FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Exploring Visual Programming Concepts for Probabilistic Programming Languages

Gabriel Cardoso Candal

WORKING VERSION

U. PORTO

FEUP **FACULDADE DE ENGENHARIA**
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira

February 11, 2016

Exploring Visual Programming Concepts for Probabilistic Programming Languages

Gabriel Cardoso Candal

Mestrado Integrado em Engenharia Informática e Computação

February 11, 2016

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

ADT	Abstract Data Type
DARPA	Defense Advanced Research Projects Agency
DOM	Document Object Model
ML	Machine Learning
IDE	Integrated Development Environment
PP	Probabilistic Programming
PPAML	Probabilistic Programming for Advancing Machine Learning
PPL	Probabilistic Programming Language
PR	Probabilistic Reasoning
PVL	Purely Visual Language
VDP	Visual Dataflow Programming
VMP	Variational Message Passing
VP	Visual Programming
VPE	Visual Programming Environment
VPL	Visual Programming Language

Chapter 1

Introduction

This first chapter aims to provide the reader with an overview of this dissertation. It starts by introducing the context this work is inserted in, identifying the problem which we aim to solve, how we plan on solving it, and the expected outcome. Lastly, it gives a bird's eye view of this report's structure.

1.1   Context

There is, among several domains with interesting and relevant problems to solve (computer vision [KT15], cryptography, biology, fraud detection, recommender systems [Alp10], . . .), the recurring necessity to be able to make decisions in the face of uncertainty using machine learning (ML) methods.

Successful ML applications include Google's personalized advertising and context-driven information retrieval, Facebook's studies of how information spreads across a network or UC Berkeley's AMPLab contributions towards Amazon Web Service and SAP's products [BAF+15].

Typically, there are two alternatives for solving this class of problems: either use an existing machine learning model (such as KNN, neural networks or similar) [Sch08] and try to fit your data into the model, or build a probabilistic model for your own particular problem so you can better leverage domain knowledge [Gri13].

In the second alternativa one common way to approach it is by using bayesian reasoning, where you model unknown causes with random variables, feed the model the data you have gathered and then perform inference to query for the desired, and previously unknown, variables [Dow12]. The problem in choosing this method is this last step, since it is non-trivial to write an inference method [DL].

The solution to this has been building generic inference engines for graphical models, so that modeling and inference can be treated as separate concerns and people can focus on the modeling [JW96]. However, not all models can be represented as graphical models, and that's why we now

Figure 1.1: Top 10 Analytics Tools [Pia15]

have Probabilistic Programming Languages (PPLs). Probabilistic Programs let you write your model as a program and have off-the-shelf inference [Pre03].

1.2 Problem

In spite of these examples of applications in the industry, ML has been identified by Gartner, in its Hype Cycle annual review, to be in the "Peak of Inflated Expectations" stage, still far from the "Plateau of Producitvity" [Sta15].

It has also been said that ML's applications are rarely seen outside the academia, with Wagstaff claiming that there is a "frequent lack of connection between machine learning research and the larger world of scientific inquiry and humanity" [Wag12].

Arguably the scenario is even worse for PPLs, having even less adoption among tech companies than other ML methods, such as neural networks or other supervised learning models. One factor which may be contributing to this lack of usage, despite PP's power and flexibility, is the difficulty for data scientists to adapt to the textual interface these languages provide, which lack the graphical intuiton provided by other tools they are accustomed to. In a poll made to find out the most popular data-mining tools among data scientists (see Figure 1.1), half of the top 10 tools are graphically-interactable.

1.3   Motivation and Goals

The Defense Advanced Research Projects Agency (DARPA), one of the funders behind PPLs' research, has recognized some of the problems identified by Wagstaff and started a program called Probabilistic Programming for Advancing Machine Learning (PPAML) to address the shortcomings of current ML methods [Jag13]. It identifies five strategic goals:

- Shorten machine learning model code to make models faster to write and easier to understand

- Reduce development time and cost to encourage experimentation

- Facilitate the construction of more sophisticated models that incorporate rich domain knowledge and separate queries from underlying code

- Reduce the level of expertise necessary to build machine learning applications

- Support the construction of integrated models across a wide variety of domains and tool types

The purpose of this work is to try addressing the first four. In order to do so we aim to overcome the difficulties in learning a new language, either for unexperienced developers or seasoned ones, such as learning yet another syntax or getting accostumed to the language's idioms. It is known that typical languages are difficult to learn and use [LO87a] and that there are advantages in providing a language with a visual interface [AA92]. Also, studies have shown that programmers and data scientists alike resort to mental imagery when solving problems [Das02][PB99], so by providing such an interface we can approximate how people think and how they use the language to solve the problem at hand.

So, the goal of this dissertation will be to develop a Visual Programming Language (VPL) with probabilitics programming capabilities. The targeted audience are programmers and data scientists with background knowledge in statistics who aren't still comfortable with full blown PPLs, but wish to educate themselves in the topic so they can eventually leverage the power of this novel machine learning approach.

The way to do so would be developing a graphical node-based editor, similar to RapidMiner or Blender Composite Nodes, but that runs in the browser. The given editor would have the capability to compile its graph to the textual representation in the target PPL so that the user can run what he has designed, either as a standalone script or even by integrating it with his existing projects.

The hypothesis under consideration is this graphical representation is more intuitive and easy to learn that a full-blown PPL. We intend to validate such hypothesis by ensuring that classical problems solved in the literature by PPLs are also supported by our graphical representation, and then measure how quickly a group of people trained in statistics would produce a viable model in both alternatives.

1.4 Outline

Para além da introdução, esta dissertação contém mais x capítulos. No capítulo 2, é descrito o estado da arte e são apresentados trabalhos relacionados. No capítulo 3, ipsum dolor sit amet, consectetuer adipiscing elit. No capítulo 4 praesent sit amet sem. No capítulo 5 posuere, ante non tristique consectetuer, dui elit scelerisque augue, eu vehicula nibh nisi ac est.

Chapter 2

Background & State of the Art

This chapter has two purposes: describing the foundations on which this work is built on, namely Machine Learning (ML), Probabilistic Reasoning (PR), Prograbilistic Programming (PP) and Visual Programming(VP) while enumerating different tools which are based on one of these concepts.

2.1 Machine learning

Machine learning (ML) is a field which can be seen as a subfield of artifical intelligence that incorporates mathematics and statistics and is concerned with conceiving algorithms that learn autonomously, that is, without human interventation [Bri16][Sch08]. It has the potential to impact a wide spectrum of different areas such as biology, medicine, finance, astronomy [Ama13], computer vision, sales forecast, robotics [Alp10], product recommendations, fraud detection or internet ads bidding [Gri13].

Learning from data is commercially and scientifically important. ML consists of methods that automatically extract interesting knowledge in databases of sometimes chaotic and redundant information. ML is a data-based knowledge-discovering process that has the potential not only to analyze events in retrospect but also to predict future events or important alterations [Geo16].

2.2 Probabilistic Reasoning

Probabilistic reasoning (PR) is the formation of probability judgments and of subjective beliefs about the likelihoods of outcomes and the frequencies of events [?], it is a way to combine our knowledge of a situation with the laws of probability. There are subjective belifs because, in non-trivial decision-making there are unobserved factors that are critical to the decision in conjunction with several sources of uncertainty [GSD], such as:

- Uncertain inputs, due to missing or noisy data

- Uncertain knowledge, where multiple causes lead to multiple effects, or there is an incomplete knowledge of conditions, effects and causality of the domain or simply because the effects are inherently stochastic.

So, probabilistic reasoning only gives probabilistic results.

It is one way to overcome cognitive bias and be able to make rational decisions [SG01]. A trial has been made [CSB82] where physicians were asked to estimate the probability that a woman with a positive mammogram actually has breast cancer, given a base rate of 1% for breast cancer, a hit rate of about 80%, and a false-alarm rate of about 10%. It reported that 95 of 100 physicians estimated the probability that she actually has breast cancer to be between 70% and 80%, whereas Bayes's rule gives a value of about 7.5%. Such systematic deviations from Bayesian reasoning have been called "cognitive illusions,". We will describe both Bayes's rules and Bayesian reasoning in the next section.

2.2.1 Bayesian Reasoning

One way to approach PR is by using bayesian reasoning, which is inspired in the Bayes Theorem (or Rule, or Law). An equivalent formula to the theorem, in its simplest form (applied to a single event) is:

$$P(A \mid B) = \frac{P(A \wedge B)}{P(B)}$$

Where P(A|B) defines the probability of event A given that B occurred. The theorem defines how hidden causes (A) relate to observed events (B), given a causality model (P(A, B) or P(B|A)*P(A)) and our knowledge of the probability of the occurrence of events (P(B)). The inverse is also true, as we will see further ahead in this section. As an example, P(penalty | goal) defines the probability that a penalty kick was scored, knowing that there was a goal.

There are at least two interpretations to the theorem and regarding how one may think about its results [Fie06]:

- Frequentist interpretation: probabilities are defined by the relative frequency of events, given a natural sampling. Meaning, the probabiltiy of obtaining 'Heads' when rolling a dice is equal to the number of 'Heads' obtained after rolling the dice a sufficient number of times relative to the total number of times the dice has been rolled.

- Epistemological interpretation: probabilities represent a measure of belief. It can either be a result logical combination of probabilities through the usage of axioms (it's closely related to Aristotlean logic) or it can also reflect a personal belief (which is called a subjective view).

Table 2.1: Alarm system confusion matrix

	alarm	$\neg alarm$
burglary	0.09	0.01
$\neg burglary$	0.1	0.8

2.2.1.1 An example

One example of the application of this theorem is [GSD]: you know your home's alarm is ringing, but you don't know whether that was caused by a burglar or something else (maybe a bird triggered it, or there was a malfunction in the alarm system). How confident are you that you're being robbed? Consider that the alarm company, based on quality trials, defined in the confusion matrix for P(alarm, burglary) (Table 2.1).

You can interpret each table's cell as P(A, B). For instances, the top left cell is the probability that the alarm rings and there is a burglar, while the bottom left cell is the probability that the alarm rang but there was no burglar (a false positive).

If we substitute the values of Bayes' rule described above, we get:

$$P(burglar \mid alarm) = \frac{P(burglar, alarm)}{P(alarm)}$$

Where results is 0.09 / 0.19 = 0.47. So, even if the alarm is ringing, there is just a 47% probability that the house is actually being robbed.

The previous example illustrates the simplest case of applied BR, but it is also possible to combine several variables. One way to represent this kind of scenario is by expressing the variables in a directed acyclic graph, where the relation "Parent" stands for "May cause" and you can specify the conditional probabilities of a child given a parent's result. This graphical model is called a Bayesian Network.

2.2.1.2 Bayesian Networks

We can extended our alarm example further, by considering not only a burglar can trigger the alarm, but an earthquake also can (while there can still be false positives). Also, consider that we have 2 neighbors (Mary and John) who may call us whether the alarm is ringing or not. This problem is represented in figure **??**.

Some interesting question we can ask, given this scenario are:

If John calls saying the alarm is ringing but Mary doesn't, what are the odds it really is ringing?

If the alarm is ringing, was there an earthquake?

What are the chances that both my neighbors call, the alarm is ringing, but there is neither a burglary nor an earthquake?

This last example, for instances, would be calculated as: P(J, M, A, $\neg B, \neg E$) $= P(J|A) * P(M|A) * P(A|\neg B, \neg E) * P(\neg B) * P(\neg E) = 0.9 * 0.7 * 0.001 * 0.999 * 0.998 = 0.00062$.
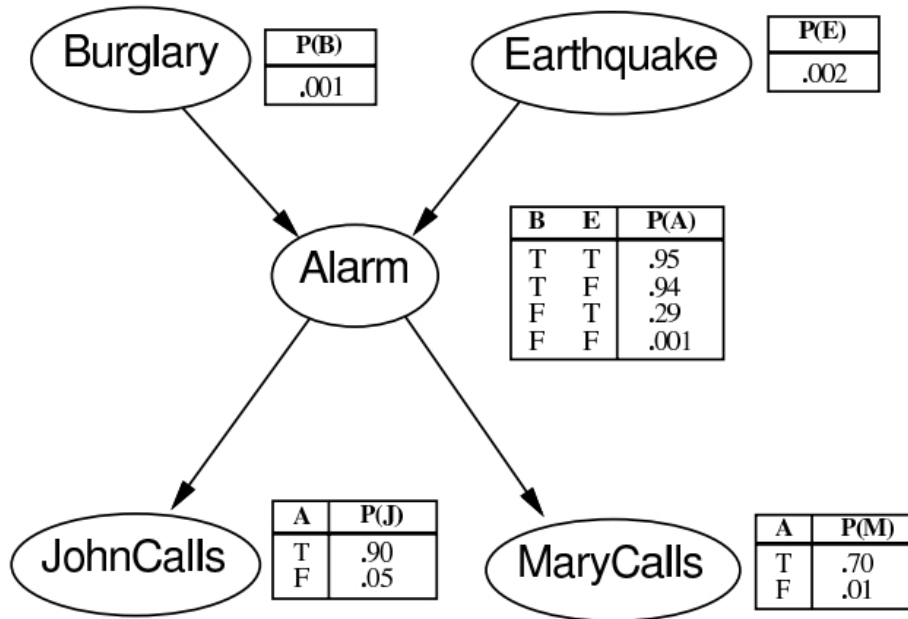
Figure 2.1: Belief network for the alarm problem [Wan02]

Notice how counter-intuitive this example is: the probability of there being an earthquake is about 32 times larger then there being an earthquake, the alarm ringing and the neighbors calling us, even if the conditional probabilities are reasonably high (0.95, 0.9 and 0.7). This is the result of the calculation of the joint probability being an highly combinatorial problem, which is yet another argument in favor of using PR rather than subjective heuristics.

2.2.1.3 Bayes' and data streams

In practical ML applications, it is often the case that there is an incoming stream of new data, rather than one-time batch calculations. BR can accomodate this way of thinking, which A. Downey called diachronic interpretation [Dow12], where diachronic means that something is happening over time (in this case the probability of the hypotheses, as new data arrives). In order to make sense of this definition, we may rewrite Bayes' rule as:

$$P(H \mid D) = \frac{P(D \mid H) P(H)}{P(D)}$$

Where:

- H: hypothesis

- D: data

- p(H): probability of the hypothesis before the new data is taken into account. Also called **prior**. It can either be calculated using background information or subjectively defined using

domain knowledge. Loses significance as new data is added, so its choice is not determinant to the model's performance in the long run.

- p(H|D): what we want to calculate, the probability of the hypothesis after considering the new date. It is called **posterior**.

- p(D|H): probability of the data if the hypothesis was true, called the **likelihood**.

- p(D): probability of the data under any hypothesis, called the **normalizing constant**.

Under this interpretation, you may continuously feed data into the model and see the probabilities getting updated. We will see more practical examples of this in section 2.2.2.

2.2.1.4 Beyond Bayesian Graphical Model

At first glance, someone who is learning for the first time about PR applied to ML, may think that graphical models such as the one presented in Figure 2.1 are the best there can be done in terms of using a graphical interface for solving this kind of problems and that the only thing is missing is an automated way to make the calculations.

While it is true we have never refered mentioned techniques or tools that automatically do inference over a Bayesian Network, there are several tools with that capability (including an R package [øjsgaard2013] or standalone tools [Res10]).

However, not all PR can be done via Bayesian Networks and not all graphical models are enough for complete PR [DL13]. PP are the largest class of models available, and there are also more algorithms for inference than just the calculation of joint probabilities (like we did in the alarm example), as we will discuss in Section 2.2.2.

Bayesian Networks are not the only kind of graphical model. Another one would be Markov Chains, which is yet another example of a model which is not able to represent all PR problems. This is clear when we realize that, while PPLs support a large number of different distributions (such as Normal, Laplace, Gamma, Half-Cauchy or *t*), all Bayesian Networks and Markov Chain can be represented in a PPL by just using Bernoulli distributions [Gor14]. We can see an example of such a translation in Figure 2.2.

<—- rever com base em prob inference for graphical models

2.2.2 Probabilistic Programming Languages

2.2.2.1 The Probabilistic Program-Model duality

A probabilistic program (PP) is an ordinary program (that can be written in mainstream languages such as C, Java or Haskell) whose purpose is to specify a probability distribution of its variables. This is done by sampling over several executions of the program. The only needed construct the language has to support, in order to be able to write a PP, is having a random number generator [DL13]. This whole concept couldn't be better explained than in this text by Freer and Roy, regarding Church (a PPL, which we describe in Section 2.4.1.2) but common to any PP:
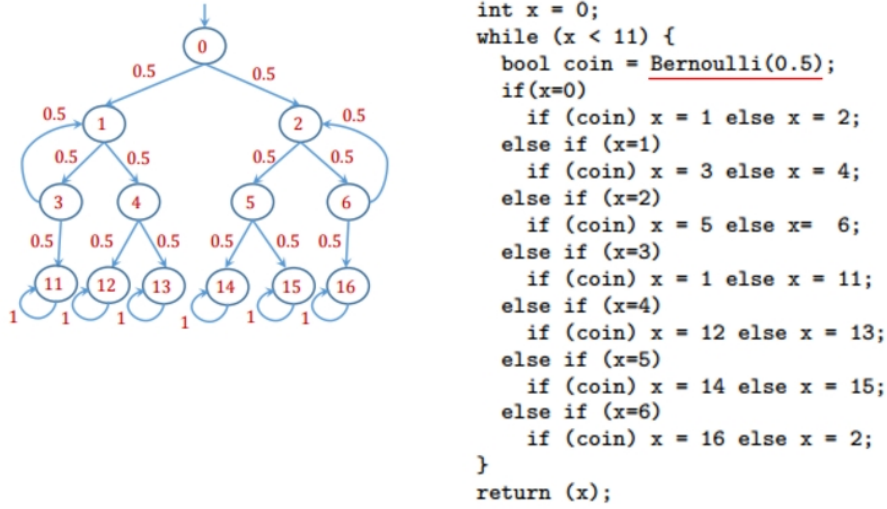
```
int x = 0;
while (x < 11) {
  bool coin = Bernoulli(0.5);
  if(x=0)
    if (coin) x = 1 else x = 2;
  else if (x=1)
    if (coin) x = 3 else x = 4;
  else if (x=2)
    if (coin) x = 5 else x=  6;
  else if (x=3)
    if (coin) x = 1 else x = 11;
  else if (x=4)
    if (coin) x = 12 else x = 13;
  else if (x=5)
    if (coin) x = 14 else x = 15;
  else if (x=6)
    if (coin) x = 16 else x = 2;
}
return (x);
```

Figure 2.2: Translation of Discrete Time Markov Chain to a PPL [Gor14]

"If we view the semantics of the underlying deterministic language as a map from programs to executions of the program, the semantics of a PPL built on it will be a map from programs to distributions over executions. When the program halts with probability one, this induces a proper distribution over return values. Indeed, any computable distribution can be represented as the distribution induced by a Church program in this way" [FR12]

One way to think about this notion is by considering that the program itself is the model. An example of the relation between a model (expressed in a PPL) and the implied distribution over its variables (obtained using an inference method) can be seen in Figure **??**, where a variable *flip* is set to be a Bernoulli distribution and *x* is defined in terms of *flip*. We can then see how the graphic of the inferred distributions of *flip's* and *x's* values looks like and confirm what was to be expected: for *flip's* values lower than 0.5 we see *x* follows a normal distributon, whereas for values greater than 0.5 it follows a gamma distribution instead. The goal of PP is to enable PR and ML to be accessible to most programmers and data scientists who have enough domain and programming knowledge but not enough expertise in probability theory or machine learning.

2.2.2.2 PPLs vs regular PLs

What is then, a Probabilistic Programming Language (PPL)? First of all, it can be a standalone language or an extension to a general purpose programming languag. We'll be analyzing examples of languages from either these categories in Section 2.4, but many more exist, such as as Figaro [RT15] (hosted in Scala), webppl [GS14] (embedded in Javascript) or Dimple [HBB+12] (has both a Java and a MATLAB API). The key difference between these languages and a PPL is the latter has the added capability of performing conditioning and inference [ADDJ03].

```
flip = rand < 0.5
if flip
    x = randg + 2     % Random draw from Gamma(1,1)
else
    x = randn         % Random draw from standard Normal
end
```
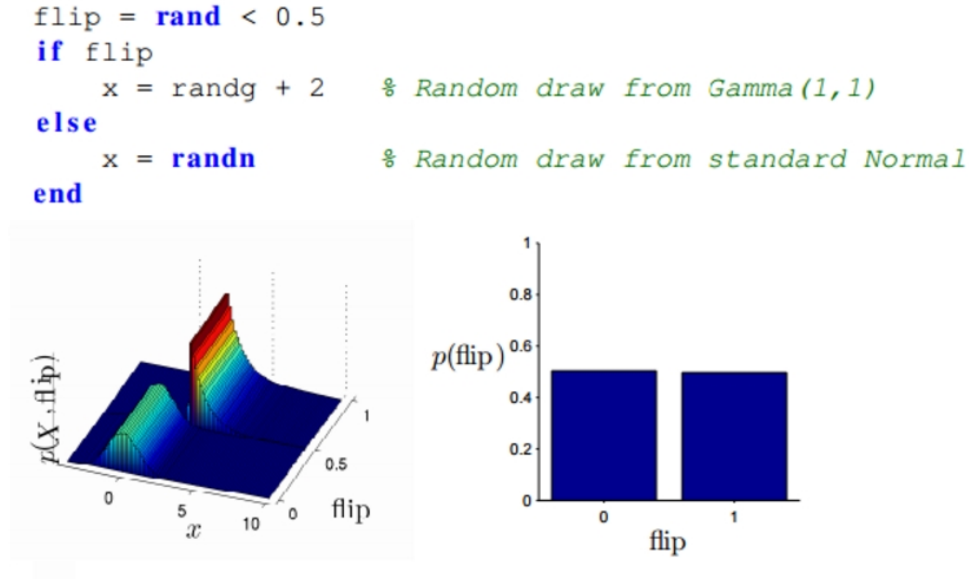

Figure 2.3: Implied distributions over variables [DL13]

Conditioning is the ability to introduce observations about the real world in the program. That way, you update the prior probability based on those observations. Consider the example in Figure 2.4 (which is a simplified version of how Microsoft applies PP in its Xbox matchmaking algorithm [MWGK12]) where the prior is a normal distribution with equal parameters for all players (shown by the graphic in the top). Then, it defines how the performance of the player is based on his skill (which at the initial point in time, is equal to every one of them) and proceeds to make several observations regarding games between them. Finally, it shows the inferred probability distribution of the posterior on the bottom graphic.

2.2.2.3 Inference

We said that a PPL empowers the user to formalize a model and then query for the probability distribution of its variables, which is automatically done via inference. While general-purpose language require you to write one-time inference methods that tightly coupled to the PP you are inferring on, PPLs ship with an inference engine suited to most PP programs you can write [FMR10].

An inference engine of a PPL acts similarly to a compiler in a traditional language: rather than requiring each user to engineer its own, which requires significant expertise and is a non-trivial and error-prone endeavour, every PPL has one incorporated. Olivier Grisel called this separation of concerns between the language and its inference engine "openbox models, blackbox inference engine" [Gri13].

Having the engine work as a separate model opens up a myriad of new possibilities, mainly in the form of knowledge and tool sharing, as we have seen in the past in the compiler space. Examples of this would be new compiler compiler and interpreter techniques (such as working towards scalability or parallelization), optimizers, profilers or debuggers.

```
// prior distributions, the hypothesis
let skill() = sample (Gaussian(10.0,20.0))
let Alice,Bob,Cyd = skill(),skill(),skill()
// observe the evidence
let performance player = sample (Gaussian(player,1.0))
observe (performance Alice > performance Bob) //Alice beats Bob
observe (performance Bob > performance Cyd) //Bob beats Cyd
observe (performance Alice > performance Cyd) //Alice beats Cyd
// return the skills
Alice,Bob,Cyd
```
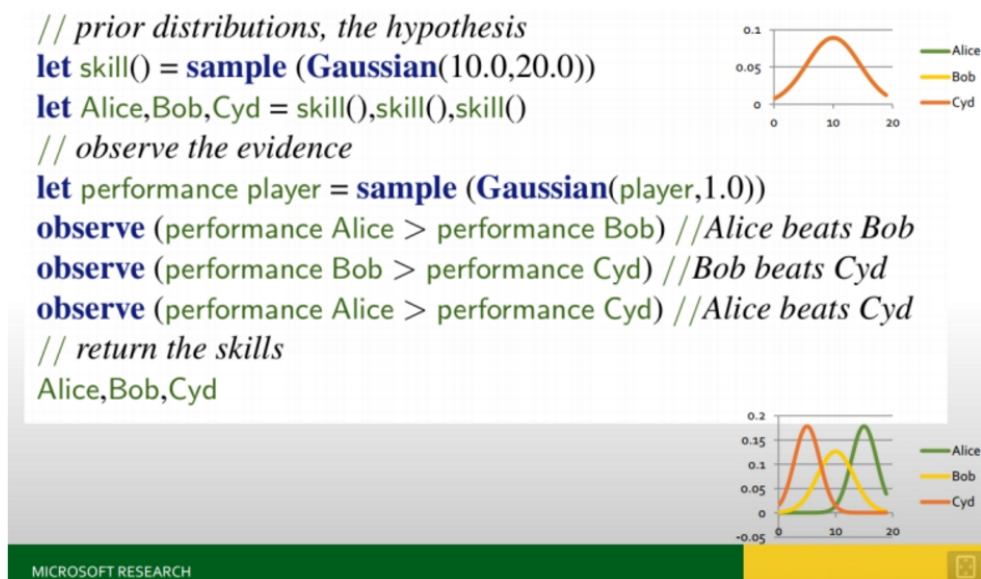
Figure 2.4: Microsoft Xbox Live True Skill [MWGK12]

Another great advantage of having a modular inference engine is that we can try different inference algorithms and pick the one that best fits the problem at hand. When analyzing if algorithm suits a certain use case, there are certain characteristics worth noting [Min99]:

- Determinism - in equal initial conditions, an algorithm always yields the same result.

- Exact result or approximation

- Guaranteed convergence - an algorithm may or not be guaranteed to reach a result at some point in time. If not, it's possible that it will run forever.

- Efficiency - related to how fast can it reach a result.

Microsoft's Infer.NET provides three inference algorithms [Res]:

- Expectation Propagation - deterministic, provides exact solutions only in some cases, is not guaranteed to converge and is labelled as "reasonably efficient".

- Variational Message Passing - also deterministic, but always gives approximates results and is guaranteed to converge. It's considered to be the most efficient of the three for most cases.

- Gibbs sampling - non-deterministc, may be able to reach exact result if given enough time to run, has guaranteed convergence and is regarded as not-so-efficient as the other two.

We can divide the three algorithms into two categories: variational bayesian methods [WBJ05][Min99] and sampling methods (in the case of Gibbs sampling, it's based on Markov chain Monte Carlo) [ADDJ03]. The main difference, as far as the end-user is concerned, is that variational methods

provide faster results but are subject to bias, whereas sampling methods have the potential to pro-
duce more accurate results (the downside being it's slower and its convergence is hard to diagnose)
[SAC⁺10].

Please notice than none of these algorithms provides the same kind of exact solution as the
calculation of the joint probabilities we did in Section 2.2.1. The reason for that is that calculating
an exact solution takes time exponential in the number of variables to run, even if we have smarter
algorithms than the naive calculations we did [ZP94].

2.2.2.4 Openbox models

When compared to traditional machine learning methods (such as random forests, neural networks
or linear regression), which take homogeneous data as input (requiring the user to separate their
domain into different models), probabilistic programming is used to leverage the data's original
structure. This is done by empowering the user to write his own models while taking advantage of
a re-usable inference engine. Olivier Grisel called this combination "Openbox models, blackbox
inference engine"[Gri13].

Rather than using most of his time performing feature engineering (that is, trying to fit the
problem and the data into an existing model), the user will have the tool necessary to design the
model that best fits the domain he is working on. Plus, it provides full probability distributions
over both the predictions and parameters of the model, whereas ML methods can mostly only give
the user a certain degree of confidence on the predictions.

2.2.3 Conclusion

Summarizing, PPLs are a step forward in using PR to solve ML problems since it helps overcome
the difficulties in using PR in real world problems. This is done by adding automated general
inference on top of a precise specification (the program), where in the past models were com-
municated using a mix of natural language, pseudo code, and mathematical formulae and solved
using special purpose, one-off inference methods.

This encourages exploration, since different models require less time to setup and evaluate,
and enables sharing knowledge in the form of best practices and design and development patterns.

However, using a full fleged programming language might still be an entry barrier. We want to
help statisticians and data scientists alike to learn faster and be more productive using a PPL in a
way similar to the tools they are accostumed to. In order to do so, we'll be combining a PPL with
Visual Programming (Section 2.3).

2.3 Visual Programming

Visual Programming (VP) can be defined as "any system that allows the user to specify a program
in a two-(or more)-dimensional fashion." [Mye86]. In a textual programming language, even
though there are two dimensions (one being the text itself, and the other the optional line-breaking

characters), only one of them has semantics, as the compiler processes text as a one-dimensional stream.

Examples of systems with aditional dimentions are ones that allow the use of multidimensional objects, the use of spatial relationships, or the use of the time dimension to specify "before-after" semantic relationships [Bur99].

Research has identified several advantages in the use of VP, such as natural expressibility, easy readability and interaction, language independence (though this is not applicable to the work of this thesis, as detailed in 2.3.1), programming at higher levels of abstraction or rapid prototyping [Jam14], which is achieved by providing immediate visual feedback [Gre95].

The advantages of programming at highers levels of abstraction are known, and one of them is it exposes users who are not used completely fluent in programming to a reduced number of concepts [Gre95], while decreasing the verbosity of programs, which can even be useful for seasoned programmers [Mye90]. It also reduces the importance of being familiar with syntax, a common cause of difficulty of adaptation among less experienced programmers when learning a new language [CTB86][CWHH05]. This difficulty of translating ideas into syntactictically correct statements can also be solved by finding alternative ways to communicate instructions to the computer [KP05].

So, the point of using a VP tool to aid in programming is to overcome the difficulties that many people have when learning a conventional language [LO87b]. It has been shown this approach can help users without prior, or little, programming experience to create fairly complex programs [Hal84]. This is specially true within certain small domains, where the language can be tailored for a subset of tasks rather than trying to be suitable for all kinds of applications [KP05]. An example of such domain would be developing ML applications resorting to probabilistic programming. Excel-like (spreadsheet) tools are a prime example of this, where the following benefits were identified [Amb87][LO87a]:

- The graphics on the screen use a familiar, concrete, and visible representation which directly maps to the user's natural model of the data

- They are non-modal and interpretive and therefore provide immediate feedback

- They supply aggregate and high-level operations

- They avoid the notion of variables (all data is visible)

- The inner world of computation is suppressed

- Each cell typically has a single value throughout the computation

- They are non-declarative and typeless

- Consistency is automatically maintained

- The order of evaluation (flow of control) is entirely derived from the declared cell dependencies

Smith and other authors claim that the human thought process is clearly optimzed for multi-dimensional data [Smi77][CC86], so all the aforementioned advantages can be explained by how graphical programming is closer to our mental representation of problems when compared to a textual interface [Car02]. As said by Fischer, Giaccardi, Sutcliffe and Mehandjiev:

> "Text-based languages tend to be more complex because the syntax and lexicon (terminology) must be learned from scratch, as with any human language. Consequently, languages designed specifically for end users represent the programmable world as graphical metaphors ... (*aim to*) reduce the cognitive burden of learning by shrinking the conceptual distance between actions in the real world and programming." [GEGN04]

This idea as been tested in an empirical study: in a algorithms course of the United States Air Force academy, students consistently show a preference for solving problems visually, while it also seems that doing so helped them achieving better scores in problem-solving exercises and overperform their colleagues who used a regular programming language [Car02]. However, shortcomes of using VP were also identified, as we will discuss in 2.3.4.

2.3.1 Visual Programming Environment

Boshernitsan proposed a classification scheme for VPLs [BD04] that divided VPLs in purely visual languages (PVLs), hybrid text and visual systems, programming-by-example systems, constraint-oriented systems and form-based systems.

In the context of this dissertation, as we want to leverage the advantages of both a VPL and a PPL while avoiding implementing a PPL, there is no other choice than using an hybrid text and visual system. We will be calling this system a visual programming environment (VPE). The difference between a VPE and a purely visual language (PVL) is that, while a PVL is a language *per se* (meaning that it there is a direct mapping between graphics and execution), VPEs offer a middle ground between regular textual languages and PVLs: they provide a graphical interface that can be used to generate code for a target language [Bur99]. An example of a PVL would be MIT's Scratch [RMMH$^+$09] and one for a VPE is the Eclipse IDE plug-in WindowBuilder [Fou16].

If applied correctly, VPEs can help addressing some of the issues raised by critics of VP (detailed in section 2.3.4), such as VP's inability to solve real-world large-scale problems. This is done by applying VP to only a subset of the program, making it possible to combine a general-purpose programming language with the advantages of VP [Bur99], since the code generated by a VPE can be seseamlessly integrated in any project built with its target language.

Concretely, VPEs are able to overcome the tradeoff of control for simplicity commonly made by VPLs: even if a certain idiom cannot be represented by its graphical form, the user can later edit the generated code to include it. This makes it possible to design scalable programs, both in terms of performance (since the user can still access all the target language's low-level features) and development (because all the advantages of programming visually are still present).

2.3.2 Visual Dataflow Programming

2.3.3 Evaluation

Even if the goal of VP is clear (to make programming more understandable, ensure correction and be faster to do) the best way to achieve it is still a matter of discussion. Like in the design of any programming language, there are some best practices to guide the design of a VPL, some of which have been identified by Burnett [Bur99] and are listed below.

- Concreteness - it's oposite of abstractness. The program express values and instances of values rather than meta-information such as types or classes.

- Directness -

- Explicitness - everything that there is to be known about the program can be easily understood by looking at the graphical representation, the user does not need to infer semantics by himself

- Immediate visual feedback - every change to the program should be immediately propagate to change in the affected output. Spreadsheets are an example of this.

However, these are just guidelines and do not guarantee the efficacy of a VPL. Whitley and Blackwell [WB97] said that, because the design decisions in a VPL lack formal basis, the only way to assess if they really contribute to facilitate the programmer's cognitive processes while programming is through empirical studies. In their studies, they have found that while subjects cited ease of learning as a benefit of VPLs, they opinions differed when asked if using a VPL had a positive impact in productivity during a project.

They also claim that there is a gap between academia and industry programmers. In contrast to the first group who tends to focus on theories of cognition to justify the use of VPL, the second is more interested in potential improvements in "potential improvements in productivity that arise from straightforward usability issues".

Some metrics that can be taken into account when assessing a VPL's efficacy are learning time, execution speed and retention [Mye90].

2.3.4 Criticism

In this section, we'll be trying to summarize some of the criticism made to VP, while proposing solutions to some of the issues mentioned and discard some of the others as non-applicable in the context of this thesis.

There are people who claim VPLs lack visual abstraction mechanisms that are as effective as those offered by text-based languages, so they are not well-suited to develop large applications [Jam14]. Some of the techniques currently used in real-world software development include iterative design and interactive prototyping, two principles that are promoted by the usage of VPLs.

Also, it has been shown that the richness of the visual paradigm introduces new ways of approaching programming problems, particularly for those not trained in traditional software development methods [Jam14] (such as data scientists, which constitute the target audience of this work). Studies also shown that fairly complicated algorithms, such as garbage collection, could be described graphically [Mye90].

Green also discusses, contrary to some other evidence we discussed before [Car02][Bur99], how lab studies failed to collect evidence in favor of the productivity gain of VPLs, even though it admits user like and use them [Gre95]. He also points how that VP systems "do nothing that can't be done as well or better with straight text" and identifies the real issues as "how layout and locality can be used to convey meaning". According to our definition of VP, every system that allows the user to express himself in more than one dimension (such as using layout and locality, as proposed by Green), so it seems that the proposed alternative could be a VPL.

In his *"No silver bullet - essence and accidents of software engineering"* paper, Brooks says that *"A favorite subject for PhD dissertations in software engineering is graphical, or visual, programming—the application of computer graphics to software design.... Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will. In the first place, ... the flowchart is a very poor abstraction of software structure.... It has proved to be useless as a design tool.... Second, the screens of today are too small, in pixels, to show both the scope and the resolution of any seriously detailed software diagram.... More fundamentally, ... software is very difficult to visualize."* [Bro86]. While one may be tempted to be conviced by Brooks' initial rethoric, what he wrote does not seem to apply to the work of this thesis because: a) we won't be using executable flowcharts, but rather a dataflow VPE approach, as described in **??** and b) screens are getting bigger and with higher resolutions [], a low-end screen by today's standards would be state of the art in 1986, when Brooks wrote that. The final claim that software is hard to visualize, backed by Dijkstra's letter where he states that *"I was recently exposed to ... what ... pretended to be educational software for an introductory programming course. With its "visualizations" on the screen, it was ... an obvious case of curriculum infantilization.... We must expect from that system permanent mental damage for most students exposed to it."* [D$^+$89], is contrary to many studies done in cognitive science [LO87a][CTB86][CWHH05].

Myers admitted that the key for a successful application of VP to a real-world problem was to identify "appropriate domains and new domains to apply these technologies to" while recognizing that we have already witnessed how VP can help non-programmers work in limited domains [Mye90].

We do believe that data mining can be one of those domains (as shown by the popularity of RapidMiner) [Pia15], so it would make sense to extend the current state of the art with a tool that combines VP and PP.

solução: para ultrapassar problemas comuns de VP em sistemas reais, larga-escala, usar apenas VP em partes especificas do projeto onde VP possa ser utilizado com sucesso. usar para desenhar GUIs foi um sucesso (android, windowbuilder)

fazer programar mais facil para audiencia especifica [Bur99]

Myers also identified some of the problems that are yet to be solved regarding VP [Mye90]:

- **Difficulty with large programs or large data**. Almost all visual representations are physically larger than the text they replace, so there is often a problem that too little will fit on the screen.

  We intend to solve this issue by studying the application of some techniques that provide a greater level of abstraction in order to be able to transmit more semantics while avoiding the layout to get cluttered with details [BBB⁺95].

- **Poor representations**. Programs are hard to understand once created and difficult to debug and edit. The larger the size, the worse this problems becomes.

  This is related with the previous point and the proposed solution is similar.

- **Need for automatic layout**. When a program gets too large the layout becomes too hard to manage, a single addition of a piece may oblige the user to move a great number of blocks in order to avoid collisions and preserve readability. One way to deal with this would be to automatically generate an attractive layout.

  This problem is out of the scope of this thesis, although it certainly seems important for the future of VPL.

- **Lack of formal specification**. Currently, there is no formal way to describe a VPL such as the Backus-Naur Form, even if there is some work towards one was already been made [SK88].

  Again, this is out of scope, even if we will make an attempt to specify a grammar that defines the boundaries of what would be a valid graph.

- **Lack of Portability of Programs**. A program written in a textual language is as portable as a text file but VPLs require special software to view and edit.

  While the implementation of the VPE's frontend to be done in this thesis is still a matter of study and is undecided, there is the possibility of building one using the HTML/CSS/-javascript stack, making it portable across browsers for view and edition. It is also our aim to provide an intermediate representation in a data-representation format such as JSON or XML, so that a program built in a certain frontend can be processed by any other.

- **Tremendous difficulty in building editors and environments**. In 1990, each graphical language required its own editor and environment, and there were no general purpose VPL editors.

  Currently there are alternatives of re-usable frontends for VPLs, such as Blockly [fE15] or GoJS [Sof15].

- **Lack of evidence of VPLs' worth**.

This issue was discussed in section 2.3.3, and we conclude that within certain domains (such as VP applied to PP) and target users (unexperienced programmers), there may be benifits in productivity when using a VPE.

Another question that arises is how to represent and manipulate arrays. In the empirical study made by the creators of RAPTOR, students performed statistically significantly worse on the array question when using a VPL [Car02]. This is something to consider in the future, to investigate if handling arrays functionally (by considering them as immutable values where common functions over iterable data structures could be performed, such as map, filter and reduce) could improve users' usage of the VPL.

One of the concerns of another empirical study's respondents was that high-level VPLs might deny them access to the low-level facilities of the machine that are so important in PC programming [WB97] but, as stated before, this problem is alleviated (if not completely eliminated) by using a VPE rather than a purely visual VPL, since the user can later edit the generated code, where he has access to all the language's features.

2.4 State of the Art

The purpose of this section is to try giving an overview over the existing tools currently used in either VP (purely visual VPLs or VPEs) or PPLs.

2.4.1 PPLs

2.4.1.1 WebPPL

[GS14]

2.4.1.2 Church

[?]

2.4.1.3 Infer.NET

2.4.1.4 PyMC

2.4.1.5 Figaro

[?]

2.4.2 Tools using VDP

2.4.2.1 NoFlo

NoFlo is a visual open-source implementation in JavaScript of Flow-Based Programming [Ber] (which is a form of dataflow programming) and was designed for general-purpose programming,

Figure 2.5: Example of node expansion in NoFlo [Ber]

even if it is more suitable for web programming (being written in JavaScript it has easy access to DOM manipulation capabilities), there are examples of it even being used for controlling a drone.

From experimenting with the tool, we found some characteristcs that serve as a lesson of what works well and what doesn't:

- A node has the least information possible, yet it can be clicked to show details, such as some usage notes or seeing some inputs or parameters. An example of a common NoFlo layout with a node expanded can be seen in Figure 2.5. This helps saving space in the layout while maintaining flexibility.

- You can search for a block by its name (see Figure 2.6).

- There is a minimap that helps navigating the screen, so we can have an higher level view of the layout 2.6).

- Weak typing makes it confuse to use some of the nodes, it is unclear when we can connect input and outputs with mismatching labels, because sometimes they are compatible and other times they aren't.

- Sometimes errors appear immediately, others we can just see them in runtime. This would be fine as long as it was clear that there was an error, but that is not the case because the messages blend in with the UI and are not easily noticeable.

- When choosing new blocks, it is often hard to understand what is their functionality because most nodes have no description other than their name.
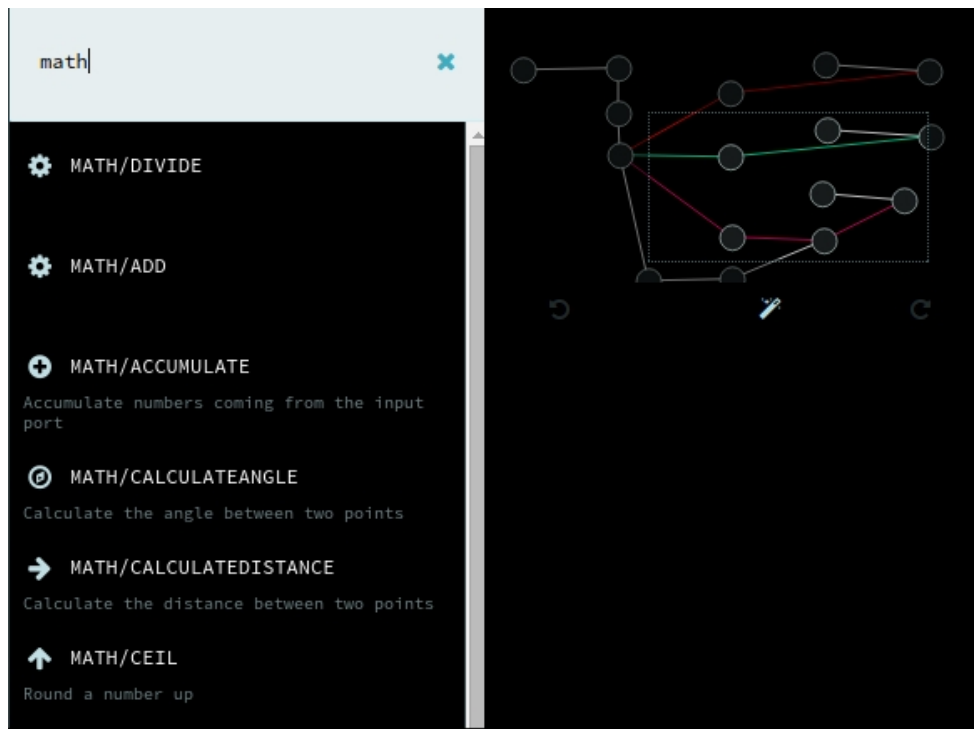
Figure 2.6: NoFlo node search and minimap navigator [Ber]

- Blocks have icons, which presumably provide some visual information about their functionality (see Figure 2.6), but is hard to understand what the difference between the various icons is. What seems to be a good idea, to have icons in blocks to identify them by functionality, as the opposite effect and negatively affects usability.

The last four items can convice the reader of the following conclusion: in VPLs, explicit is better than implicit. Clarity can arguably make all the difference between a language that can be intuitively used and one that, despite all its virtues, could take some more time to master.

2.4.2.2 RapidMiner

According to the popular data mining portal KDnuggets' latest tool popularity poll [Pia15], RapidMiner is the most popular visual tool to use in applied machine learning. It is also the most complete, featuring 1500 different blocks [Rap].

The product has too many features to be able to deeply analyze each one of them, including cloud and repository capabilities for storing and sharing work, mechanisms for deploying models developed locally to a production server or an API for developing new blocks. RapidMiner is actually more of a suite rather than just a tool, so in our analysis we'll restrict ourselves to the design of a model.

The most differentiating factor between RapidMiner and the other VPLs we studied is that it splits its view between process and results. In the first one, the user can design the model and
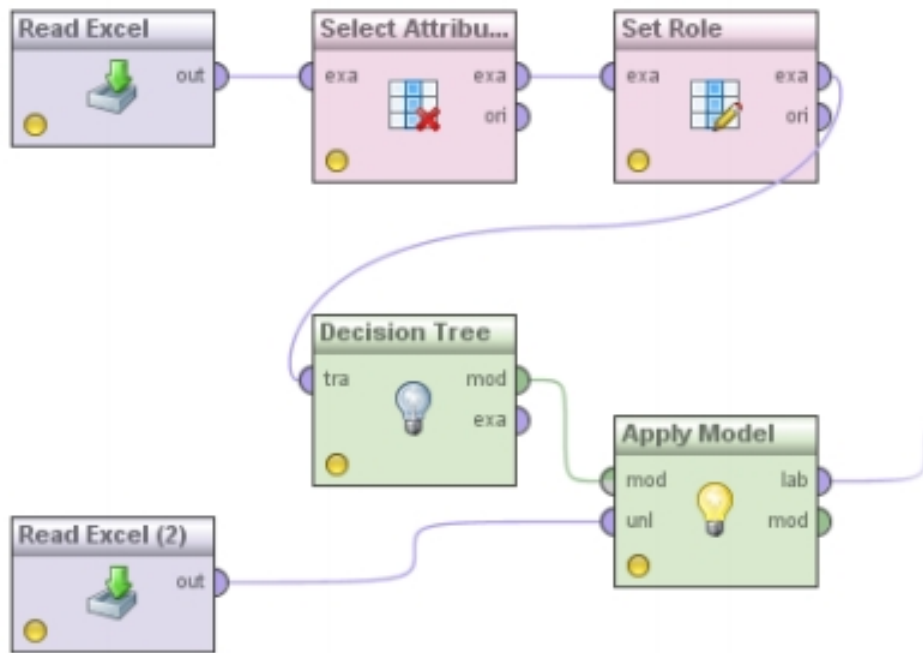
Figure 2.7: Classifier in Rapidminer [Rap]

then he must run it and go to results to see the output. Even if goes against the principle of providing immediate feedback to the user advocated by some VPLs' designers (as discussed in 2.3), it does seem to make sense in its use case. The reason for that is that production machine learning applications running batch jobs can take a reasonable amount of time to run, and the analysis of its results before changing the model can also take some time. This is different from people using 3D editors (like Blender), where they the trial and error cycle is much shorter, and users need to get feedback about their changes much more often.

Having no need to be continuously looking at the output reduces the importance of it being shown in the same space as the program we're designing, and we can save space in the layout by moving it to a different tab. On the other hand, RapidMiner is flexible enough that allows the user to place both the process and the results windows side by side, so it really is up to the user to decide how important is it to be seeing everything at the same time. A tool which doesn't support moving windows and tabs must make a choice whether or not to have the output shown alongside with the program.

Other strenghts of RapidMiner include having error indicators in each block and thorough descriptions of each node's functionality, input, output and parameters, including examples. It also provides visual clues about a node's functionality by using not only meaningful icons but also colors. This can be observer in Figure 2.7, where file reading nodes are grey, model-related ones green and those related with data preparation are in pink.

Something that does seem to work particularly well for someone using the tool for the first time are the abbreviated labels in the ports. They are cryptic enough that an unexperienced user
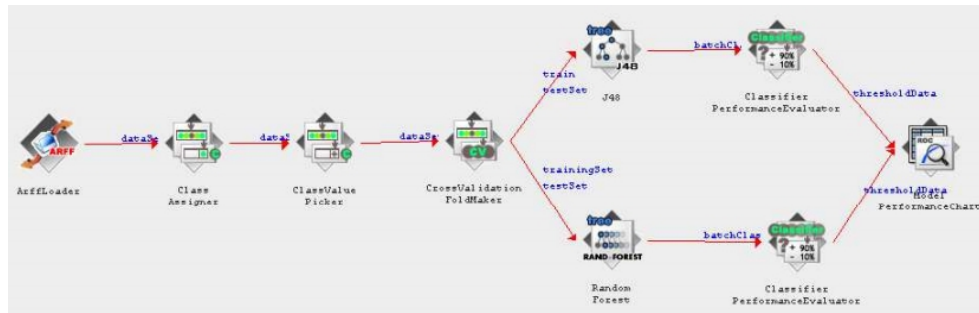
Figure 2.8: ROC curve in Weka Knowledge Flow [HR08]

has to click the node and read the description to understand what they are. Maybe it would make sense to either write the full label for the sake of clarity, or just don't show anything at all.

2.4.2.3   Weka Knowledge FLow

Weka Knowledge Flow is yet another frontend (others being APIs and a GUI Explorer) to WEKA's core algorithms [HR08].

In this tool, inputs are not connected to specific ports, but rather to a block. The semantics of the connection is expressed on the edge's label. This can save some space in the layout as the nodes can be smaller, but in order to understand a node's input, the user has to waste time following all the edges backwards until he finds the label, which doesn't seem a reasonable tradeoff.

Weka Knowledge makes a good use of icons to differentiate between different node types. Not only is it easy to the user to distinguish between source, processing and sink nodes, but also between the different processing nodes' categories (such as filters, classifiers and clusterers).

2.4.2.4   Blender Composite Nodes

Blender Composite Nodes is part of the Blender open-source 3D computer graphics software and provides a way for the user to visually build (through a dataflow representation) a script that can be applied to images or Blender scenes [Wik].

It incorporates common VDP idioms such as source nodes (such as render layers or RGB), processing nodes (where you can apply blur, shift an RGB curve, etc) and sink nodes (it's the way to finalize a script and save, or display, the result). It is interesting to note that Blender Composite Nodes does not incorporate processing nodes with filter semantics [1], which is an example of a VPL that intentionally did not incorporate a common idiom in VDP because it does not make sense for its domain.

There are several interesting characteristics that are worthy highlighting:

- Composite Nodes assigns a color to each type, so that the user can easily understand the type of each input or output (see Fig. 2.9).

---

[1] We mean by filter that the result is a subset of the input. Blender has graphical filters, which are analogous to a functional map operation.
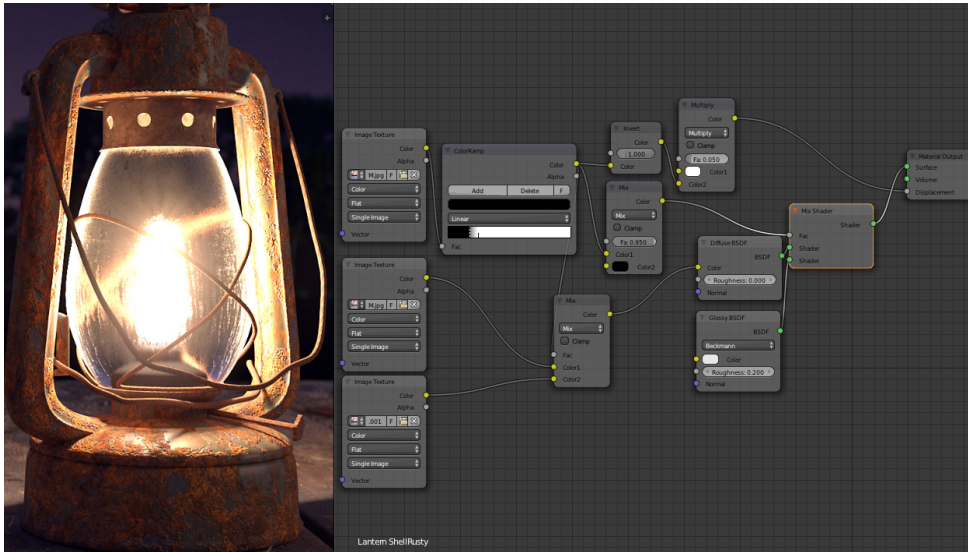
Figure 2.9: Script for texture transformation in Blender [Wik]

- Implicit conversions for some of the types. For instances, a conversion from a vector (which are always three-dimensional) to a value yields the mean of the vector's elements. In figure 2.9 we can see that some values are connected to a different type (among others, the first input node's color is connected to a simple value named 'Fac'), which is valid behavior.

- Incorporating a custom interface within each node. This helps reducing the space occupied by the program, since there is no need to create inputs for every single input in a processing node. It also simplifies the VPL's usage, because since some types can only exist within certain nodes, and by not being able to instantiate them outside those nodes, we reduce the possible choices for input nodes. An example would be the image type enumeration, that we can see in figure's 2.9 input nodes.

- Node groups which are essentially an encapsulation of a set of nodes, resulting in another node that can be re-used, help the user create his own set of abstractions. Our texture transformation example could be saved into a node group, so that we can use it in another script without needing to copy everything from one place to the other or cluttering the new script's interface with several new nodes. The concept is analogous to functions in programming.

2.4.3 VPEs

In this section we'll be describing three visual programming environments. Meaning, tools where the user can express the program in a graphical manner that will later be translated into valid code.
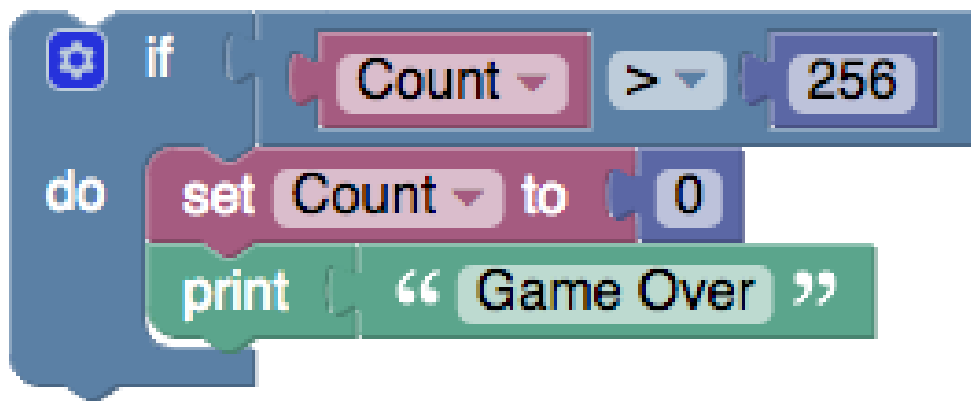
Figure 2.10: Blockly example [fE15]

2.4.3.1 Blockly

Blockly is more than a VPE, it actually is a library for building VPEs [fE15]. It is fully open source, runs on the browser (client-side) and is completly extensible.

Blockly does not adopt the VDP paradigm, it instead uses a system of blocks similar to a puzzle, as you can see in Figure 2.10. Another useful feature is basic type checking, so that users can't connect blocks that wouldn't make sense, such as converting a number to uppercase. The way Blockly is built, it is impossible for the user to produce an invalid program.

According to the authors, the reason why they didn't choose to adopt VDP is that they believe the generated code doesn't look like the VDP graph, making it harder from a user who later transitions from the tool to textutal programming to adapt.

Tt is possible to define new blocks as well completly controlling how generated code will look like, as well as defining new types. These features seem to be enough that Blockly can be considered a framework complete enough to be suitable for building most VPEs and, because it is open-source, it can be unlimitedly extended.

2.4.3.2 Viskell

2.4.3.3 WindowBuilder

One domain where VPLs have been successfuly applied is in UI design. Instead of going through the tedious task of writting boilerplate code (often having to memorize complicated function signatures) and needing to memorize APIs, users can just drag and drop elements on the screen.

The Eclipse IDE provides several plugins on its marketplace, one of which is the Window-Builer plugin [Fou16] to build Java SWT/Swing UIs.

WindowBuilder bi-directional and WISIWIG tool (What You See Is What You Get), meaning that it generates code that faithfully represents the drawn UI and it is also possible to directly edit the code and see the updated result back in the design view.

Figure 2.11: WindowBuilder's design view [Fou16]

Since defining UIs is a very different domain from PP there are not many features that we can use from WindowBuilder in our future VPE. The exception is the bi-directionality of code, which could be useful in certain scenarios.

2.4.4 VIBES

VIBES is a tool, developed by John Winn during his PhD, that allows variational inference to be performed automatically on a Bayesian network [WBJ05] defined graphically. It was written to serve as proof of concept for the variational message passing (VMP) algorithm, so it has reduced focus on human-computer interaction. Other than its representation of determinisc inputs with a different edge style than undeterminisc ones and usage of plates to represent loops, from a usability point of view it's not very interesting.

2.4.4.1 WinBUGS

Similarly to VIBES (described in 2.4.4), WinBUGS is a framework with a graphical component which s part of the Bayesian inference Using Gibbs Sampling (BUGS) project. It allows the user to, graphically or textually, define a model and automatically perform inference on it (unlike VIBES, the inference algorithm is Markov chain Monte Carlo) [LTBS00]. Its interface is very similar to VIBES and the only difference between them, besides the algorithm used, is that WinBUGS provides a bi-direcitonal conversion between graphical and textual representations.

In Figure 2.13, we can see that WinBUG's textual representation of a simple model can already be harder to understand than the mathematical formalism it represents. This is exactly the problem we want think it can be solved with a proper VPE.

2.5 Conclusions
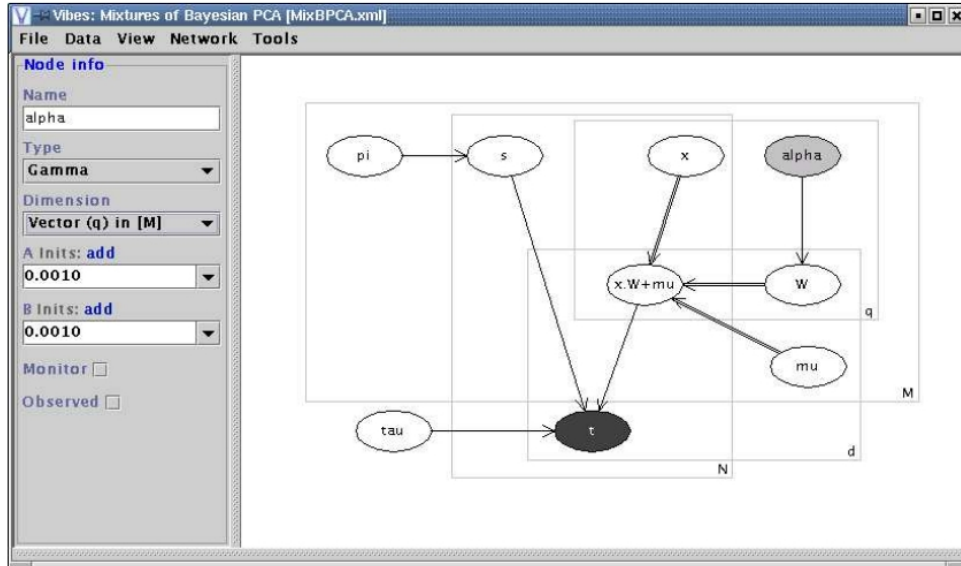
Figure 2.12: Example of a model using VIBES [WBJ05]

$$\left\{ \begin{array}{ccl} \alpha,\ \beta & \sim & \mathrm{NORMAL}(0,10^{-6}) \\ \tau & \sim & GAMMA(10^{-3},10^{-3}) \end{array} \right. \quad,\quad \left\{ \begin{array}{ccl} Y_i & \sim & \mathrm{NORMAL}(\mu_i,\tau) \\ \mu_i & = & \alpha + \beta(x_i - \bar{x}) \end{array} \right.$$

```
model
    {for (i in 1:N)
        {Y[i]~dnorm(mu[i],tau)
         mu[i]<-alpha+beta*(x[i]-mean(x[]))}
    alpha~dnorm(0,1.0E-6)
    beta~dnorm(0,1.0E-6)
    tau~dgamma(1.0E-3,1.0E-3)
    sigma<-1/sqrt(tau)
    }
```

Figure 2.13: Example of a model translated to WinBUGS textual form [Cra04]

Chapter 3

Solution prototype

3.1 The problem it solves

As already described in Section 1.3, this dissertation aims to solve the problem of PPLs having too much of a steep learning curve for someone who is unexperienced in programming, even if that person would've enough knowledge in statistics to leverage PPLs' power in applied machine learning. We propose to do so by developing a visual programming environment for a PPL that is user-friendly and yet flexible enough to be able to program solutions for non-trivial problems.

Since the existing work joining VP with PP is still nonexistent, the notable exception being VIBES and WinBUGS (even if it has its shortcomings, as described in 2.4.4), we have identified margin for improvement.

VPEs have been successfuly applied to other domains, and considering previous studies in VP that suggest it is well suited for both limited domains and unexperienced programmers, it is the author's belief that development in PPLs would also benefit from such a tool. Therefore, we intend to validate the following hypotehsis:

> "When a user instead of specifying a model textually, produces the model via a graphical representation that automatically translates itself into executable code, he will do so in a shorter amount of time, make fewer errors (both during development and regarding the final solution) and will reach a final representation that is more understandable and thus easier to maintain."

Because there isn't a standard method for evaluating if a language is easier to work with than another, the question arises on how to evaluate success. The optimal way to do so would be to make an empirical study. However, this requires having a tool mature and stable enough so that the study's results can be considered reliable. The goal of this dissertation is to assess several ways how such a tool could be built, resulting in a prototype; it is not to develop production-grade software.

An alternative is to gather examples of probabilistic programs expressed in a PPL (either the one who chose to serve as backend, or another with similar capabilities) in its traditional textual

form, translate them into our graphical language, and compare the two forms. This what we will be doing.

3.1.1 An empirical study

The way the study would work would be by compare how fast an user can define a model for a given set of problems when using a VPL in a regualr way or through our graphical interface. We'd then count the number of syntax and type erros done with each representation. By selecting users who never used the given PPL, we could not only measure execution speed but learning time.

It would also be valuable to assess, not only how fast can someone develop with either of the alternatives, but also the quality of the output. That could be done in two steps: starting by verifying if the program correctly models the problem and then asking the participants in the study if they believe the model they have developed graphically is easier to understand than its textual counterpart (and vice-versa). Although subjetive, we believe getting the participants' opinion regarding the output quality could provide valuable insights in order to understand if VP can really enhance an user's experience when using a PPL. Another method we could use to help us make an assessment of the validity of the hypothesis would be asking participants questions regarding usability. Even if it may seem redundant, since we would already have the time measurements, it is a way of identifying strenghts and weaknesses with reduced granularity.

3.1.2 Target audience

The resulting tool is aimed at people with knowledge in statistics who are unexperienced programmers. This may include data scientists, researchers, mathematicians or staticians. In short, anyone who would apply PP to problem solving but are not fluent in programming.

3.1.3 Expected contributions

By the end of this work, we would expect to have built a VPE for PPLs that can be extended with more PPLs, even if we are only implementing the adapter for one. By doing so we will: have a platform that enables other people to experiment and make usability research on, define a visual language that can be applied to PP in general, identifying what works and what to avoid, and ultimately assessing the viability of applying VP to PP to enhance end-user's productivity.

3.2 Outline

3.3 Architecture

3.4 Implementation

"controlled dataflow vpl"

3.4.1 Picking a front-end

3.4.1.1 Blockly

3.4.1.2 GoJS

3.4.1.3 Custom implementation

3.4.2 Picking a target PPL

3.4.2.1 WebPPL

3.4.2.2 PyMC

3.4.2.3 Infer.NET

3.4.3 Defining a Grammar

3.4.4 Handling Cycles and Conditionals

3.4.5 Handling objects and arrays

3.4.6 Inverse Compilation

3.4.7 Instant visual results

3.4.8 Opening to extension

3.5 Tutorial

3.6 Conclusions

Solution prototype

Chapter 4

Evaluation

4.1   Practical examples

4.2   Problems solved

4.3   Problems detected

4.4   Conclusions

Evaluation

Chapter 5

Conclusions

5.1 Future Work

In the time I'll be dedicated to accomplish what I have proposed to do, developing a visual programming environment and gathering knowledge of what works best in a visual representation and what doesn't, I'll be following the work plan presented in Figure 5.1.

During these nearly 5 months, the plan is to start weighting the pros and cons of using certain PPLs and picking one of them to be used. The next step would be picking an apropriate frontend, deciding between Blockly, a custom HTML+Javascript solution, or other alternative. Then, I'll start by gathering the fundamental language constructs that must be included, defining how they could be represented in the visual language and developing the frontend and code generation for those constructs. From this point onwards until the end, I've planned to make iterations: of choosing new language features to integrate, adapt the frontend and code generation for them, converting textual examples to the visual representation and studying the differences between the two representations. Finnally, there will be some time to make a tutorial on how to use the VPE.
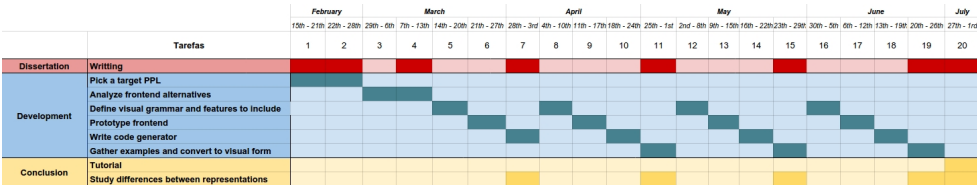
Figure 5.1: Work plan for the work of this dissertation

References

[AA92] K S R Anjaneyulu and John R Anderson. The Advantages of Data Flow Diagrams for Beginning Programming. 1992.

[ADDJ03] Christophe Andrieu, Nando De Freitas, Arnaud Doucet, and Michael I. Jordan. An introduction to MCMC for machine learning. *Machine Learning*, 50(1-2):5–43, 2003.

[Alp10] Ethem Alpaydin. *Introduction to Machine Learning*. MIT Press, second edition, 2010.

[Ama13] Xavier Amatriain. Beyond data: from user information to business value through personalized recommendations and consumer science. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*, CIKM '13, pages 2199–2200, New York, NY, USA, 2013. ACM.

[Amb87] Allen Ambler. Forms: Expanding the visualness of sheet languages. In *1987 Workshop on Visual Languages*, pages 105–117, 1987.

[BAF+15] Andrei Broder, Lada Adamic, Michael Franklin, Maarten de Rijke, Eric Xing, and Kai Yu. Big data: New paradigm or "sound and fury, signifying nothing"? In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*, WSDM '15, pages 5–6, New York, NY, USA, 2015. ACM.

[BBB+95] M.M. Burnett, M.J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. Scaling up visual programming languages, 1995.

[BD04] Marat Boshernitsan and M.S. Ms Downes. Visual programming languages: A survey. *Control*, (December), 2004.

[Ber] Henri Bergiu. Noflo. Available at http://noflojs.org/, accessed in 7th Feb 2016.

[Bri16] Encyclopædia Britannica. machine learning. Available at http://www.britannica.com/technology/machine-learning, accessed in 3rd Jan 2016, 2016.

[Bro86] Frederick Phillips Jr. Brooks. No silver bullet - essence and accidents of software engineering. *Proceedings of the IFIP Tenth World Computing Conference*, pages 1069–1076, 1986.

[Bur99] Margaret Burnett. Visual programming. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 32(1-3):275–283, 1999.

REFERENCES

[Car02]   Liberato Cardellini. An interview with Richard M. Felder. *Journal of Science Education*, 3(2):62– 65, 2002.

[CC86]   Olivier Clarisse and Shi-Kuo Chang. *Visual Languages*, chapter Vicon, pages 151–190. Springer US, Boston, MA, 1986.

[Cra04]   Ciprian Crainiceanu. A short introduction to winbugs. 2004.

[CSB82]   Jay JJ Christensen-Szalanski and Lee Roy Beach. Experience and the base-rate fallacy. *Organizational Behavior and Human Performance*, 29(2):270–278, 1982.

[CTB86]   Nancy Cunniff, Robert P Taylor, and John B Black. Does programming language affect the type of conceptual bugs in beginners' programs? a comparison of fpl and pascal. *ACM SIGCHI Bulletin*, 17(4):175–182, 1986.

[CWHH05]   Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries, and Steven M. Hadfield. Raptor. *Proceedings of the 36th SIGCSE technical symposium on Computer science education - SIGCSE '05*, (January):176, 2005.

[D$^+$89]   Edsger W Dijkstra et al. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, 1989.

[Das02]   M. Dastani. The role of visual perception in data visualization. *Journal of Visual Languages and Computing*, 13(6):601–622, 2002.

[DL]   David Duvenaud and James Lloyd. Introduction to probabilistic programming.

[DL13]   David Duvenaud and James Lloyd. Talk on 'introduction to probabilistic programming'. 2013.

[Dow12]   Allen B. Downey. *Think Bayes*. Green Tea Press, 2012.

[fE15]   Google for Education. Blockly is a library for building visual programming editors. Available at https://developers.google.com/blockly/, accessed in 2nd Feb 2016, 2015.

[Fie06]   Stephen E. Fienberg. When did Bayesian inference become "Bayesian"? *Bayesian Analysis*, (1):1–41, 2006.

[FMR10]   Cameron E Freer, Vikash K Mansinghka, and Daniel M Roy. When are probabilistic programs probably computationally tractable? *NIPS 2010 Workshop on Monte Carlo Methods in Modern Applications*, pages 1–9, 2010.

[Fou16]   Eclipse Foundation. Windowbuilder - is a powerful and easy to use bi-directional java gui designer. Available at https://eclipse.org/windowbuilder/, accessed in 2nd Feb 2016, 2016.

[FR12]   Cameron E. Freer and Daniel M. Roy. Computable de Finetti measures. *Annals of Pure and Applied Logic*, 163(5):530–546, 2012.

[GEGN04]   Fischer G, Giaccardi E, Sutcliffe a G, and Mehandjiev N. Meta-Design : A Manifesto for End-User Development. *Communications of the ACM*, 47:1–7, 2004.

[Geo16]   Pétia Georgieva. Machine learning for big data processing. 2016.

REFERENCES

[Gor14]   Sriram K. Gordon, Andrew D. and Henzinger, Thomas A. and Nori, Aditya V. and Rajamani. Probabilistic Programming. *International Conference on Software Engineering (ICSE Future of Software Engineering)*, pages 267–, 2014.

[Gre95]   T.R.G. Green. Noddy's Guide to ... Visual Programming. *The British Computer Society ● Human-Computer Interaction Group*, 1995.

[Gri13]   Olivier Grisel. Keynote on 'trends in machine learning and the scipy community'. Available at https://youtu.be/S6IbD86Dbvc, accessed in 3rd Jan 2016, 2013.

[GS14]   Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2016-1-14.

[GSD]   Andreas Geyer-Schulz and Chuck Dyer. Lecture notes on stanford's 'cs63: Knowledge representation and reasoning - chapter 10.1-10.2, 10.6'.

[Hal84]   Daniel Conrad Halbert. *Programming by Example*. Phd, University of California, Berkeley, 1984.

[HBB$^+$12]   Shawn Hershey, Jeffrey Bernstein, Bill Bradley, Andrew Schweitzer, Noah Stein, Theophane Weber, and Benjamin Vigoda. Accelerating inference: towards a full language, compiler and hardware stack. *CoRR*, abs/1212.2991, 2012.

[HR08]   Mark Hall and Peter Reutemann. Weka knowledgeflow tutorial for version 3-5-8. Available at http://software.ucv.ro/ eganea/AIR/KnowledgeFlowTutorial-3-5-8.pdf, accessed in 7th Feb 2016, 2008.

[Jag13]   Suresh Jagannathan. Probabilistic programming for advancing machine learning (ppaml). Available at http://www.darpa.mil/program/probabilistic-programming-for-advancing-machine-Learning, accessed in 3rd Jan 2016, 2013.

[Jam14]   Lothar Jamal, Rahman and Wenzel. The Applicability of Visual Programming to Large Real-World Applications. *Igarss 2014*, (1):1–5, 2014.

[JW96]   Michael I Jordan and Yair Weiss. Probabilistic inference in graphical models. *Lauritzen, S. L*, 16(510):140–152, 1996.

[KP05]   Caitlin Kelleher and Randy Pausch. Lowering the Barriers to Programming : A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Computing Surveys (CSUR)*, 37(2):83–137, 2005.

[KT15]   Tejas D Kulkarni and Joshua B Tenenbaum. Picture : A Probabilistic Programming Language for Scene Perception. *Cvpr*, 2015.

[LO87a]   Clayton Lewis and Garay Olson. Can principles of cognition lower the barriers to programming? *Empiral Studies of Programmers:Second Workshop*, 17:248–263, 1987.

[LO87b]   G Lewis and G Olson. Can principles of cognition lower rhw barriers to programming? *Empirical Studies of Programmers: Second Workshop*, 2(15):248–263, 1987.

REFERENCES

[LTBS00]   David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS –
             A Bayesian modelling framework: Concepts, structure, and extensibility. *Statistics
             and Computing*, 10:325–337, 2000.

[Min99]   Thomas P Minka. Expectation Propagation for Approximate Bayesian Inference F
             d. *Statistics*, 17(2):362–369, 1999.

[MWGK12]   Tom Minka, John Winn, John Guiver, and David Knowles. Infer .net 2.5. *Microsoft
             Research Cambridge*, 2012.

[Mye86]   B. a. Myers. Visual programming, programming by example, and program visual-
             ization: a taxonomy. *ACM SIGCHI Bulletin*, 17(4):59–66, 1986.

[Mye90]   Brad A. Myers. Taxonomies of visual programming and program visualization.
             Journal of Visual Languages & Computing, 1(1):97–123, 1990.

[PB99]   Marian Petre and Alan F Blackwell. Mental imagery in program design and visual
             programming. *Int. J. Hum.-Comput. Stud.*, 51(1):7–30, 1999.

[Pia15]   Gregory Piatetsky.   16th annual kdnuggets software poll.   Available
             at   http://www.kdnuggets.com/2015/05/poll-r-rapidminer-python-big-data-
             spark.html, accessed in 4th Jan 2016, 2015.

[Pre03]   Probabilistic Programming. *Handbooks in Operations Research and Management
             Science*, 10(C):267–351, 2003.

[Rap]   RapidMiner.   Rapidminer   studio   manual.   Available   at
             http://docs.rapidminer.com/downloads/RapidMiner-v6-user-manual.pdf, accessed
             in 7th Feb 2016.

[Res]   Microsoft   Research.   http://research.microsoft.com/en-
             us/um/cambridge/projects/infernet/docs/Working   http://research.
             microsoft.com/en-us/um/cambridge/projects/infernet/docs/
             Working%20with%20different%20inference%20algorithms.aspx.
             Accessed: 2016-01-14.

[Res10]   Microsoft Research. Microsoft bayesian network editor. Available at
             http://research.microsoft.com/en-us/um/redmond/groups/adapt/msbnx/, accessed
             in 10th Jan 2016, 2010.

[RMMH+09]   Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Eve-
             lyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian
             Silverman, et al. Scratch: programming for all. *Communications of the ACM*,
             52(11):60–67, 2009.

[RT15]   Michael Reposa and Glenn Takata.   Figaro.   Available at
             https://github.com/p2t2/figaro, accessed in 14th Jan 2016, 2015.

[SAC+10]   Yuan Shen, Cedric Archambeau, Dan Cornford, Manfred Opper, John Shawe-
             taylor, and Remi Barillec. A comparison of variational and Markov Chain Monte
             Carlo methods for inference in partially observed stochastic dynamic systems.
             Journal of Signal Processing Systems, 61(1):51–59, 2010.

REFERENCES

[Sch08]   Rob   Schapire.   Lecture   notes   in   'cos   511:
            Theoretical   machine   learning'.   Available   at
            http://www.cs.princeton.edu/courses/archive/spr08/cos511/scribe$_n$otes/0204.pdf, accessedin3rdJa

[SG01]   P Sedlmeier and G Gigerenzer. Teaching Bayesian reasoning in less than two hours.
            Journal of experimental psychology. General, 130(3):380–400, 2001.

[SK88]   Ted Selker and Larry Koved. Elements of visual language. In *Visual Languages,
            1988., IEEE Workshop on*, pages 38–44. IEEE, 1988.

[Smi77]   David Canfield Smith. *Pygmalion: a computer program to model and stimulate
            creative thought*, volume 40. Birkhauser, 1977.

[Sof15]   Northwoods Software. Blockly is a library for building visual programming editors.
            Available at http://gojs.net, accessed in 2nd Feb 2016, 2015.

[Sta15]   Conn Stamford. Gartner's 2015 hype cycle for emerging technologies identi-
            fies the computing innovations that organizations should monitor. Available at
            http://www.gartner.com/newsroom/id/3114217, accessed in 3rd Jan 2016, 2015.

[Wag12]   Kiri Wagstaff. Machine Learning that Matters. *Proceedings of the 29th Interna-
            tional Conference on Machine Learning*, pages 529–536, 2012.

[Wan02]   DeLiang Wang. Ohio state cis 730, lecture notes on 'probabilistic reasoning - belief
            networks'. 2002.

[WB97]   K N Whitley and Alan F Blackwell. Visual programming: the outlook from
            academia and industry. *Papers presented at the seventh workshop on Empirical
            studies of programmers*, pages 180–208, 1997.

[WBJ05]   John Winn, Cm Bishop, and T Jaakkola. Variational Message Passing. *Journal of
            Machine Learning Research*, 6:661–694, 2005.

[Wik]   Blender   Wiki.   Composite   nodes.   Available   at
            http://wiki.blender.org/index.php/Doc:2.4/Manual/Composite$_N$odes, accessedin7thFeb2016.

[ZP94]   Nevin L Zhang and D Poole. A simple approach to Bayesian network computations. *Proc. of the
            Tenth Canadian Conference on Artificial Intelligence*, pages 171–178, 1994.