclusions

We introduced Sandwood, a probabilistic programming language for describing and compiling JVM based models that can be dynamically targeted to different runtime environments. Sandwood's models share a common object oriented interface which exposes many common operations making it easy to abstract over modelling decisions when integrating them into larger applications. Sandwood differs from existing work in the field by using a language familiar to Java developers to construct encapsulated models, separating the concerns of model use from the model description.

References

[1] Stuart Geman and Donald Geman. 1984. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Trans. Pattern Anal. Mach. Intell.* 6, 6 (Nov. 1984), 721–741. https://doi.org/10.1109/TPAMI.1984.4767596

[2] Daniel Goodman, Adam Craig Pocock, Jason Peck, and Guy L. Steele Jr. 2021. Vate: Runtime Adaptable Probabilistic Programming for Java. In *EuroMLSys@EuroSys 2021, Proceedings of the 1st Workshop on Machine Learning and Systemsg Virtual Event, Edinburgh, Scotland, UK, 26 April, 2021*, Eiko Yoneki and Paul Patras (Eds.). ACM, 62–69. https://doi.org/10.1145/3437984.3458835

[3] Noah D. Goodman. 2013. The Principles and Practice of Probabilistic Programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 399–402. https://doi.org/10.1145/2429069.2429117

[4] Tim Harris and Stefan Kaestle. 2015. Callisto-RTS: Fine-Grain Parallel Loops. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 45–56. https://www.usenix.org/conference/atc15/technical-session/presentation/harris

[5] W. K. Hastings. 1970. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika* 57, 1 (1970), 97–109. http://www.jstor.org/stable/2334940

[6] Doug Lea. 2000. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande* (San Francisco, California, USA) *(JAVA '00)*. Association for Computing Machinery, New York, NY, USA, 36–43. https://doi.org/10.1145/337449.337465

[7] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C Pocock, Stephen Green, and Guy L Steele. 2014. Augur: Data-Parallel Probabilistic Modeling. In *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger (Eds.), Vol. 27. Curran Associates, Inc., 2600–2608. https://proceedings.neurips.cc/paper/2014/file/cf9a242b70f45317ffd281241fa66502-Paper.pdf

A Linear Regression

This section contains a Sandwood model for linear regression and the corresponding Java code to train the model and then use the trained result to generate values of y for an array of new x's.

A.1 Model code

```
package org.sandwood.lafi;

public model LinearRegression(double[] x,
                              double[] yMeasured) {
  int numSamples = x.length;
  double b0 = gaussian(0.0, 2.0).sample();
  double b1 = gaussian(1.0, 5.0).sample();
  double variance = inverseGamma(1.0, 1.0)
                                      .sample();
  double[] y = new double[numSamples];
  for(int i = 0; i < numSamples; i++) {
    y[i] = gaussian(b0 + b1 * x[i], variance)
                                      .sample();
  }
  y.observe(yMeasured);
}
```

A.2 Java code

```
double[] xs = ...
double[] ys = ...

// Construct an instance of the model
LinearRegression l = new LinearRegression(xs, ys);

// Configure the model
l.setDefaultRetentionPolicy(RetentionPolicy.MAP);
l.setExecutionTarget(ExecutionTarget.forkJoin);

// Train the model with 1000 steps of Gibbs
// sampling
l.inferValues(1000);

// Inspect the inferred values
double b0 = l.b0.getMAP();
double b1 = l.b1.getMAP();
double v = l.variance.getMAP();

// Fix the model parameters to the most probable
// values
l.b0.setValue(b0);
l.b1.setValue(b1);
l.variance.setValue(v);

// Change the values of x
double[] alt_xs = ...
l.x.set(alt_xs);

// Set all values of y to be saved
l.y.setRetentionPolicy(RetentionPolicy.SAMPLE);

// Generate 100 possible y values for each x value
l.execute(100);

// Recover the y samples
double[][] alt_ys = l.y.getSamples();

l.shutdown();
```

A.3 Training Java Code

B Sandwood Function

An example of a Sandwood function. This function is using the reduce construct to sum the elements of an array.

```
private double sum(double[] a) {
  return reduce (a, 0, (i, j) -> { return i + j; });
}
```