Report : Exploring Figaro

Arijit Pramanik

December 9, 2018

Abstract

Figaro is a probabilistic programming language that is built as an embedded library on Scala, which is a *Turing-complete* language and makes use of both the functional and the object-oriented programming paradigm.

1 Introduction to Figaro

Figaro helps in modeling **probabilistic relational models**. Considering the model of a *simple coin toss*, we have a *random variable* that models the outcome, by assigning a value 1 when heads, and 0 when tails. This outcome has an *associated probability distribution*, namely, Bernoulli distribution with parameter p describing the fairness of the coin.

```
val coin = Flip(0.4) // coin is biased towards Tails
```

The constructs of Figaro are heavily used to model **Bayesian** and **Markov networks** which are used heavily in the field of Artificial Intelligence today for inference modeling.

1.1 Brief Recap on important Scala concepts

- *Traits* : Similar to abstract classes in Java, these encapsulate field and method definitions, which can be reused by other classes. However, unlike class inheritance, a class can inherit any number of traits.
- *Case classes* : Scala provides support for case classes, which do not need any instantiation with the new operator and the object construction is carried out by the default implicit `apply` method
- *Lazy evaluation* : Scala defers time-consuming computation until it is strictly required by another method in the program, which helps speed up computation.
- *Type inference* : Scala is a strongly-typed static language and hence there are no run-time ambiguities regarding type, and the type of all variables/values can't be substituted and are determined at compile-time.
- *Immutability* : All variables with keyword `val` are immutable by default, while those with keyword `var` are mutable by default. This is similar to the immutability of records in Ocaml
- *Object-oriented paradigm* : This allows Scala to treat functions as objects and hence can be passed around the program. A singleton static object can be used to instantiate the whole program
- *Others* : Scala has features like higher-order functions and curried functions, with improved concurrency control. It also supports linear pattern-matching like Ocaml

1.2 Basic Figaro constructs

Monad : A Functional programming concept that defines a data type, and how the values of that data type are combined; functions that use those data type, and compose them together into actions, following the rules defined on those data types.

Figaro extensively uses the **probability monad**, which builds the computation from values to the probabilistic models built on those values. A Monad uses a type constructor with type M to convert a value of type a to a Monadic value of type $(M\ a)$ and puts it into a container. We can analogously say that :

- Monadic unit := Constant. e.g. Constant(1.0) generates a probabilistic model that always returns true with a probability 1
- Monadic bind := Chain. `Chain[T, U]` helps model the conditional distribution of the child U, `T => Element[U]` given the distribution of the parent, T, `Element[T]` e.g. `Chain(Flip(0.7), (b: Boolean) => if (b) Constant(1); else Select(0.4 -> 2, 0.6 -> 3))`. This will choose the first clause with a probability 0.7 and the second one with a probability of 0.3
- Monadic fmap := Apply, leverages Scala functions operating on values to Figaro elements. e.g. `Apply(Select(0.2 -> 1, 0.8 -> 2), (i: Int) => i + 5)` yields 6 with a probability of 0.2 and 7 with a probability of 0.8
- Monadic containers := Process. The general trait of Figaro collection is a Process, which represents a possibly infinite collection of random variables. It is a generalized mapping from an index set to an element. `map` and `chain` use the default constructor to generate a new `Process` or collection of `Elements`. e.g. `p.chain(Normal(_,1))` will produce a new collection in which every element is normally distributed with mean equal to the value of the corresponding element in the original process.
  `p.map(_> 0)` will produce a `Process[Int,Boolean]` mapping every element in the existing container to False

Apart from the monadic constructs defined above, Figaro provides some syntactic sugar which makes it easier to define the probabilistic relational models. These are :

- Conditions : This helps us to assign certain outcomes which violate the specified condition attached to the random variable, generated by the probability model with a probability 0. e.g. `val x1 = Select(0.1 -> 1, 0.2 -> 2, 0.3 -> 3, 0.4 -> 4)`
  `x1.setCondition((i: Int) => i == 1 || i == 4)`. This assigns a probability value of 0 to the other outcomes 2 and 3, while the probability of outcomes 1, 4 remain 0.1 and 0.4 respectively
- Constraint : These are functions which map the `Values` to `Doubles`. The probability of particular outcomes are multiplied by the specified weights and then re-normalized which helps to model the bias of a distribution towards specific outcomes for random variables. e.g. `val x1 = Select(0.1 -> 0, 0.2 -> 1, 0.3 -> 2, 0.4 -> 3) x1.setConstraint((b: Boolean) => if (b) 1.0; else 0.1)`. This enforces that the outcome 0 is ten times more likely compared to the other outcomes.
- Universe : A universe is a collection of elements, on which the given reasoning algorithm operates. In case of a dynamic model, a static universe can model the intrinsic property of the elements itself, while a dynamic universe (creating a new universe at each time step) updates the current properties of the model adhering to the invariants specified in the static universe. Subsequently, elements in a universe can be activated or deactivated depending on whether we want to run an inference algorithm on them. e.g. a valve‘s condition can be specified using its intrinsic property of failure of the material in a static universe, and its condition can be updated at each time step by creating a new universe

Abstract classes, class inheritance and addressing objects by reference are used heavily by Figaro from Scala, to model the relationships between different classes representing related entities

2 Analysis of the language constructs

An `Element` is the core component of a probabilistic model. Elements can be understood as defining a *probabilistic process*. Elements are parameterized by the type of *Value* the process produces, e.g. `Element[Int] or Element[Double]`. Comprising of a *random* as well as a *deterministic* component, the `generateRandomness` method samples random values from the given probability distribution. The `generateValue` generates an output value given the randomness. Atomic elements extend the `Atomic[T]` trait, since they are independent of other variables. Similarly, for continuous probability distributions, these generally extend the `Continuous[T]` trait.

Similarly, `ElementCollection[T]` is a trait, which refers to a collection of elements and can be used to reference them, and is by default extended by the `Universe`. Similarly, Evidence is asserted using a sealed abstract class, along with other basic constructs like `Flip, Chain, Constraint, Parameter(for learning) and Pragma(for abstraction)`

2.1 Reasoning Algorithms

Figaro provides for computing the conditional probability of query elements given evidence (conditions and constraints) on elements. An important part of this is evidence, which describes all the observations that have been made about the model. Evidence can refer to constructs like `setCondition`, `setConstraint` or `observe`. This can also make use of inbuilt references in Scala to associate evidence with an object. The reasoning algorithms provided are :

- *Variable Elimination* : This is an exact inference algorithm and hence requires the universe to remain finite upon expansion. This converts the elements into a set of factors, and applies variable elimination to each of the factors
- *Belief Propagation* : This is an approximate inference algorithm for continuous random variables, but exact for discrete random variables. This also expands the universe, and converts variables into factor nodes, and operates on message passing between the factor and variable nodes.
- *Lazy Factor Inference* : Sometimes, the model may not be finite and the inference may involve an infinite no. of variables, in which case Figaro performs a Lazy factor inference that creates a factor graph of finite depth, and performs inference on this graph, while accounting for the effect of the unexplored portion of the factor graph.
- *Importance Sampling* : This can be applied to a universe whose expansion is not finite, but the number of elements generated is finite.This works on the simple approach of sampling and accepting the value if it satisfies the condition, and otherwise rejecting it, and taking into account the weight of the sample in case of a constraint.
- *Markov Chain Monte Carlo methods* : Consists of the Metropolis-Hastings and the Gibbs sampling methods. These basically propose a new *proposalScheme*, i.e. a new proposal distribution for *randomness* at the beginning of each iteration, and it accepts or rejects the samples according to that. This involves the `nextRandomness` method of `Element[T]`

The query interface provided by Figaro comprises of probability of the outcome of the inferred variable, probability that the outcome satisfies a particular predicate, and calculating the expected outcome given some function on the outcome. e.g. `ve.distribution(e2)`, `ve.probability(e2, predicate)`, `ve.expectation(e2, (b: Boolean) => if (b) 3.0; else 1.5)`
Since, it is difficult to work with continuous values, Figaro provides for various schemes of selecting out a set of abstract points given a set of concrete points belonging to the probability distribution. These can be specified by `addPragma` method

Figaro also provides for `LearningParameters`, which can be learnt by using learning algorithms and only EM algorithm is provided as default for this purpose. This initalizes random estimates for parameters, tries to set weights for them in the expectation stage using the likelihood of data given parameter and tries to maximise the likelihood of the parameter by tweaking these weights in the maximization step. This converges to the *maximum a-posteriori* value of the parameter, which represents the maxima of the posterior distribution. However, Figaro only provides for the Beta and Dirichlet distributions as conjugate priors so all kinds of random variables entailing different distributions can't be learnt so easily using data.

3 Extending the language

Figaro provides immense flexibility in the sense that, random variables and associated probability distributions can be modeled by inheriting the base class `Element[T]` and more complex distributions can be modeled by extending pre-existing classes with different traits like `Atomic[T]` and `Cacheable[T]`. e.g. `AtomicUniform` distribution can be defined with the trait `Atomic[T]` since it is an independent atomic element and does not depend on any other elements.

Similarly, Figaro provides a number of inference algorithms. All the inference algorithms inherit from the `Algorithm` class, which define the basic framework for starting, stopping and killing the algorithm, and the default methods for `initialize` and `cleanup` can be over-ridden for book-keeping purposes. The algorithm can be defined for a fixed number of iterations or can run as long as required.

3.1 Defining a new Atomic Class

I have implemented a new Atomic class which inherits from `Element[T]` as do all other classes for sampling from the Uniform or the Gaussian distribution. There is a default uniform distribution, which inherits from `AtomicUniform` class and maps every element to a probability value of 1.0 implying that every element is equally probable. My class models a sampling algorithm of generating a number of random values uniformly and returning the maximum of them.
Scala has an inbuilt random number generator which by default, generates numbers between 0 and an upper bound, sampling them from a uniform distribution. Extending from the *abstract* class `Element[T]`, `generateRandomness` samples a number of values between 0 and upper bound specified, into a `List[Int]`. Here the randomness is of type `Int`. Finally, `generate` method returns the maximum from the list of sampled integers.

Different test suites are available for testing the randomness of the generated integers, including the chi-squared test, which can be easily tested on statistical software, such as R.

3.2 Creating a new Compound class

Similarly, new classes can be created which are *compound*, in the sense that they are comprised of dependencies among atomic elements, We can inherit from the `Chain` class, which takes in a sequence of *Element*s and values are randomly sampled for each of those options. Then the *AtomicUniform* distribution can be invoked over all the values sampled, and then the resulting value can be chosen uniformly. Here, we have the parent elements as Atomic elements which have their own distributions from which random values are sampled, and then, the child uniform distribution is conditioned on the values of the parents.

3.3 Analysing and building a new algorithm

Every algorithm requires a list of target query elements and the universe. It consists of expansion to include in all possible elements, for building the factor graph and the sampling method, where algorithms can choose to extend default samplers. In the case of extending an abstract class, we provide a companion class using factory constructors. However, the traits for the anytime version and the fixed iteration version are different. So, a customised algorithm must extend the Algorithm trait, and then the fixed iteration or anytime trait, and over-ride the *sample function* if required.

4 Evaluating the language

I have tried out a couple of examples, some of which have been written from scratch, while others have been extended from the given examples. Some of the demonstrated examples are :

- An example involving car engines which uses an **abstract class** to model an engine. The three different engine versions *inherit* from the superclass Engine, which is an *abstract class*, so that can't be instantiated. Here, we also create objects of these engine classes and define a child variable engine, depending on the power of the engine using the **CPD construct**. A **Markov network** is also defined which allows us to model the manufacturer bias among the engines, and hence, we can now set *constraints* between different engine versions. We also record our observations in the form of *conditions* on the engine random variables. Then we run an inference on this above defined model to infer some probability queries regarding the different engine versions using *20,000 samples* of the **Metropolis Hastings algorithm**. This is not an *exact* inference algorithm, and hence doesn't provide an exact same answer every time, but provides a value within 0.01% of the actual theoretical value. Using **Variable Elimination** on the speed variable, conditioned on the engine, we query the expected value of speed of the car, and obtain an exact answer for the same since the *factor graph* for this Markov Network is finite and this is an *exact* inference algorithm. Here, an extension was implemented for the Metropolis Hastings algorithm since we can define the *proposal scheme* for this algorithm, or use `proposalScheme.default`

- A simple example involving a **Bayesian network** was tried out using different random variables involving calculating of the probability of wet grass, which depends on rain and sprinkler operation, which in turn depend upon the rainy season. Here, along with simple query inference, the *learning algorithm* leveraging **Expectation Maximization** was tried out, where data was provided in the form of a Scala sequence. Here, we learn the probability of observing wet grass given the observations made on the wet grass. The learning algorithm involves **Belief Propagation** which is required to estimate the *sufficient statistics* corresponding to the distribution, which is used by the learning algorithm in the sampling step. e.g. *mean* are *variance* are sufficient statistics for the normal distribution

- A problem modeling the evolution of three valves in time was explored which uses *case classes* and subsequently *case objects* in order to model the valve states. This uses a *dynamic reasoning algorithm*, which is `FactoredFrontier`, and the only default one provided by Figaro. Here, the case objects do not need instantiation and can be accessed anywhere inside the singleton class. We explore the use of creating new universes, and using *references* from other universes, in this inference algorithm, which works only on factors instead of sampling. Here, a *static universe* is modeled upon the intrinsic property of failure of the valves, and then we have that, a universe is created dynamically at each time step to model the state of the valve with time. This algorithm also internally uses *Belief Propagation*. We then run our queries as per the default interface provided by Figaro to infer about the valve states for the valves in our model. This however works at each iteration and provides an exact accurate answer for each time step

- A *decision making problem* has been modeled using Figaro, which involves a market with different values for the current market status. Then, a survey random variable has been created, which is conditioned upon market and models the perception of different people regarding the current market status. A *utility function* is specified which evaluates the decision taken, given the current market status. Figaro provides a *decision variable* which consists of a set of actions and a random variable of which it is a child. Figaro provides a `DecisionVariableElimination`, which provides the *most optimal decision*, given the utility, and sets that using the `setPolicy` method. We can also extend this to a multiple decision problem, via which we can also explore further objects and class relationships provided by Scala.

  Other examples have been also been written to show the effectiveness of `LazyVariableElimination` and *lazy evaluation* provided by Scala while modeling relations between different classes which have fields dependent on one another.

5 Shortcomings of Figaro

I noticed that the Bayesian networks that are generally used for modeling is kept simple since inference on a huge model can be computationally expensive. Figaro utilises this fact and CPD for a child can't be defined with more than 5 parents. This is the default for the `CPD` construct in Figaro. `RichCPD` which allows a much more flexible decision also allows only 5 parents though it does not inherit from `CPD`. Similarly, the `Apply` function defined in Figaro, can not be applied on more than 5 instances of `Element[T]`. Since, for every `Element` defined, this function is implicitly applied to the list of arguments passed, we have that the `CPD` construct, which uses this, can't be applied to more than 5 parent elements.

The Scala scope and Figaro scope of objects are not the same. It tries to make heavy use of implicit arguments and conversions. Since, traits are used for instantiation, so arguments required to be declared at instantiation time require more user effort.
Additionally, since the universe has a lot of random variables, it needs a lot of data structures to keep track of them for inference algorithms running on the universe, so it is not much memory-friendly

6 Conclusion

Figaro is open source, Scala library where one can create probabilistic models with little AI and ML experience, using the power of recursion in Scala and `Chain` in Figaro. Figaro is a language and platform with which one can explore new types, paradigms and ways of building probabilistic models. Object-oriented programming is used to build these models, so maybe we can use this in Java, but the functional aspects of Figaro makes using it much easier.