Interactive Development Environment for Probabilistic Programming

Computer Science Tripos, Part II
Clare College
May 14, 2015
Proforma

Name and College: Maria I. Gorinova, Clare College
Project Title: Interactive Development Environment for Probabilistic Programming
Examination: Computer Science Tripos Part II, 2015
Word Count: 11902
Project Originator: Alan Blackwell and Advait Sarkar
Supervisor: Advait Sarkar

Original Aims of the Project
To implement an interactive development environment, which allows the programmer to use the probabilistic programming framework Infer.NET. The environment must support visualisations of the graphical models described by the source code, as well as of the probability density functions of random variables that have been declared. In addition, the environment must be live, meaning it provides run-time feedback about the execution of the source code. Update must be edit-triggered, meaning that re-compilation occurs when the code is changed, without the compiler having to be explicitly invoked by the user.

Work Completed
The work outlined in the original project proposal was successfully implemented, resulting in an IDE which meets all success criteria. In addition, several small features such as text highlighting and code navigation were added to the user interface. A user study was conducted to evaluate the usability of the software. The environment proves to significantly improve users’ experience by reducing the probabilistic programming tasks completion time and the level of confusion within a user. Moreover, the IDE improves the learning process of students encountering probabilistic programming concepts for the first time.

Special Difficulties
None.

\footnote{This word count was computed by TeXcount: \url{http://app.uio.no/ifi/texcount/}}
Declaration

I Maria Ivanova Gorinova of Clare College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed:
Date: May 14, 2015
## Contents

1 Introduction............................................................................. 1
   1.1 Motivation......................................................................... 2
   1.2 Previous Work.................................................................... 3
   1.3 Overview........................................................................... 4

2 Preparation ........................................................................... 5
   2.1 Background Theory.......................................................... 5
       2.1.1 Graphical Models......................................................... 5
       2.1.2 Probabilistic Inference.................................................. 8
       2.1.3 Probabilistic Programming.......................................... 10
       2.1.4 Performance Bottlenecks............................................. 13
   2.2 Further Planning............................................................... 13
       2.2.1 Requirements Analysis............................................... 13
       2.2.2 Development Strategy............................................... 15
       2.2.3 Implementation Approach.......................................... 16
       2.2.4 Languages and Tools.................................................. 18
   2.3 Summary........................................................................... 19

3 Implementation ..................................................................... 21
   3.1 High-level Architecture.................................................... 21
3.2 Checker Module ......................................................... 24
   3.2.1 Extracting the Declared Variables ....................... 24
   3.2.2 Limitations of the language ............................ 26
3.3 Injection Phase ...................................................... 27
3.4 Graph Module .......................................................... 29
   3.4.1 Factor Graphs .................................................. 30
   3.4.2 Model Graph Class ......................................... 31
   3.4.3 Converting a Factor Graph to a Bayesian Network .... 33
3.5 Providing Liveness ..................................................... 35
   3.5.1 Edit-triggered Re-compilation ........................... 35
   3.5.2 Concurrent Re-compilation ............................... 36
3.6 User Interface .......................................................... 38
   3.6.1 Graph Layout ................................................ 38
   3.6.2 Visualising Inferred Distributions ....................... 41
   3.6.3 Console ....................................................... 43
   3.6.4 User Experience Improvements .......................... 44
3.7 Summary ................................................................. 46

4 Evaluation ................................................................. 47
4.1 Experiment Design ..................................................... 48
   4.1.1 Considerations ............................................... 48
   4.1.2 Overall Design .............................................. 48
   4.1.3 Workbook Design .......................................... 50
   4.1.4 Experimental Procedure ................................. 51
4.2 Results ................................................................. 54
   4.2.1 Usability ...................................................... 54
# List of Figures

1.1 Why is visualisation important? ........................................ 3  
2.1 The *cloudy-rain-sprinkler-wet* example as a Bayesian network. 8  
2.2 Markov blanket of node A. *Diagram taken from Wikipedia.* .... 9  
2.3 The *cloudy-rain-sprinkler-wet* example .......................... 10  
2.4 Infer.NET’s Inference Engine ....................................... 11  
2.5 Simplified flowchart describing the functionality of the IDE. ... 17  
2.6 Distribution of requirements between project phases. .......... 17  
2.7 Mock up of the user interface. .................................... 18  
3.1 Screenshot showing the IDE visualising the *Cloudy-Rain-Sprinkler-Wet* example written in Infer.NET. ..................... 22  
3.2 Model View ViewModel pattern .................................... 23  
3.3 High-level overview of the Visualisation Update Subsystem of the IDE. ................................................................. 23  
3.4 Overview of Checker Module’s functionality ...................... 24  
3.5 A closer view of the Checker Module ............................ 26  
3.6 Overview of Injection Phase’s functionality ...................... 28  
3.7 Overview of Graph Module’s functionality ........................ 29  
3.8 Example of a Factor Graph. ........................................ 31  
3.9 UML diagram of the Model. ........................................ 32
Acknowledgements

First and foremost, I would like to thank my supervisor, Advait Sarkar, for his guidance and support throughout the project. This work could not have been completed without his ceaseless encouragement and invaluable advice. I would also like to thank him for conceiving the idea of such an interesting project together with Alan Blackwell.

I owe very special thanks to Don Syme for giving me plenty of useful comments and suggestions, helping me with subtleties of the F# programming language and for supporting the project throughout.

Next, thank you to Alex Spengler for taking the time to answer my questions, keeping me updated on changes in the Infer.NET binaries and providing me with materials to help my implementation work.

Finally, thank you to everyone who devoted time to participate in my experiment.
Chapter 1

Introduction

“Graphical elegance is often found in simplicity of design and complexity of data.”
— Edward Tufte

This dissertation describes the process of planning, creation and evaluation of an interactive development environment for the probabilistic programming framework Infer.NET. It shows how the IDE was built to visualise underlying graphical models and random variables’ distributions and, moreover, how that was done so the environment is live, i.e. with realtime interactivity.

An experiment with human participants was designed and conducted to study the way the IDE affects users’ experience with Infer.NET. Results show that the environment improves usability of the framework while completing probabilistic programming tasks, by increasing speed and accuracy compared to a plain text editor. Moreover, the experiment showed that the environment is beneficial for the process of learning about probabilistic programming concepts and set some interesting questions about self-confidence.

The rest of this chapter explains the motivation behind building tools that provide live visualisations while programming and while using probabilistic languages in particular (§ 1.1) as well as previous work on visualising graphical models (§ 1.2). The chapter finishes with a short overview of the rest of the dissertation.
1.1 Motivation

Today, machine learning methods have large application in statistics, robotics, biology, neuroscience, finance, artificial intelligence, cognitive science. Extensive work in these areas was possible partially due to the introduction of probabilistic graphical models. However, in many cases these are used as high-level descriptions and solved using special purpose inference methods, rather than being precisely specified to be suitable for automatic inference.

This has given motivation for designing and creating probabilistic programming languages [17, 16, 24, 28], which unify general purpose programming with probabilistic modelling to introduce a more abstract and powerful way of applying these models to real-world problems.

Real-world applications can become extremely complex and difficult to work with. One reason for this difficulty is that there exist a gap between the program text and its behaviour. Software developers need to constantly execute parts of the code in their mind to keep track of what their program is doing [22, 27]. When using a high-level programming language, such as a probabilistic programming language, the programmer needs to understand the behaviour and relationships between abstract objects. This turns the “imaginary execution” in an even more difficult task.

In his essay “Learnable Programming” [37], Bret Victor discusses design principles of programming languages and development environments, which could help solving this problem, by narrowing the gap between the program text and execution and allowing the programmer to “see”. He describes the goals of a programming system as being:

- to support and encourage powerful ways of thinking
- to allow programmers to see and understand the execution of their programs

Recent products such as the Larch environment [14, 15], the Swift language [9] and the Glitch programming model [31] make an attempt of achieving those goals. They seek a next-level abstraction of what programming is, by introducing suitable visualisations that update continuously while editing the code, to provide most recent feedback of the consequences of those edits. This, so called live programming, improves on existing programming practise by bridging the gap between code and program execution, as results of execution are embedded in the environment itself.
1.2. PREVIOUS WORK

This alternative approach to the “edit-compile-debug” style of programming, when applied to probabilistic programming could potentially lead to more powerful ways of thinking about underlying graphical models, so one can be more productive and produce fewer errors. Moreover, newcomers should have access to tools that would encourage and help their learning process. Figure 1.1 shows how important the ability to “see” in the context of probabilistic programming is.

Inspired by previous work on smart tools that help programmers learn and understand how their program would behave better, this dissertation proposes building an IDE which allows the programmer to work with the probabilistic programming framework Infer.NET in an interactive way. The environment was built to support live visualization of the probabilistic graphical models being created. The software is novel in that it both graphically represents the program data and provides feedback about the program execution in a live way. There exist tools which support visualisation or liveness, however neither combines the two in a single product.

1.2 Previous Work

Previous work in the area mainly focuses on individual parts of what has been done in this work and neither has assessed the effect of visualising Bayesian networks on learning about probabilistic programming.

ViSMod [38] is a tool that showed to help students understand through visualisations their student’s model, which was described by a Bayesian network.
Moreover, the tool made students curious, it motivated them to explore further the model and learn more about it.

Bayesia [2] and Netica [6] are both tools built for personal and business use, that allow users to create Bayesian networks interactively. However, these do not provide any transparency of the underlying code and have restricted functionality due to their commercial focus.

The closest implementations of this project are probably VIBES [11] and WinBUGS [23] which both allow the user to describe a probabilistic model and perform inference on it. The model in both cases is described by either XML script/BUGS language or pictorially, via a graphical interface. Neither studies how the graphical interface relates to the usability of the product nor incorporates it in a way so that visual feedback is provided in a live way.

A close implementation on the “live side” is Tabular [18] – a probabilistic language which allows programs to be written as annotated relational schemas and brings the power of probabilistic modelling to spreadsheets. Tabular was implemented using Infer.NET and is available as an Excel addin, however its interface is purely spreadsheet based and does not provide visualisations.

1.3 Overview

This chapter described the motivation behind building a live environment for probabilistic programming. The rest of the dissertation presents the process of planning, successfully building and evaluating such an IDE for Infer.NET.

The next chapter explains what preparatory work was undertaken before the development work started. This includes studying the background theory needed, making some important decisions on what languages and libraries to use and what development strategy to commit to.

Chapter 3 presents the implementation of the IDE, which provides a live, edit-triggered visualisation of the Bayesian network described by the source code and of the probability density functions of any random variables declared.

In order to study the effects on learning and Infer.NET’s usability of the tool, a user study was designed and conducted. The process and the results are described in Chapter 4.

The dissertation concludes with some summarised thoughts about the effects of the visual environment on learning and usability in Chapter 5.
Chapter 2
Preparation

“First learn computer science and all the theory. Next develop a programming style. Then forget all that and just hack.”
— George Carrette

This chapter explains what preparatory work was done before starting the implementation of the IDE. It begins with a brief explanation of the theory needed § 2.1 which is then followed by analysis of the requirements, reasoning behind the tools and languages that have been chosen and some further planning § 2.2.

2.1 Background Theory

This section describes the background theory on probabilistic programming that needed to be learnt and understood before the implementation work could start. Firstly, §§ 2.1.1 introduces the concept of graphical models and Bayesian Networks in particular. Then §§ 2.1.2 briefly describes what probabilistic inference is. The section finishes with an introduction to probabilistic programming, some characteristics of Infer.NET and how it works §§ 2.1.3.

2.1.1 Graphical Models

Probabilistic graphical models are graphs, whose vertices represent random variables, and whose edges represent dependencies between those random variables.
They are used to represent, in a compact way, joint probability distributions. In his book *Pattern Recognition and Machine Learning* [10], Bishop mentions a few very useful properties of graphical models:

- They provide a simple visualisation of probabilistic models.
- Properties of the model, such as conditional independence properties, can be seen from the graph.
- They allow complex computations to be expressed as graphical manipulations.

This dissertation focuses on a particular subset of the family of graphical models – *Bayesian networks*.

**Bayesian networks**

Bayesian networks are directed acyclic graphical models. The edges in a Bayesian network are directed, so they show *causal* relationships between variables. That is if given the random variable A we know the distribution of B, there is an edge from A to B.

To motivate the use of Bayesian networks consider the following problem:

We would like to investigate the probability of the grass in our lawn being wet. We believe that the following is true:

- There is 0.5 chance to be cloudy.
- It is raining with probability 0.8 if it is cloudy, 0.2 if it is not.
- The sprinkler is on with probability 0.1 if it is cloudy, 0.5 if it is not.
- The grass could get wet with probability:
  - if it is raining: 0.99 if the sprinkler is on, 0.9 if it is off.
  - if it is not raining: 0.9 if the sprinkler is on, 0.0 if it is off.

To solve the problem, the probabilities of being cloudy, raining, having the sprinkler on and the grass being wet are represented through random variables –
2.1. BACKGROUND THEORY

C, R, S, W respectively. In this case, it is believed that each of the above is a Bernoulli random variable, taking the values True or False. The dependencies between them could be described through conditional probability tables (CPTs) as follows:

| C       | P(R = true|C) | P(R = false|C) |
|---------|-------|------------|
| true    | 0.8   | 0.2        |
| false   | 0.1   | 0.9        |

| C       | P(S = true|C) | P(S = false|C) |
|---------|-------|------------|
| true    | 0.1   | 0.9        |
| false   | 0.5   | 0.5        |

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>0.99</td>
<td>0.01</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>0.9</td>
<td>0.1</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>0.9</td>
<td>0.1</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>0.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

The Bayesian network describing those dependencies is shown in Figure 2.1.

To be able to answer queries about this model, the joint probability distribution P(C, R, S, W) has to be computed. This could be expressed, using the chain rule, as:

\[
P(C, R, S, W) = P(C) \times P(R|C) \times P(S|R, C) \times P(W|S, R, C)
\]

Conditional independence

Two random variables, X and Y, are said to be conditionally independent given a third random variable, Z, if \( P(X, Y|Z) = P(X|Z) \times P(Y|Z) \). Moreover, this means that \( P(X|Y, Z) = P(X|Z) \). Conditional independence of X and Y given Z is commonly written as \( X \perp Y|Z \).
Figure 2.2
Bayesian networks have the property that any node is conditionally independent of any other node given its Markov blanket. The Markov blanket of a node is the set of all its parents, children, and any other parents of its children, as shown on Figure 2.2.

Therefore, in the cloudy-rain-sprinkler-wet example, we have:

- \( R \perp S | Z \), thus \( \mathbb{P}(S|R, C) = \mathbb{P}(S|C) \).
- \( W \perp C|R, S \), thus \( \mathbb{P}(W|S, R, C) = \mathbb{P}(W|S, R) \).

The above joint probability could then be simplified to become:

\[
\mathbb{P}(C, R, S, W) = \mathbb{P}(C) \times \mathbb{P}(R|C) \times \mathbb{P}(S|C) \times \mathbb{P}(W|S, R)
\]

This shows one of the benefits graphical models provide – having knowledge about conditional dependencies can help with simplification of the terms of joint probabilities and thus reduce the complexity of the inference tasks performed.

2.1.2 Probabilistic Inference

The most common task that Bayesian networks are used for is probabilistic inference. This is the process of updating our knowledge about the distributions of random variables in the network, given some evidence.

The prior probability of being cloudy is \( \mathbb{P}(C = \text{true}) = 0.5 \). However, the posterior probability of being cloudy, given values of other random variables in the model – the evidence \( E \) – is written as \( \mathbb{P}(C = \text{true}|E) \) and it is different. For example,
suppose it was observed that the grass in the lawn is wet. The probability of being cloudy, as shown by Figure 2.3 is no longer 0.5. It can be inferred as follows:

\[
P(C = \text{true}|W = \text{true}) = \frac{\sum_{S,R} P(C = \text{true}|R, S, W = \text{true})}{\sum_{S,R} P(R, S, W = \text{true})} \approx 0.65
\]

This is a way of performing exact inference. However, in practice graphical models become much more complex than the cloudy-rain-sprinkler-wet example, with continuous and multivariate random variables, thus exact inference becomes very slow. In consequence, research focuses on developing algorithms for approximate inference.
CHAPTER 2. PREPARATION

Figure 2.3: The cloudy-rain-sprinkler-wet example before and after observing that $W$ is true.

2.1.3 Probabilistic Programming

Probabilistic programming languages are high-level languages which unify general purpose programming with probabilistic modelling to introduce a more abstract and powerful way of applying these to real-world problems. They attempt to provide easy-to-use tools for describing probabilistic models by treating random variables as primitive objects. Inference is performed automatically and its detailed are abstracted away.

Infer.NET

Infer.NET is one such probabilistic programming extension for .NET languages. It provides a rich modelling language, allowing the programmer to construct models through arithmetic operations, constraints, boolean operators, linear algebra. It could handle two types of graphical models – Bayesian networks and Markov random fields, that could contain various discrete and continuous variables, both univariate and multivariate.

The framework currently supports three inference algorithms: Expectation Propagation, Variational Message Passing and Gibbs sampling. In this project I chose to use the default algorithm of Infer.NET to perform background inference when visualising the user’s code – Expectation Propagation \[25\]. The reason I chose it out of the three is that in most cases, the algorithm is more efficient than Gibbs Sampling and produces fewer errors in approximating distributions compared to Variational Message Passing.
Operation

Working with Infer.NET involves three main actions:

- **Modelling**: a model is specified by describing what random variables are involved and the dependencies between them. It also sets their priors.

- **Observing**: evidence could be observed to allow existing knowledge to be taken into account.

- **Inferring**: the posterior marginal distribution of variables could be inferred.

Infer.NET inference works by firstly compiling the specified model and set of inference queries into a source code describing the algorithm that is needed in order to perform inference on that exact model. Observed values are then “fed in” to the algorithm to re-calculate the marginal distributions of variables. A diagram illustrating how Infer.NET’s *Inference Engine* works is shown in Figure 2.4.

![Figure 2.4: Infer.NET’s Inference Engine. From http://research.microsoft.com/infernet.](http://research.microsoft.com/infernet)

The Cloudy-Rain-Sprinkler-Wet example in Infer.NET

To define $P(C = true) = 0.5$, a new random variable Cloudy is created, similarly to an ordinary program variable:

```
let Cloudy = Variable.Bernoulli(0.5)
```

$R$, $S$ and $W$ also take values in \{true, false\} and are defined through conditional probability tables, so Rain, Sprinkler and Wet are defined:
let Rain = Variable.New<bool>()
let Sprinkler = Variable.New<bool>()
let Wet = Variable.New<bool>()

Those are then initialised by describing the respective CPTs. For example, the CPT for Rain is given with the following code:

\[
\begin{align*}
\text{begin} & \quad \text{use ifC = Variable.If(Cloudy)} \\
& \quad \text{Rain.SetTo(Variable.Bernoulli(0.8))} \\
\text{end} \\
\text{begin} & \quad \text{use ifNotC = Variable.IfNot(Cloudy)} \\
& \quad \text{Rain.SetTo(Variable.Bernoulli(0.1))} \\
\text{end}
\end{align*}
\]

After the probabilistic model is fully described, an Inference Engine is defined and can be used to perform inference. The call of Infer() below triggers the compilation of the model to a high performance code. The result of inferring the marginal distribution of Wet is the output to the console. In the case when there are no observed variables, this results in 0.5657.

let engine = new InferenceEngine()
printf "%A" (engine.Infer(Wet))

Following the example from §§ 2.1.2, Wet has to be observed to have the value true:

Wet.ObservedValue <- true

This changes the marginal distribution of Cloudy and could be seen by inferring the random variable as below. However, this time the call of Infer() does not cause recompilation of the model to an inference algorithm, as the model has not changed. The value of 0.6533 is sent to standard output.

printf "%A" (engine.Infer(Cloudy))

Full code of the example is given in Appendix A.
2.1.4 Performance Bottlenecks

Infer.NET’s compiler, even though is built to produce highly efficient inference code, is not itself particularly efficient. As the IDE described in this project invokes this compiler on every re-compilation, its performance is bound by the time that Infer.NET takes to compile existing models.

2.2 Further Planning

The aim of this section is to describe the project planning process. Firstly, § 2.2.1 presents the requirement analysis of the project. Then § 2.2.2 argues why the chosen, incremental development strategy is reasonable and § 2.2.3 describes the implementation approach. Finally, § 2.2.4 explains and justifies important choices taken regarding the languages and tools that have been used.

2.2.1 Requirements Analysis

This subsection extends the aims suggested in the project proposal and discusses necessary considerations regarding the evaluation and preparation for eventual extensions.

Detailed Project Aims

The original project proposal suggest building an IDE for the probabilistic programming framework Infer.NET that provides a live, edit-triggered visualisation of the Bayesian networks described by the source code and of the probability density functions of random variables that have been declared.

Liveness of a programming environment is understood as the level of responsiveness of the system – the amount and level of complexity of the feedback the system provides regarding the execution of a program.

Tanimoto [35] describes 4 possible level of liveness of visual programming systems, which were recently extended with another two levels in [36]. These levels were used as a guidelines while planning the process of implementation the software, as will shortly be described in § 2.2.3. Moreover, these also motivated extensions, where the timespan of this project didn’t allow considering them as a part of the main requirements.
This project focuses on building an IDE that provides level 3 liveness – the environment responds to an edit of the program with immediate feedback, i.e. automatically triggers re-execution to update the program representation. The aims of the project are summarised as building an environment that allows the user to:

- create, edit, compile and execute Infer.NET code.
- see a responsive representation of the graphical models that are being described.
- see a responsive visualisation of the data – density functions of random variables that have been declared.

**Preparation for Extensions**

Extensions were considered as follows:

- allowing the user to see when inference happens. As discussed in [Chapter 4](#), many users did not find the process of inference intuitive. This process could be perceived as more of a side effect than an explicit “reassignment” as with conventional programming. Making the time visible would extend the level 3 liveness to allow the user see not only the state after execution, but the execution itself, minimising the level of confusion.

- connect regions of code with elements of the visualisation to allow the user directly manipulate their program through its graphical representation.

The latter feature was carefully thought through and further planning was made to ensure that the current software could be easily extended to support it.

**Requirements**

Referring to the taxonomies of visual programming and program visualisation given in [26], the IDE in this dissertation could be classified as a compiled, data visualisation, static system, providing program visualisation. A summary of the requirements and their associated priority and difficulty is described in [Table 2.1](#).
2.2. FURTHER PLANNING

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Priority</th>
<th>Difficulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding Bayesian Networks approach</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Learning about the functionality of Infer.NET</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Implementing a front-end enabling the user to write, compile and execute F# code</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Implementing a module that extracts the graphical model from the input code</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Implementing a module that invokes Infer.NET’s compiler to infer variables’ distributions</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Visualising the graphical model obtained</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Visualising the variables’ distributions obtained</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Making re-compilation edit-triggered</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>Making the IDE responsive and live</td>
<td>Medium</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table 2.1: Requirements analysis.

Evaluation

Evaluation, taking a major part of this project (about half of the time was spent in development work, half of the time – designing the user study), was considered and thought about since the very beginning.

There were several options for evaluation. The software could either be tested on people who are already proficient with Infer.NET or on learners. The first evaluation strategy could have only provided a way to study the usability of the tool. The second option, however, was seen as a way to answer another interesting question: whether the presence of the IDE beneficial for the process of learning about probabilistic programming.

The project evaluation was thus decided to take into consideration two aspects - learning and usability in a single experimental setting. A full description of the final design of the experiment and the results are presented in Chapter 4.

2.2.2 Development Strategy

The implementation process of the IDE has been following the Incremental development model. Moreover, some Agile principles were applied mainly towards
the end of the implementation.

There are several reasons why the Incremental model was chosen as a development strategy:

- Major requirements were well understood from the beginning, so it was possible to apply the model by planning straight away what the two major prototypes will be.

- However the model still gives enough flexibility for the detail and allows quick changes and details evolving with time, which is suitable for the user-driven nature of the project.

- There are high risk features and goals – the project requires some working prototype to be implemented, so it could be evaluated and described in a dissertation.

- The nature of the project requires rapid development and having working prototypes early, so they could be discussed with a supervisor, overseers, etc.

Agile stages were included to give freedom in later stages of the project, when rapid changes in user interface were needed.

A description of how this strategy was implemented is given in the next subsection.

2.2.3 Implementation Approach

This section describes in what way the work was split into increments.

Guided by Tanimoto’s levels of liveness mentioned earlier, the following prototypes were specified as checkpoints between iterations:

- Prototype 1 is building up the IDE to level 2 liveness – *program is entirely specified by the representation*. This would include implementing and testing the system described in Figure 2.5.

- Prototype 2 is building up the IDE to level 3 liveness – *responsive* – any changes of the source code triggers a re-execution and an update of the representation.
2.2. FURTHER PLANNING

Compile and execute source code

Find random variables described in the source code

Infer random variables’ distributions

Obtain Infer.NET’s graphical model

Compute model graph

Visualise variables’ distributions

Visualise graphical model

Figure 2.5: Simplified flowchart describing the functionality of the IDE.

Requirements 1
Implement a Visualisation Update Subsystem:
- compile F# code
- extract and visualise BNs described
- extract and visualise distributions

Design and Development 1

Prototype 1
level 2 liveness

Requirements 2
- ensure re-compilation is edit-triggered
- ensure the system is responsive

Design and Development 2

Prototype 2
level 3 liveness

Figure 2.6: Distribution of requirements between project phases.
CHAPTER 2. PREPARATION

Figure 2.7: Initial mock up of the user interface. Three main panels were planned: (a) a text editor panel, (b) a graphical model visualisation panel and (c) a distribution visualisation panel.

The second prototype then satisfies the success criteria of the project. Several additional Agile increments were implemented to make small, mainly stylistic updates, where they were seen as beneficial.

The way requirements were split between project phases is described in Figure 2.6 and an initial mock-up of the user interface, in Figure 2.7.

2.2.4 Languages and Tools

Choice of Languages

- User Input Language – F#. Infer.NET is a .NET framework, thus it could be used from any .NET language. There are several reasons why F# was chosen as the input language:
  - F# Compiler Services: these allow a straightforward way of communicating with F#’s compiler and interpreter to parse, typecheck, interpret code.
2.3. SUMMARY

– Code injection: F#’s interpreter allows interpreting code line by line, thus receive information about random variables one by one, as will be described in § 3.3. Thus, performance could be gained by allowing more code to be executed in parallel.

– Lightweight syntax: this provides a way of having simpler, concise code, which would help the user write and understand Infer.NET more quickly.

• Development Languages – C# and F# – F# provides easy use of the Compiler Services binaries. Pattern matching supported by the language made it easier to implement the module that visualises variables’ distributions. On the other hand, C# is suitable for building graphical user interface with the aid of Windows Presentation Foundation (WPF).

Choice of Libraries

• Probabilistic programming framework – Infer.NET [21]

• Visualisation of the graphical model – Graph# [5]

• Visualisation of the random distributions – F#Charting [3]

• Text editor inside the IDE – AvalonEdit [1]

Development Environment

• IDE – Visual Studio Ultimate 2013

• Version control – Git

• Backup strategy – OneDrive cloud services and GitHub

2.3 Summary

This chapter provides an account of the preparatory work conducted before the implementation began. The actual implementation work is described in the next chapter.
Chapter 3

Implementation

“A picture is worth a thousand words. An interface is worth a thousand pictures.”
— Ben Shneiderman

This chapter describes the way the development environment is implemented. An outline of the system and extended reading guide are given in § 3.1. The chapter continues with a description of the Visualisation Update Subsystem in § 3.2 § 3.3 and § 3.4 followed by an overview of the way the system was made live in § 3.5 and more details on the user interface in § 3.6.

3.1 High-level Architecture

The software consists of a Visualisation Update Subsystem (VUS), which is executed every time a recompilation is triggered. This execution is carefully ran in a different thread on every edit, to make the environment live.

All components were successfully implemented, resulting in an IDE for Infer.NET that provides live visualisations of the random variables and graphical models described by the source code. Figure 3.1 shows a screenshot of the working software.
The Visualisation Update Subsystem

The VUS obtains necessary information regarding the graphical models and variables’ distributions described in the code and visualises that information appropriately. The implementation follows the best practises of the Model View ViewModel (MVVM) architectural pattern [33, 19]. The pattern, as also described in Figure 3.2, splits visualising of components into working on three parts:

- **View**: an XAML markup that describes the way the structure is visualised. This is the presentation of the data.

- **ViewModel**: an object containing bindings to the View, a bridge between the View and the Model.

- **Model**: an object that stores the logistics data. In the case of this dissertation, the Model is the object that the VUS updates.

In brief, the Visualisation Update Subsystem works as follows:

1. Parse and type-check the code and then build a list containing information about Infer.NET random variables declared (Checking Phase).
2. Interpreted the code and start “injecting” inference queries for interpretation, to obtain the posterior marginal distribution of each random variable in the list (Injection Phase).

3. In additions, while inferring, objects describing the graphical models in the program are being serialised by Infer.NET. These are handled by the Graph Module, which updates the Model object (part of the MVVM pattern as described above), being then visualised on the screen. The Model is built up by adding the information about all graphical models that Infer.NET serialises within a single run of the VUS.

Figure 3.3 shows an overview of the VUS, while the next three sections describe its components in more detail.
3.2 Checker Module

The F# Checker Module, presented in Figure 3.4 could be roughly described as follows:

1. Input an F# source file.
2. Parse and type-check the source code to produce a typed abstract syntax tree (TAST).
3. Walk over the typed tree to find declarations of Infer.NET random variables.
4. Output list of random variables names.

![Checker Module Diagram](image)

Figure 3.4: Overview of Checker Module’s functionality

To implement the module, I made use of the F# Compiler Services package [4], which is a component derived from the F# compiler that provides tools, allowing the programmer to implement a hosted compiler or interpreter for the F# language, to get information about auto-completion, tooltips and parameters, to obtain and work with the abstract syntax tree of parsed code.

3.2.1 Extracting the Declared Variables

In order to infer variables’ distributions, their names have to be extracted from the source file. To do so several strategies were possible:

- Walk the abstract syntax tree – provides a lot of information but is messy.
3.2. **CHECKER MODULE**

- Use regular expressions to walk through source code – highly inaccurate but quick.
- Walk the typed abstract syntax tree – provides enough information, quick enough, not messy.

I chose the last option, as it provides all the information (namely the types) needed to determine if an expression is a declaration of a variable and determine if this variable is an Infer.NET random variable. It also gives opportunity to write clean and concise code.

Implementation involved searching through the types of top level declarations and finding those of variables of type `MicrosoftResearch.Infer.Models.Variable` (referred to as `Variable` for simplicity below).

F#’s declarations include information about the name of the variable declared, its type, the exact line of the original source code at which this happens. This information is used for to things:

- record the name and location of the declaration by creating a new variable object in the output list, to allow this information to be passed in the injection phase and used to infer variables, implement UI-to-code navigation, etc.

- change the source file, so that Infer.NET variables are “named” as explained in the next paragraph.

**“Naming” the Random Variables**

Nodes in the graph objects Infer.NET serialises are initially randomly named, instead of having the name of the program variable as a label. The framework has a special method that a programmer could use to name her variables as she wishes – `Variable.Named()`. As will be described in the next section, having the names of the nodes being the same as the corresponding program variables is vital to the correct execution of the IDE. Furthermore, it removes unnecessary complications and minimises the need for syntax overheads, which proves to be especially useful for novices.
Figure 3.5: A closer view of the Checker Module having encountered a declaration of a random variable (X) while searching the TAST.

Following the type-checking from the beginning of the subsection, top-level declaration search is implemented as follows:

- Whenever a declaration of type `Variable` is encountered, the `name` of the element declared and the `line number` of the declaration are extracted and included in the output.

- A “naming” function is called to add the respective `.Named()` call at the specified line in the source file. This file is kept on the disk to be interpreted later and the “naming” changes stay hidden from the user.

This is also shown in Figure 3.5.

### 3.2.2 Limitations of the language

Due to the timespan of this project, the IDE could not be constructed to support F# code fully. Thus, some important decisions regarding limiting the language used in the IDE had to be made.

Firstly, the IDE does not support full-scale projects, with several source files connected and external binaries used. It allows the user to work with a stand-alone F# script file and the Infer.NET framework.
Moreover, the IDE would compile and execute any F# source file that matches this descriptions. However, there exist some limitations in what would be visualised. Infer.NET models declared within a function or a separate class would not be visualised, as the checker only walks on top-level declarations. Even that extending the checker to recursively walk expressions is possible, it was decided to not implement this, as it would introduce implementation complications, such that any inference of a variable must happen within the scope of the function, which declares that variable. Implementing this was considered out of the scope of this project.

Finally, because of the way Infer.NET variables are “named”, if the user decides to specify a name of a random variable by herself, that name would be overwritten, enforcing the variable to adopt its program name in the Bayesian network. However, this could be easily extended in later versions of the product to provide the user with the freedom to set their own names for the visualised variables.

### 3.3 Injection Phase

This section describes the steps performed after checking the code and extracting the variables. As shown in Figure 3.6 these are as follows:

1. **Code interpretation**: Interpret the (modified after checking) script using the F# interpreter.

2. **Code injection**: Communicate with the interpreter, by injecting a series of infer statements, to obtain the distribution of each random variable in the environment.

3. On need, call the **Graph Module** to obtain and update information about the overall structure of the graphical model.

To be able to communicate with the F# interpreter, an F# Compiler Services object must be created. This object represents an F# Interactive evaluation session and once created can be used to interpret single line command, evaluate single line expression or interpret an entire, stand-alone script file.

A single evaluation session object is created on start-up of the IDE and firstly used to load Infer.NET binaries.

After processing the code and extracting desired information using the F# Checker module, the modified source file is interpreted. Information about the
CHAPTER 3. IMPLEMENTATION

Figure 3.6: Overview of Injection Phase’s functionality.

graphical models in the user’s code is obtained by “injecting” single line commands for interpretation, starting with the creation of an inference engine by interpreting an expression such as `let ie = new InferenceEngine()`.

To obtain marginal distributions information for each variable, it is enough to go through every single variable name written in the random variable’s data list returned from the Checker Module and inject an infer statement to be interpreted by the F# interpreter. In other words, for each variable name \( X \) in \( RVs \), the marginal distribution of \( X \) is \( ie.Infer(X) \).

However, obtaining the graphical model (or models) described by the entire user’s code, requires more work. Recall that Infer.NET’s inference engine works by first of all compiling the specified model into source code describing the inference algorithm for that particular model. Observed values are then “fed in” to the algorithm to re-calculate the marginal distributions of variables. Compilation of a model to source code happens on the first infer statement of a variable that is part of that model. This also causes Infer.NET to serialise a factor graph (§§ 3.4.1), which describes the model being compiled. Deserialisation and processing of the factor graph are handled by the Graph Module, as explained in the next section.

Thus, execution of the Graph Module is only needed in the cases when inferring the current variable causes compilation of a model. Notice the following:

- Inferring a variable causes compilation of a model if and only if that variable is not present in the model(s) that the inference engine currently supports.

- As a result of execution of the Graph Module, all currently supported variables are in the `Model` object (the datastructure keeping information about
3.4. GRAPH MODULE

the graphical model to be visualised.

The Injection Phase described with simple pseudocode is:

1: for all $X \in RVs$ do
2: infer $X$’s distribution
3: if $X \in Model$ then
4: add distribution information to $X$’s node
5: else
6: deserialise factor graph and update Model
7: add distribution information to $X$’s node
8: end if
9: end for

3.4 Graph Module

This section presents the Graph Module, which is used to handle Infer.NET’s serialised objects and process them to update the data to be visualised. An overview of the structure of the module is given in Figure 3.7

![Figure 3.7: Overview of Graph Module’s functionality.](image)

The Graph module’s work could be then roughly described as:

1. Infer.NET serialises a factor graph which describes the graphical model on which inference is performed.
2. The Graph Module deserialises the file into a DirectedGraph object.
3. This factor graph is converted to a Bayesian network described by a `ModelGraph` object.

4. This object is then united with the rest of the `Model`, which is itself a `ModelGraph` object.

### 3.4.1 Factor Graphs

Factor graphs [20][21] are bipartite graphs that provide a graphical representation for directed or undirected graphical models. As the name suggests, they represent models by emphasising the fact that a joint probability function of several variables could be factored into a product of joint probability functions (factors) of subsets of those variables.

Factor graphs consist of two types of nodes – *variable nodes* (represented by circles) and *factor nodes* (represented by squares). In the example below, there are also *constant nodes* (no formatting), as a special case of variable nodes. An edge between a variable node $n$ and a factor node $f$ exists when the function $f$ depends on $n$.

For example, consider the *Learning a Gaussian* problem:

> Given some observed data, which is supposed to be drawn from a Gaussian distribution, estimate the parameters (mean and variance) of that distribution.

To solve this, both the data and the parameters of its distribution are represented as random variables as follows:

- $\hat{\mu}$, the mean of the data, is a Gaussian random variable with mean 0 and variance 1.
- $\hat{\sigma}^2$, the variance of the data, is a Gamma random variable with shape 1 and scale 1.
- $\hat{X}$ is a Gaussian random variable with mean $\mu$ and variance $\sigma^2$.

Figure 3.8 shows how this problem can be represented as a factor graph (left) and as a Bayesian network (right). This consists of three variable nodes – one for
3.4. GRAPH MODULE

Figure 3.8: On the left is an example of a Factor Graph. The graph illustrates the *Learning a Gaussian* example – $X \sim \mathcal{N}(\mu, \sigma^2)$ is a random variable, whose mean $\mu \sim \mathcal{N}(0, 1)$ and variance $\sigma^2 \sim \text{Gam}(1, 1)$ are also random variables. On the right is the representation of this as a Bayesian Network.

Each of $\mu$, $\sigma^2$ and $X$, and three factor nodes – representing the functions $\mathcal{N}(0, 1)$, $\text{Gam}(1, 1)$ and $\mathcal{N}(\mu, \sigma^2)$.

Infer.NET uses a factor graph to represent the graphical model on which inference is performed and it could serialise this to a DGML file when required. 

Appendix B gives the DGML file describing the *Learning a Gaussian* example, as serialised by Infer.NET 2.6.

### 3.4.2 Model Graph Class

To visualise the Bayesian network, I made use of the library Graph#. Graph# is a graph layout framework for WPF applications that depends on the data structures and algorithms of the library QuickGraph.

The *Model*, as specified by the MVVM pattern, is an instance of the *ModelGraph* class, which extends the QuickGraph’s `BidirectionalGraph<ModelVertex, ModelEdge>` class. It is a bidirectional graph with vertices being instances of `ModelVertex` and edges being instances of `ModelEdge`.

---

1 Directed Graph Markup Language – XML-based markup language used to describe directed graphs.
CHAPTER 3. IMPLEMENTATION

Figure 3.9: UML diagram of the Model.

ModelEdge. A UML diagram describing those inheritance relationships is shown in Figure 3.9. The ModelGraph object implements methods and constructors to support the deserialisation of Infer.NET’s factor graphs and the process of updating the Model until a complete graph representation of the probabilistic models in the source code is obtained.

The ModelVertex class is the substance of the graph object – the one that holds the information we would visualise about random variables. Some of the information it holds for each random variable is the following:

- the Name as specified in the source file, to include this in the visualisation.
- a Type specifying if the variable is observed or not, if it is an intermediate one, etc. This is used to provide visualisations distinguishing different kinds of variables from each other, as shown in §3.6.1.
- the Distribution as a string representing the distribution object returned by Infer.NET after inferring the variable.
- the Location of the definition in the source code, to allow UI-to-Code navigation to be implemented, as in §3.6.1.

Figure 3.10 shows the structure of the ModelGraph objects. Having the Model being an instance of it turns the implementation of additional features, such as incorporating variables’ distributions into the graph itself, having tooltips or hints, visualising the factor graph, into simply a matter of extending and handling appropriately the ModelVertex object.
3.4. GRAPH MODULE

3.4.3 Converting a Factor Graph to a Bayesian Network

As previously shown, a factor graph contains additional nodes, which are not part of the Bayesian network describing a probabilistic model. Thus, given the Infer.NET’s factor graph, the underlying Bayesian network must be extracted.
An important note is that the serialised by the framework file is assumed to
represent the factor graph of a Bayesian network (rather than of a Markov random
field, for example) and the IDE will not work correctly in other cases.

The conversion of a deserialised DGML file to the desired ModelGraph object
(which is to be united with the rest of the Model) happens in two steps:

1. Build a ModelGraph object, which represents the factor graph de-
scribed in the deserialisation target instance of a DirectedGraph object. DirectedGraph is a direct translation of the DGML markup, which means
it contains only an array of labels of particular attributes, rather than ele-
ments containing pointers to related elements. Thus, the presence of this
step eases the implementation of step 2 and, furthermore, allows the pro-
gram to be easily extended to support factor graph visualisation if necessary.

2. Convert the factor graph of step 1 into a Bayesian network. This can be
done by simply adding a node for every variable node of the factor graph
(i.e. each circular node) and then performing a simple search from each
node, to reach another variable node. A pseudocode describing the way the
IDE performs the factor-to-bayesian graph conversion, using a depth first
search, is given below:

```plaintext
procedure FactorToBayesian(FactorGraph)
    BayesianNet.Vertices ← Filter(FactorGraph.Vertices)
    for all V ∈ FactorGraph.Vertices do
        if V ∈ BayesianNet.Vertices then
            Stack ← new(V.Neighbours)
            while Stack is not empty do
                Current ← Stack.Pop()
                if Current is not visited then
                    if Current is a variable node then
                        BayesianNet.Edges+ = Edge(V,Current)
                    else
                        push all neighbours of Current to Stack
                    end if
                end if
            end while
        end if
    end for
    return BayesianNet
end procedure
```
3.5 Providing Liveness

So far, this chapter focused on the Visualisation Update Subsystem (VUS) of the IDE, which ensures that once the programmer decides to compile her program, the code is type-checked, compiled, executed and visualised. This section describes how the user’s interaction with the edit-compile-run cycle is made smoother by implementing liveness in the environment.

3.5.1 Edit-triggered Re-compilation

Re-compilation was made edit-triggered, to eliminate the need to re-compile and code at repeated intervals. This means that the programmer has to change her code to initialise re-compilation.

Edit-triggered re-compilation could have been implemented by simply using the WPF TextBox event TextChanged. However, this would mean re-compiling every time a new key is being hit, which is unproductive, given that the user would normally type several characters in a row. Similarly to some implementations of a search boxes, where a list of suggestions is only shown after the user makes a pause while typing, I implemented throttling of the input, using ReactiveX [8].

ReactiveX is an API for asynchronous and event-based programming that extends the Observer pattern to allow the programmer to compose data or event sequences together, while abstracting away low-level threading, synchronisation, etc. All data/event sequences are Observables to which an Observer can subscribe. The Observer then receives asynchronous notifications as the Observable emits data.

In my program, a new Observable is created, which emits the current script written in the IDE 500 milliseconds after it was last changed by the user. A Subscriber method, Recompile, is then subscribed to the Observable object to be called every time script is emitted. Recompile runs the VUS in a new thread as described in the next subsection.
3.5.2 Concurrent Re-compilation

Threading had to be implemented carefully, to ensure that the IDE stays reactive, so the user can make changes at any time, no matter if the VUS is currently executing or not. Moreover, it had to be ensured the IDE behaves as expected without crashing due to, for instance, incorrectly accessing a shared resource.

To allow the IDE to stay reactive, recompilation is ran in a new thread. The Visualisation Update Subsystem is split into three critical sections: the Checking Phase, the Injection Phase, and the transition between the two.

The pseudocode below shows how this was implemented.

```plaintext
module VUS(
    acquire checkLock
    Checking(
        if no parse errors then
            abort injectingThread
            acquire injectLock
            injectingThread ← this
            release checkLock
            Injection(
                else
                    report error and exit VUS’s execution
                    release checkLock
                    end if
                end module
    
    The presence of the first critical section ensures that at any time, only one thread is parsing and type-checking code, which prevents the temporary file containing the user’s source code to be corrupted and allows F#’s Compiler Services to work correctly. The second critical section aims to eliminate problems occurring due to simultaneous trails to compile two or more Infer.NET models or infer two or more sets of variables. The sections are kept separate, as Injection Phases’s execution doesn’t interfere in any way with that of the Checking Phase and allows one run of the VUS to interleave with another, which improves performance. To ensure mutual exclusion of the respective critical section, the code of each section is executed only after obtaining the respective lock associated with it – checkLock or injectLock.

However, finishing the execution of the Injection Phase is not always needed. It is
possible that the user makes another change to the code before waiting to see the result of an older one. Thus, the IDE should switch to updating the visualisations with respect to the most recent code entered by the user, cancelling executions that use code, which is no longer valid.

To implement this, the program keeps track of the last VUS thread that entered the Injection Phase and aborts its execution whenever a more recent thread wants to start injecting. Figure 3.11a shows an example of this situation. The update of the object that represents this last thread is the transition between the Checking and Injection Phases and is the third critical section, entered after acquiring both checkLock and injectLock. Cancelling older threads execution after the Checking Phase rather than before entering the VUS, allows the IDE to still visualise the last correctly parsed code even in the presence of a newer input, which could not compile due to, say, a syntax error. Figure 3.11b describes this situation with a concrete example.

(a) Example of a newer thread terminating the execution of an earlier one because it posses a correctly parsed up-to-date version of the code.

(b) Example of a newer thread leaving the execution of the previous, correctly parsed version of the code uninterrupted because of the presence of parse errors.

Figure 3.11: Illustration of different possible simultaneous executions of the Visualisation Update Subsystem.
3.6 User Interface

This section describes in more detail what elements were implemented in the View and why. Firstly, the visualisation of the main components, Bayesian network (§§ 3.6.1) and random variable's distributions (§§ 3.6.2) is described. The section continues with some notes about the simulation of a console (§§ 3.6.3) and finishes with other user interface components that were included to enhance user experience (§§ 3.6.4).

3.6.1 Graph Layout

The graph describing the graphical model is being layout by the Graph# library. It is set to use the commonly used Sugiyama algorithm [34] for the layout of directed graphs. This algorithm orders vertices so they appear in horizontal rows or layers with the edges generally directed downwards and minimising any crossings between edges. However, Sugiyama is enforced only when the graph visualisations is refreshed. After that the user could freely move around nodes as they wish.

![Graph drawn with Sugiyama algorithm](image1.png) vs. ![Graph with random node position](image2.png)

Figure 3.12: Graph drawn with Sugiyama algorithm (on the left) compared to a graph which node position was assigned at random (on the right).

Visualising Observed Variables

Recall that observed variables are those random variables, for which a concrete value has been observed. To be able to distinguish them from non-observed ones, the user sees them filled in with solid colour, as in Figure 3.13.
3.6. USER INTERFACE

Figure 3.13: Screenshot showing observed variables visualisation. The random variables visitedAsia and abnormalXRay have been observed.

Visualising Array Variables

These are shown as rectangles instead of circles – Figure 3.14. A possible extension would be improving this visualisation to include *plates* [12] – a standard notation of variable arrays. However, this was considered out of the scope of this project as it requires keeping more complicated information about how variables are related.

Figure 3.14: Screenshot showing array variables visualisation. The variables $x$ and $y$ are array variables both of size $n$. $x$ has been observed, $y$ has not been.

Visualising Intermediate Variables

Infer.NET sometimes creates additional variables that are also shown in the DGML file. Consider the following example:

```csharp
let A = Variable.Bernoulli(0.2)
let B = Variable.Bernoulli(0.9)
let C = Variable.Bernoulli(0.3)
let D = A &&& B &&& C
```
Infer.NET would translate this to:

```latex
let A = Variable.Bernoulli(0.2)
let B = Variable.Bernoulli(0.9)
let C = Variable.Bernoulli(0.3)
let vbool0 = A &&& B
let D = vbool0 &&& C
```

To distinguish those temporary variables in the visualisation and minimise user’s confusion about them appearing, a grey colour was used for their labels as shown in Figure 3.15. An extension would provide the option of seeing a graph, which removes those entirely.

![Intermediate Variables Visualisation](image)

**Figure 3.15:** Screenshot showing intermediate variables visualisation. *vbool213* is an intermediate variable produced by defining *all* as *A &&& B &&& C*.

### Tooltips

Each node also has a tooltip that appears when the mouse is hovered over that node, as shown in Figure 3.16. The tooltip text is the exact string representation of the object returned by Infer.NET after inferring the corresponding variable. That was included to allow the user easily see that string representation, in the cases when visualisation is not supported or not descriptive enough for the respective distribution. For example, a Gaussian distribution with large variance would look completely flat, a Bernoulli random variable with probability of 0.9999 being true, won’t be distinguishable of one, which is always true, etc.

This shows the direct correspondence between each node in the View and their respective nodes in the underlying Model. Accordingly, the IDE could easily be extended to collect and exploit other useful information about variables.
3.6. USER INTERFACE

Figure 3.16: Screenshot showing tooltips visualisation. Tooltips appear when hovering over the nodes of the graph, showing the exact string returned by Infer.NET when inferring the respective variable.

UI-to-Code Navigation

As will be described in the next chapter, the results of the user study could not prove with significance that the IDE benefits completing debugging tasks, although it improves performance when completing other kinds of tasks and overall. The absence of clear pointers from elements of the visualisation to the associated code was seen as a possible reason for that. Thus, UI-to-Code Navigation was implemented, in the sense that when the mouse enters a graph node, the code that defines the variable this node represents is highlighted. Figure 3.17 shows a screenshot of the usage of this feature.

This is implemented by storing additional information about the definition line in the ModelVertex objects, that represent the nodes of the graph. Even though this does not provide much on its own, it could be seen as a first step towards a more complicated, bidirectional UI-Code Navigation, such as that described by Burckhardt et al. [13]. Moreover, it is a necessary foundation for a feature, which allows the programmer to change her code through the visualisation itself – direct manipulation [32].

3.6.2 Visualising Inferred Distributions

After injecting an Infer() statement in the implementation and injection phase, an Object is returned, which represents the posterior distribution of the variable. This is then converted to a string of the form <DistributionName>(<parameters>) (e.g. Gaussian(0.0, 1.0)) and passed to the Distributions module, which handles the visualisation.
CHAPTER 3. IMPLEMENTATION

The Distributions module is written in F# and uses pattern matching on the distribution string to find the correct way to visualise each distribution. The following F# code is an overview of the Distribution module:

```fsharp
let drawDistribution d =
    let name = d.FirstPartOfString
    match name with
    | "Discrete" -> plot a discrete distribution
    | "Gaussian" -> plot a gaussian curve
    | "Gamma" -> plot a gamma distribution
    | "Poisson" -> plot a poisson variable
    ...
```

Within a single case, the parameters are also extracted from the distribution’s string and then used, together with the respective distribution function and the F# Charting library [3] to create a Windows Forms Host element visualising the distribution. The function to be plotted (in the case of continuous variables) is directly taken through the Infer.NET binaries, to avoid inconsistencies. For instance plotting a Gaussian curve could be roughly described as follows:
let drawDistribution d =

| "Gaussian" -> plot a gaussian curve
let mean = getFirstParameter(d)
let variance = getSecondParameter(d)
let gaussian m v x =
    Gaussian.FromMeanAndVariance(m, v).GetProb x

let chart = Chart.Line (gaussian mean variance)

Visualisation is not supported for all distributions. However, programs will execute correctly even when there exist nodes whose distributions are not supported, even though a visualisation won’t appear. Adding visual support for more distribution could be done in a straightforward way, by adding another case in the drawDistribution function.

Working on the Distribution module rose some interesting questions, for example: how to visualise a Gaussian distribution informatively, to allow comparisons to happen in the programmer’s mind, instead of making all curves look the same (apart of axis scale, say). Figure 3.18 shows an example of a situation in which this applies.

### 3.6.3 Console

A simulation of a console was needed, in order to allow:

- Error messages due to parse or type-check errors to be output.
- Evaluation of print statements written in the source code to be output.

The console was implemented as a TextBox element, cleared every time a compilation is initialised. If the checking phase gives any parse or type errors, they are thrown as an exception out of the Checker Module and then output in the console. In addition, the text colour of the console is changed to red, to draw user’s attention.

If the source code is successfully checked and interpreted, the console outputs results of any print statements in the code. However, the interpretation of a print statement by F# interactive sends the result to the standard output. Thus, outputting the respective results was implemented by:
CHAPTER 3. IMPLEMENTATION

Figure 3.18: Visualisation of the distributions random variables are drawn from. $G$ is a Gamma random variable, $P$ - a Poisson random variable, $B$ - a Bernoulli one. $N_1$, $N_2$ and $N_3$ are all drawn from Gaussian distributions with mean of 0 and different variances. The figure shows how the visualisation allows comparisons to be easily made.

1. Replacing all `print` statements by `eprint` statements in the checking phase. In F#, as `print` writes data to the standard output stream, `eprint` writes data to the standard error stream.

2. For each recompilation, redirecting the standard error stream to a new `TextWriter` object, whose content is then written in the IDE console text box.

3.6.4 User Experience Improvements

A few interface elements that were not subject to the success of the project were added to enhance the user’s experience by making the IDE feeling as a more natural programming environment, showing indication of progress, etc.
3.6. USER INTERFACE

Source Code Editor

AvalonEdit [1], a text editor component binaries for WPF applications, was used for the editor embedded in the IDE. The text editor uses a spacial XSHD file (XML based markup file) to implement text colouring. As F# was not among the supported by AvalonEdit languages, it had to be manually added. This was done by adapting the C# XSHD file through adding more keywords and rules.

A standard menu allowing the user to save and open F# script files from the hard disk was also implemented.

System’s state

Visual feedback about the state of the system (i.e. is it idle, compiling, executing) is provided with the aid of:

- **Progress bar**: Shows an approximation of the amount of work that has been completed. The progress bar is hidden when the IDE is idle.

- **Text**: Next to the progress bar, a label describing the current state of the system is shown. The label takes one of “Checking...”, “Compiling...”, “Inferring...” or “Could not compile.” depending on the exact work that is being currently done by the IDE. This text label is hidden when the IDE is idle.

- **Staleness indicator**: The visualisation of the graph is dimmed by a semi-transparent grey cover whenever recompilation is initiated. Thus, the IDE signals that the current visualisation is not the most recent one and work is being done to update it.
CHAPTER 3. IMPLEMENTATION

Figure 3.19: Screenshot of the IDE while it is recompiling. The image shows the progress bar indicating that approximately a third of the work has been completed, the text label indicating that the program is currently being compiled, and the cover, greying out the graphical model as not being up-to-date.

3.7 Summary

This chapter described the implementation of the IDE, by:

1. Illustrating the work of the Visualisation Update Subsystem of the IDE – the code responsible for recompiling user’s code – in §3.2, §3.3 and §3.4

2. Showing how that was run in different threads to execute in parallel and make the IDE responsive and live in §3.5

3. Giving a brief description of the implemented UI components in §3.6
Chapter 4

Evaluation

“Computers are useless. They can only give you answers.”

— Pablo Picasso

This chapter describes the process of designing and conducting a controlled experiment to evaluate the software with respect to two main aspects:

- **Usability**: does the presence of the software improve Infer.NET programmers experience relative to a plain text editor?

- **Learning**: does the presence of the software improve the ability of novices to learn about probabilistic programming relative to a plain text editor?

In addition the experiment searched for dependencies between participants’ general programming experience and their benefit of the IDE and studied the effects of the software on self-confidence.

In summary, the results show that people performed better in terms of time, number of keystrokes and corrections when using the IDE. Moreover, those learning with the tool completed their training with significantly lower number of keystrokes. No significant evidence on improvement of self-confidence levels was found.

The design of the experiment is outlined in § 4.1 while results are extensively presented and discussed in § 4.2.
4.1 Experiment Design

This section outlines the design of the experiment. It describes how participants were chosen, justifies the choices that have been made, explains the design of the experimental tasks and finally describes the process of conducting the experiment.

4.1.1 Considerations

In order to be able to study the effects of the IDE on learning, the participants in the experiment needed to be with no or little prior experience with probabilistic programming. However, they had to have experience with general purpose programming and probability theory. Enough experience in probability theory was considered having covered the prerequisites needed for the “Artificial Intelligence II” course from the Computer Science Tripos.

The study was conducted on a total of 16 University of Cambridge Part II students, 14 of which reading Computer Science and 2 of which Mathematics. Both of these groups had enough experience in each topic due to either the very nature of their subjects, or by having AI II in their syllabus for the current year.

Finally, as will be discussed in the next section, an entirely within-user study was not possible. Also, the overall number of participants had to be reasonable with respect to the timespan of the project. Thus, to improve validity of results of the between-subjects study performed, special effort was put into decreasing the number of confounding variables by conducting the experiment in the same sitting.

4.1.2 Overall Design

To cover both hypothesis points mentioned in the beginning of the chapter, that is improving learning and improving usability, the experiment was designed in a mixed factorial manner, having a between-subjects study to evaluate learning and within-subject study to evaluate usability. This was done as follows:

- A workbook, further discussed in §§ 4.1.3 was designed to consist of two section, such that:
  - Part 1 interleaves necessary theory with examples and exercises and aims to evaluate learning.
4.1. EXPERIMENT DESIGN

Figure 4.1: Illustration of the overall design of the experiment

- Part 2 consists of additional exercises aiming to evaluate usability.

- Participants were initially split into one of two groups, on which a between-subjects study was conducted later:
  - Group A completes Part 1 using the IDE.
  - Group B completes Part 1 using a plain text editor.

- Each group was then further split into four, using a $2 \times 2$ counterbalanced Latin square design, to eliminate difference in order and difficulty effects in Part 2 of the experiment and to allow for an additional within-subject study to be performed.

Due to the very nature of learning (one can’t learn the same thing twice) within-subject study was impossible when it comes to evaluation of learning. Thus, to study the effects of learning with the IDE, part of the experiment needed to be organised in a between-subjects manner. This would mean that proving the results as being statistically significant would be harder, as the number of participants involved could not be very high. Naturally, only partial evidence for statistical significance of the effects on learning were found, as described in §4.2.
However, splitting the participants into “visual learning” and “text learning” groups also helped for fully counterbalancing the within-subject study for usability. By learning about a concept through a particular interface, a participant would also learn to use that particular interface. Thus, an experiment considering only participants who studied with the aid of the IDE or only participants who studied with a plain text editor, would have biased the results.

4.1.3 Workbook Design

This subsection gives an overview and justifications of the workbook design. The complete workbook is given in Appendix C.

Due to the context of the study, that is examining learning, the experiment materials themselves had to carry some teaching material of the background needed to conduct the experiment. Thus, it was decided to design a workbook, conceptually similar to those of the Programming in Java and Further Java courses from the Computer Science Tripos.

The workbook consists of two parts as follows:

Part 1 introduces the participant to F#, probabilistic programming and Infer.NET. It interleaves text explanations, four code examples and four exercises.

Part 2 consists of 8 additional exercises. The exercises were chosen in such a way, that each of them is part of a pair of equal difficulty. Thus, they could be split into two groups that we assume are equivalent, i.e. that each question in one group has a counterpart in the other. Each participant completes the set of exercises of one group with the IDE and that of the other – with plain text editor. To ensure that the results are counterbalanced, the participants were assigned at random in one of four groups that attempts these in a particular order of exercises and order of IDE using. This is illustrated in Figure 4.2.
4.1. **EXPERIMENT DESIGN**

The exercises in part two are two main types – *Debugging* exercises and *Observation* exercises. The former aims to explore how well the participants are able to find and correct mistakes, while the latter, how well they understand the model that they see.

All groups of participants were given the same workbook, up to order of exercises in Part 2.

### 4.1.4 Experimental Procedure

This subsection gives an insight on the process of conducting the experiment. The number of keystrokes, backspaces and time needed to complete each question were gathered. That was done by the evaluation software itself and written to the hard disk in a log file. Moreover, the evaluation software was automatically arranging exercises in the correct order and enabling/disabling visualisations for particular exercises based on the Group ID entered in the beginning of each experiment. Screenshots of the evaluation software at different stages of the experiment are given in [Figure 4.3](#). Participants were asked to vocalise their answers and were given indication if those are correct or incorrect.

Number of correct answers was not recorded, instead participants were asked to correct themselves if they have made a mistake and guided through until a correct solution is reached. That was due to the nature of study – it required understanding complicated concepts and answering questions that are rather long.

An individual study took 92 minutes on average.

**Pilot Studies**

Two pilot studies were successively carried out with two different participants to identify potential issues and improve on the design of the experiment. Changes to the workbook were made accordingly to the results from the pilot studies, where they identified questions with unclear instructions. Moreover, a few additional changes were made to the software to provide a better user experience for the participants throughout the experiments. These include improving the messages output in the IDE’s simulated console, additional tooltips support, refining the feedback that users receive about the state of the environment.
(a) An exercise loaded by the evaluation software with the IDE enabled.

(b) The same exercise loaded by the evaluation software with the IDE disabled.

(c) An example of the intermediate screen appearing between exercises.

Figure 4.3: The evaluation software
Consent Form and Questionnaires

The main experiment involved sixteen participants, who were firstly asked to complete a consent form, which explained the purpose and procedure of the experiment, the data that would be gathered and the analysis that would be carried out on that data. Participants were given a copy of the consent form and were informed of their User ID number. Each participant was then asked to complete two questionnaires, which are attached in Appendix D.

The first questionnaire, a self-assessment of participant’s programming skills, was used to later rank the participants by experience in programming, and find out if that is a confounding variable. The rank was also used to see if more experienced participants would benefit less/more of the software than inexperienced participants. No dependence between the users’ programming skills and their interaction with the IDE was found.

Moreover, to study the changes of self-confidence due to learning with or without the IDE, participants’ self-efficacy was assessed with the 10-item General Self-Efficacy Scale. Participants were asked to answer the questions with respect to probabilistic programming and they filled the questionnaire twice – once before the experiment and once after.

Completion of the Workbook

Part 1 interleaves text, four examples of F# / Infer.NET programs and four exercises. The participants could view and explore freely the examples using the evaluation software, taking as much time as they wish. While some of them could use the visualisations that the IDE provides to do that, others had to manually inspect examples by inferring values by themselves.

Part 2 consists of 8 more exercises, half of which the participants had to complete with the aid of the IDE and the other half, with the plain text editor the evaluation software provides.

Participants were instructed that the exercises are timed, that they can ask questions at any time and that they can say their answers to me.

Belief in individual’s own capacity to execute behaviours necessary to produce specific performance attainments. It reflects confidence in the ability to exert control over one’s own motivation, behaviour, and social environment.
CHAPTER 4. EVALUATION

4.2 Results

This section gives a summary of the numerical results obtained after performing suitable statistical tests on the quantitative data gathered from the experiments. Results suggest that the IDE has fulfilled the purpose for which it was designed, namely increasing usability of the probabilistic programming framework Infer.NET. Additional tests were performed to analyse the effects of the tool on learning and on self-confidence and the results of those suggest that it might be having more effect on those compared to a text editor.

An $\alpha$ value of 0.05 was chosen for all statistical tests below.

4.2.1 Usability

A within-subject study was conducted on the data gathered from the exercises in Part 2 to evaluate the usability of the tool. This was done using a paired Wilcoxon signed-rank test, rather than a paired t-test, as I found no statistical evidence that the data is normally distributed (p-value was smaller than 0.05 in almost all cases, using Shapiro-Wilk normality test). Data was treated, so it pairs each exercise with its “twin” exercise within the same user. The results of the paired tests are summarised in Figure 4.4.

Overall results for Part 2 Exercises

Figure 4.4a shows the results of the Wilcoxon tests carried out on all exercises in Part 2, regardless of the type of exercise. The p-values obtain show with statistical significance that in overall, the IDE provided an advantage in completing probabilistic programming exercises over a plain text editor.

On average, each participant completed a Part 2 exercise, 1 min 20 sec faster with the IDE than with a text editor and took 22.5 keystrokes less.

Part 2 Debugging Exercises

Debugging type exercises asked the participants to find and correct several mistakes made in the code, with respect to the particular problem described in the workbook. As shown in Figure 4.4b, the p-values for all three measurements are greater than the chosen $\alpha$ value, thus the performance gains due to the IDE could not be shown with significance. One reason for this could be that even though
4.2. RESULTS

<table>
<thead>
<tr>
<th>Measurement</th>
<th>V</th>
<th>p-value</th>
<th>IDE median</th>
<th>Text median</th>
<th>Difference median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>611</td>
<td>0.004162</td>
<td>288 sec</td>
<td>359 sec</td>
<td>80 sec</td>
</tr>
<tr>
<td>Keystrokes</td>
<td>302</td>
<td>0.002044</td>
<td>15.5</td>
<td>79</td>
<td>22.5</td>
</tr>
<tr>
<td>Backspaces</td>
<td>367.5</td>
<td>0.02397</td>
<td>3.5</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) Results for all Part 2 exercises.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>V</th>
<th>p-value</th>
<th>IDE median</th>
<th>Text median</th>
<th>Difference median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>205</td>
<td>0.2781</td>
<td>323 sec</td>
<td>337 sec</td>
<td>18 sec</td>
</tr>
<tr>
<td>Keystrokes</td>
<td>167</td>
<td>0.07112</td>
<td>34</td>
<td>90</td>
<td>64.5</td>
</tr>
<tr>
<td>Backspaces</td>
<td>188.5</td>
<td>0.2472</td>
<td>9</td>
<td>12</td>
<td>3</td>
</tr>
</tbody>
</table>

(b) Results for debugging exercises.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>V</th>
<th>p-value</th>
<th>IDE median</th>
<th>Text median</th>
<th>Difference median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>101</td>
<td>0.001667</td>
<td>245 sec</td>
<td>369 sec</td>
<td>92 sec</td>
</tr>
<tr>
<td>Keystrokes</td>
<td>12</td>
<td>0.002448</td>
<td>0</td>
<td>1.5</td>
<td>0</td>
</tr>
<tr>
<td>Backspaces</td>
<td>31</td>
<td>0.0331</td>
<td>0</td>
<td>36.5</td>
<td>0</td>
</tr>
</tbody>
</table>

(c) Results for observation exercises.

Figure 4.4: Results of paired Wilcoxon tests on Part 2 exercises. The *Difference median* column gives the median of the values obtained by taking the difference between the text editor values and the IDE values of paired exercises.

participants were able to quickly see major mistakes in the model with the aid of the visualisations, the IDE failed to guide them to the corresponding piece of code, meaning that the advantage it provides in this cases is minimal. This led to later implementation of the UI-to-Code Navigation feature, as described in §§ 3.6.1.

Nevertheless, participant took 64.5 keystrokes and 3 backspaces less on average to complete a debugging exercise with the IDE, compared to a plain text editor.
Part 2 Observation Exercises

Observation type exercises asked the participants to view an Infer.NET model with the evaluation software and explain out loud what problem this model solves, together with some important dependencies between variables. Results of the Wilcoxon tests for all three measurements [Figure 4.4c], show with significance that the IDE provided advantage in understanding and performance over text-editor. This is especially relevant for the time and keystrokes measurements, where the significance is very high ($p < 0.001$).

On average, participants were able to finish an observation type exercise 1 min 32 sec faster when making use of the IDE.

4.2.2 Learning

A between-subjects study was also conducted on the gathered data, to determine potential effect of the IDE on learning compared to a plain text editor. In brief, people who learnt with the aid of the IDE became confident with the new material more quickly than participant learning with a plain text editor. Moreover, the type of learning didn’t have any effect on the performance in Part 2, which shows that the IDE was useful in a way that doesn’t cause any disadvantage at a later stage.

Part 1 Exercises

Initially, the measurements for time, keystrokes and backspaces were treated separately and whole set of exercises in Part 1 were analysed with Wilcoxon rank sum test. Results for each measurement are shown in Table 4.1.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>V</th>
<th>p-value</th>
<th>IDE median</th>
<th>Text median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>217</td>
<td>0.1472</td>
<td>191 sec</td>
<td>220 sec</td>
</tr>
<tr>
<td>Keystrokes</td>
<td>195.5</td>
<td>0.05475</td>
<td>28.5</td>
<td>93</td>
</tr>
<tr>
<td>Backspaces</td>
<td>250</td>
<td>0.4316</td>
<td>3.5</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 4.1: Results of Wilcoxon tests on Part 1 exercises.

As all $p$ values are larger than the $\alpha$ level, the test could not prove with significance that the IDE provides benefit in any of the three measurements. A possible
4.2. RESULTS

(a) Average time taken by each participant to answer Part 1 questions.

(b) Average keystrokes taken by each participant to answer Part 1 questions.

(c) Average backspaces taken by each participant to answer Part 1 questions.

Figure 4.5: Average measures of Part 1 exercises.

explanation of these results might be that the test, when conducted in this way, doesn’t take into account the fact that some groups of exercises are completed
CHAPTER 4. EVALUATION

by the same user. Thus, the average time, keystrokes and backspaces taken to complete a Part 1 exercise were calculated for each user [Figure 4.5]. A one-way MANOVA test was run. The three measurements were treated as three different depended variables. MANOVA showed with significance that one or more of these variables is depended on the type of learning (p value of 0.00288).

The follow-up analysis included running ANOVA on each dependant variable separately. The results of those tests are summarised in Table 4.2 and show with significance that the type of learning has an effect on the number of keystrokes. The Bonferroni correction was applied to the α value, it then becoming 0.0167.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>F-value</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1.782</td>
<td>0.203</td>
</tr>
<tr>
<td>Keystrokes</td>
<td>23</td>
<td>0.000285</td>
</tr>
<tr>
<td>Backspaces</td>
<td>2.094</td>
<td>0.17</td>
</tr>
</tbody>
</table>

Table 4.2: Results of ANOVA tests on Part 1 exercises.

In both the MANOVA and ANOVA tests log of time and keystrokes values was analysed instead of the actual values, to normalise the data.

**Part 2 Exercises** To analyse further the effects of the IDE on learning experience, similar tests were also run on the Part 2 data. However, neither could show any difference between performance of participants of the two learning groups. Figure 4.6 shows boxplots describing the analysed data.

![Boxplots](image)

Figure 4.6: Boxplots showing the distribution of each of the three measurements (time, keystrokes and backspaces) in Part 2, with respect to the type of learning of participants.
4.2.3 Confidence

Changes in self-confidence were measured by asking the participants to complete the General Self-Efficacy Scale, before and after the experiment, as described in §§ 4.1.4. Their scores, a number between 1 and 4, were analysed in a between-subject manner using Wilcoxon rank sum test, which gave a p-value of 0.6733, which could not be used to show any effects of the IDE on self-confidence.

However, the data shows that the self-confidence improvement of people who completed the learning phase with the aid of the plain text editor is 0.24, while the IDE gave an improvement of 0.34. The latter’s distribution also has a smaller deviation, with the exception of two outliers, which could be clearly seen from Figure 4.7. Given the small sample size of only 8 participant per group, it is reasonable to conclude that further experiments have to be conducted to determine the effects of learning with the IDE on self-confidence.

4.2.4 Discussion

Overall, most participants found the experiment interesting and had especially strong reactions and comments when visualisations were appearing or disappearing.

The following interesting observation was made during the experiments: at least half of the participants who learned with the aid of the IDE stated at some point later, while completing an exercise with a plain text editor instead, that they are trying to visualise the Bayesian network in their heads to be able to understand
CHAPTER 4. EVALUATION

<table>
<thead>
<tr>
<th>Comment</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Correct the errors’ – how am I supposed to – the arrows were so useful for errors!</td>
<td>Debugging exercise, text editor.</td>
</tr>
<tr>
<td>Argh, no graph, nooo!</td>
<td>Change from IDE to text editor.</td>
</tr>
<tr>
<td>I would really like the thing with the arrows.</td>
<td>Debugging exercise, text editor.</td>
</tr>
<tr>
<td>Looking at the code is horrific... but looking at the graph is not that bad.</td>
<td>Observation exercise, IDE.</td>
</tr>
<tr>
<td>You see straight from the IDE...</td>
<td>Observation exercise, IDE.</td>
</tr>
<tr>
<td>I’m not going to look at the code anymore!</td>
<td>Change from text editor to IDE.</td>
</tr>
<tr>
<td>I completely missed (that) dependency.</td>
<td>Observation exercise, text editor.</td>
</tr>
<tr>
<td>Found that quite fun.</td>
<td>Finishing with the experiment.</td>
</tr>
</tbody>
</table>

Table 4.3: Some comments received during the experiments.

the dependencies between variables better. Moreover, when not having the visualisations and asked to explain out loud a model, participants seemed to be more likely to start reading the code, line by line, in order to answer the question. They were giving extensive information about the exact way prior probabilities are specified (e.g. “when Cloudy is true, Rain is Bernoulli(0.7), otherwise – Bernoulli(0.1).”). In comparison, participants having the IDE when asked to explain a model seemed more likely to talk in terms of dependencies and to see connections otherwise not that obvious from the source code.

4.3 Summary

This chapter explained the process of designing and conducting a user study to evaluate the usability of the IDE. It also analysed the results, concluding that the environment improves both the overall and the learning experience of students.
Chapter 5

Conclusion

“Tell me and I forget. Teach me and I remember. Involve me and I learn.”
— Benjamin Franklin

This work presents an IDE for Infer.NET, which is specially designed to support through live visualisations working with the probabilistic programming framework Infer.NET. Update is edit-triggered and allows the programmer to see an up-to-date interactive graph representing the underlying graphical models. Plots describing the way random variables are distributed are also included. The software is novel in that it both graphically represents the program data and provides feedback about the program execution in a live way.

The project goals, as outlined in the project proposal, were met, with all components completed on schedule. Furthermore, additional user interface features were implemented to enhance user experience, such as source code highlighting, tooltips, UI-to-Code navigation.

In addition to implementing the IDE, a user study was conducted to evaluate its usability. The results show that the tool significantly improves user experience by reducing the task completion time and the level of confusion within a user. Moreover, they suggest that the IDE also improves the learning process of students encountering probabilistic programming concepts for the first time.

Even thought the project was successful, I faced many challenges during the implementation process. The reason for the most significant one, scalability, could not be overcome. Infer.NET, even though it performs inference efficiently, does not provide efficient access to the graphical models it compiles. Furthermore,
it requires the entire state of the program to be destroyed on recompilation, as there is no way of incrementally changing the model. The need to invoke the Infer.NET compiler on every update sets limits on the amount of liveness the IDE could provide, especially if working with large models.

Probabilistic graphical models provide a very important feature from program visualisation perspective – they give the user a way to think about complicated data structures visually. This means that probabilistic programming languages have great potential to be supported by visual program representation or even visual programming. However, developing tools which provide such support would require the underlying language to be designed in a way which encourages visualisation.

In conclusion, building an IDE for probabilistic programming is an achievable goal, although it could be a challenging process, which involves lots of uncertainties. This project proves that live visualisation is beneficial in the context of probabilistic programming and could improve the learning experience of beginners. I very much enjoyed working on this problem and I feel confident that the IDE could be used to support probabilistic programming.
Bibliography


Appendix A

Sprinkler

1 let Cloudy = Variable.Bernoulli(0.5)

2

3 let Sprinkler = Variable.New<bool>()
4 let Rain = Variable.New<bool>()
5 let Wet = Variable.New<bool>()

6 // Sprinkler’s and Rain’s CPTs:
7 begin
8     use ifC = Variable.If(Cloudy)
9         Rain.SetTo(Variable.Bernoulli(0.8))
10        Sprinkler.SetTo(Variable.Bernoulli(0.7))
11 end
12
13 begin
14     use ifNotC = Variable.IfNot(Cloudy)
15        Rain.SetTo(Variable.Bernoulli(0.2))
16        Sprinkler.SetTo(Variable.Bernoulli(0.05))
17 end
18
19 // Wet’s CPT:
20 begin
21     use ifS = Variable.If(Sprinkler)
22        begin
23             use ifR3 = Variable.If(Rain)
24                 Wet.SetTo(Variable.Bernoulli(0.99))
25             end
26 end
APPENDIX A. SPRINKLER

26     end
27     begin
28     use ifNotR3 = Variable.IfNot(Rain)
29     Wet.SetTo(Variable.Bernoulli(0.9))
30     end
31     end
32
33     begin
34     use ifS2 = Variable.IfNot(Sprinkler)
35     begin
36     use ifR4 = Variable.If(Rain)
37     Wet.SetTo(Variable.Bernoulli(0.9))
38     end
39     begin
40     use ifNotR4 = Variable.IfNot(Rain)
41     Wet.SetTo(Variable.Bernoulli(0.0))
42     end
43     end
44
45     let engine = new InferenceEngine()
46     printf "%A" (engine.Infer(Wet))
47
48     Wet.ObservedValue <- true
49
50     printf "%A" (engine.Infer(Cloudy))
Appendix B

The Learning a Gaussian Example

The code below is the DGML markup of the Learning a Gaussian example, as serialised by Infer.NET. Figure B.1 is the visualisation of this file by Visual Studio.

```xml
<?xml version="1.0" encoding="utf-8"?>
<DirectedGraph xmlns="http://schemas.microsoft.com/vs/2009/dgml">

    <Nodes>
        <Node Id="node4" Label="GaussianFromMeanAndVariance" FontSize="8"
            Foreground="#ffffffff" Background="#ff000000" NodeRadius="0" />
        <Node Id="node5" Label="v" FontSize="10"
            Foreground="#ff0000ff" Background="#00ffffff" NodeRadius="100" />
        <Node Id="node6" Label="X" FontSize="10"
            Foreground="#ff0000ff" Background="#00ffffff" NodeRadius="100" />
        <Node Id="node7" Label="1" FontSize="9"
            Foreground="#ff000000" Background="#00ffffff"
            Shape="None" NodeRadius="0" />
        <Node Id="node8" Label="Sample" FontSize="8"
            Foreground="#ffffffff" Background="#ff000000" NodeRadius="0" />
        <Node Id="node9" Label="1" FontSize="9"
            Foreground="#ff000000" Background="#00ffffff"
            Shape="None" NodeRadius="0" />
        <Node Id="node0" Label="0" FontSize="9"
            Foreground="#ff000000" Background="#00ffffff"
```

69
APPENDIX B. THE LEARNING A GAUSSIAN EXAMPLE

Figure B.1: A visualisation of the Learning a Gaussian example.
Appendix C

Workbook

This workbook is divided into two sections.

Section 1 consists of a brief introduction to probabilistic programming, F# and Infer.NET and 4 interleaving exercises. While you are reading through the workbook you would be able to see the full code of underlying examples, written in the IDE provided. When you are ready to proceed with an exercise, please click the “Start” button.

Section 2 consists of 8 additional exercises.

Exercises and further instructions are placed in a grey boxes throughout this workbook. Please click the “Finish” button when you have finished with an exercise.
Section 1

1.1 F# syntax

We will start with introduction to the F# programming language and how to use the Infer.NET framework. F# is a type-safe, type-inferred functional/imperative/ object-oriented programming language for the .NET framework. For the exercises in this workbook only a few basic things about the syntax are needed and most of them must be familiar from ML. All syntax necessary for the completion of the workbook is described below.

You should be currently able to examine the full code of the F# syntax examples below, in the software provided. Feel free to play with the example code as much as you want to, while reading the next subsection. When you are ready with viewing the example, click “Finish”.

• Declaring a variable/function - similarly to ML we don’t specify a type when declaring a variable. Instead, we use the keyword let:

<table>
<thead>
<tr>
<th>F#</th>
<th>ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>let n = 5</td>
<td>val n = 5</td>
</tr>
<tr>
<td>let a = 0.5</td>
<td>val a = 0.5</td>
</tr>
<tr>
<td>let str = &quot;Cambridge&quot;</td>
<td>val str = &quot;Cambridge&quot;</td>
</tr>
<tr>
<td>let f x = x*x</td>
<td>fun f x = x*x</td>
</tr>
</tbody>
</table>

Notice that unlike ML we use the same keyword for declaring variables and functions.

• Arrays - Although as a functional language F# is designed to work efficiently with lists, arrays are also easy to use - there is only a little syntactic difference compared to, for instance, Java. You can see how arrays can be used in F# in the following example:
```fsharp
define arr = [|5; 3; 2; 5|]
define thirdElem = arr.[2]
arr.[1] <- 10
```
```java
int[] arr = {5, 3, 2, 5};
int thirdElem = arr[2];
arr[1] = 10;
```

Note that similarly to ML, by default F# variables are immutable and the statement `a = b` is not an assignment, but returns a value `true` or `false`. The elements of an array are however mutable variables and can be changed using the operator `<-`.

- **For loops** - Another structure we need to introduce for this experiment is the “for each” loop of F#, which is similar to that of Java:

```fsharp
define L = arr.Length
for i in 0..(L-1) do
    arr.[i] <- i
```
```java
int L = arr.length
for(int i=0; i<L; i++)
    arr[i] = i;
```

- **Printing to the console** - Similar to `printf` in C/C++. You can use `printfn` to add a new line character at the end of your input:

```fsharp
//To print a string:
printf "%s" str

// To print an integer:
printf "%d" 5

// To print a fp number:
printfn "%f" 0.5

// To print an object
// using the default
// layout settings:
printfn "%A" arr
```
```c
printf("%s", str);
printf("%d", 5);
printf("%f\n", 0.5);
```

In other words, use "%A" when in doubt.
Exercise 0

1. Define a new array `arr` that consists of the integers 7, 2, 4.

2. Change the elements of `arr`, such that each is the original element squared.

3. Print the first element to the console.

Click “Start” to start the exercise and “Finish” when done.

1.2 Introduction to Probabilistic Programming

Suppose we have two coins - lets call them “coin1” and “coin2”. We can see the outcomes of tossing each coin once as two random variables, taking values in the space \{heads, tails\} with equal probabilities. That is each toss can result in “heads” with a probability of 1/2 and in “tails” with probability of 1/2.

Imagine we conduct an experiment in which Thomas Bayes (an English statistician, philosopher and Presbyterian minister) tosses coin1, tells you if coin1 is heads or tails and then tosses coin2. You have to guess what the outcome of coin2’s toss is. No matter what Thomas reports, the probability that coin2 will be heads is still 1/2. coin1 and coin2 are independent:

\[
P(coin2 = heads | coin1 = heads) = P(coin2 = heads) = \frac{1}{2}
\]

However, after a while, Thomas gets bored of the setting of the experiment and suggests changing it a bit. He will now report if it’s true that both tossing coin1 and coin2 resulted in heads. This can also be seen as a random variable, lets call it “bothHeads”, which is dependent on coin1 and coin2.

Now imagine he reported that it is not true that both coins are heads. How can you find the probability that coin1 is heads now, i.e.

\[
P(coin1 = heads | bothHeads = false)
\]

Thomas suggests using his famous rule in the following way:
\[ P(\text{coin1} = \text{heads}|\text{bothHeads} = \text{false}) = \]
\[ = \frac{P(\text{bothHeads} = \text{false} | \text{coin1} = \text{heads}) P(\text{coin1} = \text{heads})}{P(\text{bothHeads} = \text{false})} \]
\[ = \frac{(1/2)(1/2)}{3/4} = \frac{1}{3} \]

However, don’t get put off by the maths here - you won’t have to do any of it while completing the workbook. Probabilistic programming allows us to describe scenarios similar to that one. One can declare a random variable and introduce dependencies, similarly to the way that would happen with any other program variable. Moreover, one can observe values of certain variables and infer the updated, posterior probabilities of others.

For instance, in our two-coins example, we would declare the random variables \( \text{coin1} \), \( \text{coin2} \) and \( \text{bothHeads} \) and add appropriate dependencies between them. This we call a graphical model. We can then observe that \( \text{bothHeads} \) is \text{false} (as Thomas told us) and infer what \( \text{coin1} \) is.

### 1.3 Infer.NET

Infer.NET is a .NET library for probabilistic programming. It provides simple tools for building and running inference on graphical models. You will now learn about several ways of interacting with the library.

You can now proceed to “Example 1” and play with it while reading the associated explanation below.

- **Declaring a fresh random variable** – Using the \textit{let} key word and the Infer.NET class \texttt{Variable}. For example, imagine that we want to declare random variables \( \text{coin1} \) and \( \text{coin2} \) corresponding to the two coins of the example discussed above. We do the following:

  \[
  \text{let coin1} = \text{Variable.Bernoulli}(0.5) \\
  \text{let coin2} = \text{Variable.Bernoulli}(0.5) \\
  \text{// NB: X being Bernoulli(p) means Prob(X = true) = p}
  \]

- **Introducing a boolean dependency** – We now want to declare a random variable \( \text{bothHeads} \) that represents the probability that both \( \text{coin1} \) and \( \text{coin2} \) will be heads. The bitwise and (\&\&\&) operator of F\# is overloaded for Infer.NET variables, so we simply write:
let bothHeads = coin1 &&& coin2

We can use bitwise or (|||) and bitwise not (∼∼∼) in a similar way.

- **Observing a random variable** – Now suppose that similarly to before, our friend Thomas told us that it is not the case that both coins are heads. We can observe that bothHeads is false by simply writing:

  bothHeads.ObservedValue <- false

- **Inferring the distribution of a random variable** – We now want to see what distribution is, say, coin1 drawn from, giving us the probability of it being heads. To do that we firstly create an “Inference Engine”:

  let ie = new InferenceEngine()

  We then infer coin1’s distribution and print the result as follows:

  printfn "%A" (ie.Infer(coin1))

Please click “Finish” when you are ready viewing the example.

---

**Exercise 1**

This exercise introduces unfair coins (with probability of being heads different from 1/2) in a more complex version of the “two coins” example. There are several coins and intermediate variables. Modify the program as follows:

1. What is the distribution of oneAndTwo?
2. Change coin1 to be defined as Bernoulli(0.9)
3. Now what is the distribution of oneAndTwo?
4. What about that of oneAndThree?
5. Add a new random variable allHeads that represents the probability that all coins are heads. What distribution does it have?

Click “Start” to start the exercise and “Finish” when done.

---

You can now proceed to “Example 2” and play with it while reading the associated explanation below.
• **The “probabilistic” If** – Often, one would need to declare a variable’s distribution depending on what value another variable takes. For example, take the “rain-sprinkler-wet” example:

  – imagine we have a garden and we want to measure what the probability of the grass being **wet** is.
  – the grass can get wet either if it **rains** or the **sprinkler** is on.
  – however, the sprinkler is not very likely to be on if it rains.

Suppose we want to declare Rain and Sprinkler according to the following conditional probability tables:

| Rain   | \( P(Sprinkler = T | Rain) \) | \( P(Sprinkler = F | Rain) \) |
|--------|-------------------------------|-------------------------------|
| \( T \) | 0.01                          | 0.99                          |
| \( F \) | 0.4                           | 0.6                           |

We define Rain as usual and Sprinkler as an empty variable:

```csharp
let Rain = Variable.Bernoulli(0.2)
let Sprinkler = Variable.New<bool>()
```

We then use the “probabilistic If” of Infer.NET on Rain, to set the appropriate values for Sprinkler:

```csharp
begin
  // description of the FIRST ROW of the conditional probability table of Sprinkler
  use ifR = Variable.If(Rain)
  Sprinkler.SetTo(Variable.Bernoulli(0.01))
end

begin
  // description of the SECOND ROW of the conditional probability table of Sprinkler
  use ifNotR = Variable.IfNot(Rain)
  Sprinkler.SetTo(Variable.Bernoulli(0.4))
end
```

Here, the keywords `begin` and `end` are simply used to describe the scope in which the probabilistic If (or IfNot) is active. The keyword `use`
indicates that the object defined through it will be automatically disposed of when it is not needed anymore (so when we exit the begin-end in this case). However, you don’t need to worry about this too much – just remember that by using probabilistic If / IfNot, we can define random variables that are conditional on other Bernoulli random variables.

Please click “Finish” when you are ready viewing the example.

Exercise 2

Here you are presented with a more complete version of the “rain-sprinkler-wet”.

1. Please explain out loud what the changes to this example are.
2. With respect to the description of the problem on the previous page, what dependency is missing?

Click “Start” to start the exercise and “Finish” when done.

You can now proceed to “Example 3” and play with it while reading the associated explanation below.

• Introducing a continuous random variable – Apart of Bernoulli random variables that we’ve seen so far, Infer.NET supports many other discrete (e.g. Binomial, Poisson), continuous (Gaussian, Gamma, Beta) and multivariate (Dirichlet, Wishart) random variables.

We can, for example, declare a random variable $X$ drawn from a Normal(0,1) distribution (that is the mean of $X$ is 0, the variance is 1), in the following way:

```csharp
let X = Variable.GaussianFromMeanAndVariance(0.0, 1.0)
```

Notice that Infer.NET uses an alternative name for the Normal distribution – Gaussian distribution.

• Arrays - In real world application, we often have large arrays of data we want to analyse and learn from. Suppose, for example, that we have several (in our example – 3) real numbers stored in the array (`data`) that we assume are distributed normally and we want to find some sample mean and variance. We declare those as a Gaussian and a Gamma random variables respectively:
let meanIneff = Variable.GaussianFromMeanAndVariance(0.0, 1.0)
let varIneff = Variable.GammaFromShapeAndScale(1.0, 1.0)

Now we can define a Gaussian variable \( X \) with mean \( \text{meanIneff} \) and variance \( \text{varIneff} \):

\[
X = \text{Variable.GaussianFromMeanAndVariance}(\text{meanIneff}, \text{varIneff})
\]

In fact, we can use a for loop to declare 3 such variables \( X \) and observe that \( X.\text{ObservedValue} \leftarrow \text{data}[i] \) for all \( i \). However, this approach is inefficient and can be very messy in real-world applications where there are large numbers of massive arrays of data.

Infer.NET has been designed to work efficiently with large arrays. However, to exploit this capability, we need to use a \text{VariableArray} object rather than an array of \text{Variable} objects. In order to do this we proceed as follows:

1. Declare a \text{Range} variable that determines how big our variable array will be. In our case that's 3:
   \[
   \text{let dataRange} = \text{Range}(3)
   \]

2. Now we declare a variable array using this range as follows:
   \[
   \text{let Xs} = \text{Variable.Array<double>>(dataRange)}
   \]

3. We can access each variable in the array, say variable \( i \), by using its index: \( Xs.[i] \). Moreover, we can address the whole variable array object by indexing it with the range it has been declared with: \( Xs.[\text{dataRange}] \).

4. Finally, we need to initialise the variable array, by setting each of its elements to an actual variable. To initialise each element to be distributed normally with mean \( \text{meanEff} \) and variance \( \text{varEff} \), we use the \text{ForEach} construct, implicitly iterating through all elements:

\[
Xs.[\text{dataRange}] \leftarrow \\
\quad \text{Variable.GaussianFromMeanAndVariance(\text{meanEff}, \text{varEff})} \\
\quad \text{.ForEach(dataRange)}
\]

We can now observe the whole array by simply writing:

\[
Xs.\text{ObservedValue} \leftarrow \text{data}
\]

This will change the distributions of \( \text{meanEff} \) and \( \text{varEff} \), giving us the desired information about how \( \text{data} \) is distributed.
Please click “Finish” when you are ready viewing the example.

**Exercise 3**

Here you will be presented with some graphical model.

1. Please explain out loud what this model describes.
2. Is $x$ observed?
3. What about $y$?
4. What distribution does $\text{mean}$ have?
5. Observe $y$ to be data2. What distribution does $\text{mean}$ have now?

Click “Start” to start the exercise and “Finish” when done.
Section 2

Exercise 4
Here you will be presented with a graphical model that represents the problem described below. However, there are several mistakes in the code. Find and correct all mistakes.

After a regrettable incident involving an inflatable gorilla, a famous College has decided to install an alarm for the detection of roof climbers.

- The alarm is very good at detecting climbers.
- Unfortunately, it is also sometimes triggered when one of the extremely fat geese that lives in the College lands on the roof.
- One porter’s lodge is near the alarm, and inhabited by a chap with excellent hearing and a pathological hatred of roof climbers: he always reports an alarm. His hearing is so good that he sometimes thinks he hears, thus reports an alarm, even when there isn’t one.
- Another porter’s lodge is a good distance away and inhabited by an old chap with dodgy hearing who likes to listen to his collection of DEATH METAL with the sound turned up. Thus there is only 50% probability of him reporting an alarm that went off, and he would never mistakenly think it went off.

This example was taken from Dr Sean Holden’s lecture notes in AI II.

Exercise 5
You will be presented with some graphical model.

1. Please explain out loud what problem this model describes.
2. Are variables hasTorC and hasLungCancer independent?
3. What about hasBronchitis and hasDyspnea?
4. And hasBronchitis and hasTorC?
**Exercise 6**

The Faculty of Witchcraft and Wizardry at University of Cambridge has been very disorganised and lost all grades that graduating students got in different courses in the past year. However, wizards have excellent memory and remember all sorts of things (if they are not grades!) about people – for example they remember if they recommended a particular student for a secretary position at the Ministry of Magic.

To recover the grades they decide to use a muggle technique and construct a graphical model from which to learn.

1. By looking at their model, explain how they are trying to do that.

2. What dependencies do they assume that exist?

3. What is the probability of a student having a First Class grade on any course, assuming no other information about the student exists?

**Exercise 7**

After successfully recovering huge number of past exam results using muggle techniques, the Faculty of Witchcraft and Wizardry at University of Cambridge decided to apply these for other problems as well. For example, the professor in Defence Against Dark Arts wants to see if she has marked students “fairly” in the past 2 years. She assumes that marks are distributed normally and that if she has been fair, the following would hold:

- Mean values in 2013 and 2014 might differ, as one year questions might have been a bit harder than the other.
- The precision \( \frac{1}{\text{variance}} \) of the marks’ distributions in 2013 and 2014 would be the same.

However, if she hasn’t been fair, the marks for the two years would be drawn from normal distributions with completely different precisions.

1. Based on the value of \( \text{Fair} \), was the professor fair in her marking?

2. The professor is certain this result must be wrong. Help her correct her graphical model.

3. Now what is the probability of her being fair?
Exercise 8
Here you will be presented with a graphical model that represents the problem described below. However, there are several mistakes in the code. Find and correct all mistakes.

A company is trying to measure the probabilities of backache among its personnel.

- Someone’s back is likely to start aching if it is tense.
- People who do a lot of sports are more likely to have tension in their backs.
- All office chairs in the company were recently changed. Unfortunately half of those chairs had a defect and are very uncomfortable.
- A very common reason for back tension is using an uncomfortable chair.
- However back tension is not the only thing a bad chair could cause – it could also make the person using it extremely moody. There is a 10% chance for a person to be moody if they have a normal chair and 50% chance if their chair is uncomfortable.

Exercise 9
You will be presented with some graphical model.

1. Please explain out loud what problem this model describes.

2. Are variables HighNumberOfJourneys and HighNumberOfAccidents independent?

3. What about BadRoadConditions and HighNumberOfJourneys?

4. And HighSpeed and BadRoadConditions?
**Exercise 10**

You will be presented with some graphical model connected to the following problem:

The police is investigating a case of Alice being attacked late at night. She didn’t see her attacker, but Bob was held into custody shortly after the attack, as a prime suspect. Bob denies having committed the attack. To determine if Bob should be kept in custody and investigated further, the police uses takes some samples of the fibres on both Alice’s and Bob’s clothings.

1. By looking at their model, explain how they are trying to do that.

2. What dependencies do they assume that exist?

3. What is the probability of there being no contact between Alice and Bob, assuming no prior information about the fibres and Bob’s thoughts about Alice exist.

**Exercise 11**

To study if their new pill is effective, a drug production company performs the following experiment:

- A group of people is given that pill every day for a week. In the end of that week, the drug company determines if the week’s pills had (true) or didn’t have (false) an effect for each person and notes that down.

- Another group, a control group, goes through the same procedure, but instead of taking the company’s pill, they take a placebo pill (without knowing it).

- The company expects the results of the two groups to be drawn from the same distribution if the pill is NOT effective and different distributions if it is.

However, the model they’ve created doesn’t behave as desired.

1. Based on the value of `isEffective`, was the treatment efficient?

2. The company suspects there might be a few mistakes in their model. Help them out by correcting them.

3. Now what is the probability of the treatment being effective?
Appendix D

Questionnaires

The first questionnaire used in the user studies, *Programming experience self-assessment*, is based on previous work [29].

The second questionnaire is based on the 10-item *General Self-Efficacy Scale* [30]. Both questionnaires are attached overleaf.
Programming Experience Self Assessment

1. Approximately how long have you been programming?

2. What score would you give yourself on a scale of 1 to 10, with 1 being extremely inexperienced at programming and 10 being highly experienced at programming?

3. In which decile of your peer group would you place your programming skills? (i.e. top 10%, top 20%, etc)

4. For each programming language you use or have used, please answer the following questions as briefly as possible. If there are several, you may pick the ones you feel you’re best at. There is no compulsion to fill all 4 spaces below. Please fill in your responses in decreasing order of proficiency; put your best language first.
   a) What score would you give yourself from 1-10, with 10 being highly proficient at this language?
   b) How long have you been using this language?
   c) How often do you use it? (In any way you like, e.g. hours per week, times per month, etc)
   d) In brief, what is the largest/most complex program you have written in this language?

Language 1: ______________________ a) b) c) d)

Language 2: ______________________ a) b) c) d)

Language 3: ______________________ a) b) c) d)

Language 4: ______________________ a) b) c) d)
UserID:

Please answer the following as they relate to probabilistic programming:

<table>
<thead>
<tr>
<th></th>
<th>Statement</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I can always manage to solve difficult problems if I try hard enough.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>If someone opposes me, I can find the means and ways to get what I want.</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>It is easy for me to stick to my aims and accomplish my goals.</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>I am confident that I could deal efficiently with unexpected events.</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Thanks to my resourcefulness, I know how to handle unforeseen situations.</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>I can solve most problems if I invest the necessary effort.</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>I can remain calm when facing difficulties because I can rely on my coping abilities.</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>When I am confronted with a problem, I can usually find several solutions.</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>If I am in trouble, I can usually think of a solution.</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>I can usually handle whatever comes my way.</td>
<td></td>
</tr>
</tbody>
</table>

1 = Not at all true   2 = Hardly true   3 = Moderately true   4 = Exactly true

Comments:
Appendix E

Project Proposal

Part II Computer Science Project Proposal

Interactive Development Environment
for Probabilistic Programming

24.10.2014

Project Originators: Alan Blackwell and Advait Sarkar

Resources Required: See attached Project Resource Form

Project Supervisor: Advait Sarkar

Signature:

Director of Studies: Lawrence C Paulson

Signature:

Overseers: Stephen Clark and Pietro Lio

Signatures:
Introduction

The increasing number of applications of various machine learning methods today, leaves us with the challenge to create useful tools that allow us to apply and develop such techniques more easily. This challenge has been tackled, for instance, by designing and developing different probabilistic programming languages, which facilitate the creation of graphical models, such as Bayesian networks for solving many machine learning problems.

Such models can often become very big and complex, which could make retaining intuition about how the model looks like and what it does difficult. Bayesian networks have a natural visual representation that follows from their graph structure. Thus, it seems natural to tackle the problem by finding an effective way of visualisation of such graphical model and indeed research in the area shows that. Zapata-Rivera and Greer created the tool ViSMode [8] to help students to inspect and experiment with Bayesian what-if-scenarios of their student model. Interacting with the tool helped students not only to understand better their complex Bayesian student model but also motivated them to explore it further and learn more. Moreover, tools like Netica [2] and Bayesia [1] provide an intuitive interface that helps the user to build, edit and easily use Bayesian networks in various context. However, these seem to be created mainly for personal or business use, which restricts their functionality.

Infer.NET [6] extends the .NET framework by introducing tools for probabilistic programming, which can be invoked from any .NET language, including F#. It can be used to solve many different machine learning problems, including customized solutions for domain-specific problems. Although it provides some visualisation via creating a static image of the underlying Bayesian model after compilation, it doesn’t allow the programmer to view a live picture of this model while programming.

Recent products such as the Larch environment [4, 5] and the Swift language [3] illustrate what the next
level of abstracting programming might be. Inspired by those ideas, the aim of this project is creating an IDE which allows the programmer to work with the Infer.NET framework in an interactive way. The environment would support live visualization of the probabilistic graphical models being created. The programmer would be able to view a responsive image of the Bayesian Networks described by the source file as well as graphs of the probability density functions of any random variables.

Figure 3: The Larch Environment allows objects to define a “visual representation”, so they are displayed visually rather than textually. This representation can also be interactive, allowing the programmer to easily change appropriate values.
Substance and Structure of the Project

The project is divided into the following main sections:

1. **Background study**

   - Familiarisation with the main Bayesian networks concepts. Studying some standard problems in the area to get an idea of how the graphical model could look like and in what way inference is typically done.
   - Learning about the functionality of Infer.NET.
   - Familiarising with previous attempts of visualising Bayesian networks to identify successful and unsuccessful ideas and gain insight on the ways the visualisation of such network can be implemented.
   - Choosing a suitable graphic user interface package

2. **Design**

   Use different analytical methods to model the way the networks and their elements would be visualised.

   ![Figure 4: A very simple mockup of what the IDE could look like.](image)

3. **Building an environment for F#** Start working on the IDE by creating a simple window with all text fields, buttons and menus considered in the design. Embed
F# Interactive into the IDE, so one is able to write F# code and pass it to the interpreter.

4. Visualising the graphical model

Following Tanimoto’s [7] description of liveliness, this can be done in two stages, achieving level 2 and level 3 liveliness respectively.

(a) Level 2 liveness - an “informative and significant”.
- Parsing the code to identify different commands as creating a new variable, observing, inferring, etc.
- Building a visual model describing the code being compiled.

(b) Extending the above to achieve level 3 liveness - “informative, significant and responsive”. The visualisation should update instantly, as the code is being changed.

Possible Extensions

The following are a few objectives that are not critical to the success of the project, but would be useful extensions for an already working implementation:

1. Embed the visualisation of the density functions of the random variables into the network visualisation itself.

2. Implement various tricks to solve the scaling problem - e.g. show only some variables, “focus” only on the part at which code is being modified, reduce the rate of live updates, etc.

3. Add elements with which the user can interact directly - e.g. change a name of a variable, add, remove or change causal relationships, etc.

Starting Point

Familiarity with F# as I used it as the main programming language in a summer project.

No prior knowledge of probabilistic programming and Infer.NET. No experience with building graphic user interface apart of Java practicals in Part IA.
Success Criteria

For the project to be successful the following two should be designed and implemented:

- An Infer.NET IDE providing:
  - a live, edit-triggered visualisation of the Bayesian network described by the source code.
  - a live, edit-triggered visualisation of the probability density functions of any random variables declared.

Where:

- “live” means that a run-time visualisation of the program execution will be embed in the IDE itself.
- “edit-triggered” means that re-compilation of parts (or all) of the code will occur when a change to the code is made, without the user to have to explicitly invoke F#’s compiler.

- A suitable user study, designed so it eliminates order and difference-in-difficulty effects and studies the usability of the Infer.NET IDE.

Evaluation

The evaluation of the project will involve creating suitable probabilistic programming tasks for human participants. For instance, there will be a few or all of the following:

- Write-from-scratch task - the user will have to write a program in Infer.NET that solves a simple inference problem.

- Modification task - the user will have to modify an already working Infer.NET code to change what it does in a particular way.

- Debugging task - the user will have to find and fix bugs in an Infer.NET program, given instructions what the code should do prior to the task.

- Think-aloud task - the user will be asked to explain what they think some Infer.NET program does.

The participants will be asked to do one group of tasks with a plain text editor and another group of tasks with the final implementation of the Infer.NET IDE to be built. To eliminate any ordering effects some groups of participants will do plain text editor tasks
followed by IDE tasks, while some groups will do these the other way around. To eliminate any difference-in-difficulty effects, the same group of tasks will be performed both with a plain text editor and with the IDE in different groups of users. Thus, the participants will be split into several groups, allowing for the order of tasks to be counterbalanced across participants.

The evaluation will be finalised by comparing speed of finishing the tasks, number of key strokes, number of corrections, etc. within subjects. Participants will also be asked to fill in a short questionnaire asking them about their experience with the product.

Timetable and Milestones

Slot 1 - Michaelmas Term

- **Weeks 1 and 2** - Background reading on Bayesian networks and Infer.NET.
- **Weeks 3 and 4** - Familiarizing with other research in the area, noting any visualisation elements that were successful and any problems that occurred.
- **Weeks 5 and 6** - Working on the design. Planning and evaluating different design ideas.
  
  **Milestone:** Having a clean, concrete model design for the product.

Slot 2 - Christmas Vacation

- **Weeks 7 to 9** - Building up to “level 2 liveliness”.
  
  **Milestone:** Having a working implementation of an IDE with informative and significant visualisations.
- **Weeks 10 to 12** - Building up to “level 3 liveliness”.
  
  **Milestone:** Having a working implementation of an IDE with informative, significant and responsive visualisations.

Slot 3 - Lent Term

- **Weeks 13 and 14** - Catch up if fallen behind or implement an extension. Prepare progress report.
  
  **Milestone:** Submit progress report.
- **Week 15** - Prepare presentation. Start planning out dissertation.
  
  **Milestone:** Having a well-structured dissertation plan.
• **Weeks 16 to 18** - Design and build user study. Continue working on dissertation.  
  **Milestone:** Having a prepared user study, including any software and questionnaires needed. **Milestone:** Completed “Introduction” section of dissertation.

• **Weeks 19 and 20** - Conduct user study. Work on “Preparation” and “Implementation” sections of dissertation.  
  **Milestone:** Completed “Introduction” and “Preparation” sections of dissertation.

**Slot 4 - Easter Vacation**

• **Weeks 21 to 23** - Analyse user study results. Make changes regarding feedback from Project Supervisor to “Introduction” and “Preparation” sections. Finish “Implementation” section and work on “Evaluation” section.  
  **Milestone:** Completed “Introduction”, “Preparation”, “Implementation” and “Evaluation” sections of dissertation.

• **Weeks 24 to 26** - Analyse user study results. Make changes regarding feedback from Project Supervisor to “Introduction” and “Preparation” sections. Finish “Implementation” section and start working on “Evaluation” section.  
  **Milestone:** Completed “Introduction”, “Preparation”, “Implementation” and “Evaluation” sections of dissertation.

**Slot 5 - Easter Term**

• **Week 27** - Write “Conclusion” section. Finish any unfinished work on other sections.  
  **Milestone:** Finished draft dissertation.

• **Week 28 and 29** - Make changes regarding feedback from Project Supervisor.  
  **Milestone:** Hand in finished dissertation before 15 May 2015.
Resource Declaration

Hardware

- Personal laptop: code development and writing dissertation.
- MCS machine if laptop fails.
- Back-up storage: 70 GB external HDD.

Software

- Visual Studio 2013: coding environment.
- Infer.NET assembly files.
- F# Compiler Services assembly files.
- A graph layout framework (e.g. NetworkX or Graph#)
References


