Microsoft Research

# Infer.NET 101

## A sample-based introduction to the basics of Microsoft Infer.NET programming

Version 1.2 – October 2018


### Abstract

This document provides a detailed, sample-based introduction to the basics of Microsoft® Infer.NET programming.

For a review of the basic concepts of probabilistic programming and Infer.NET, see "An Introduction to Infer.NET."


### In this paper


Introduction
**Chapter 1** CyclingTime1 Application: Basic Parameter Learning
**Chapter 2** Digressions: Gaussian and Gamma Distributions, and Factor Graphs
**Chapter 3** CyclingTime2 Application: Restructuring CyclingTime1
**Chapter 4** Digression: Priors
**Chapter 5** CyclingTime3 Application: Mixture Models
**Chapter 6** Digression: Discrete Distributions and the Inference Engine
**Chapter 7** CyclingTime4 Application: Model Comparison
**Chapter 8** CyclingTime5 Application: A Model for Two Cyclists
**Chapter 9** Digression: Functions of Random Variables
Resources
Appendix


**Note:**

To provide feedback, send e-mail to infersup@microsoft.com.

# Content

Introduction ..... 4
Infer.NET Basics ..... 6
Create a Model ..... 6
Observe Random Variables ..... 6
Infer Posteriors ..... 7
Use the Posteriors ..... 7
What's Next? ..... 7
**Chapter 1** CyclingTime1 Application: Basic Parameter Learning ..... 9
CyclingTime1 Model ..... 9
CyclingTime1 Application ..... 11
[1] Create the Model ..... 13
[2] Train the Model ..... 14
[3] Use the Trained Model for Prediction ..... 15
**Chapter 2** Digressions: Gaussian and Gamma Distributions, and Factor Graphs ..... 17
Gaussian and Gamma Distributions ..... 17
Gaussian Distribution ..... 17
Gamma Distribution ..... 18
Factors and Factor Graphs ..... 19
**Chapter 3** CyclingTime2 Application: Restructuring CyclingTime1 ..... 22
CyclistBase Class ..... 23
CyclistBase Fields ..... 23
CreateModel Method ..... 24
SetModelData Method ..... 26
CyclistTraining Class ..... 26
CreateModel Method ..... 27
InferModelData Method ..... 28
CyclistPrediction Class ..... 29
CreateModel Method ..... 29
InferTomorrowsTime Method ..... 30
InferProbabilityTimeLessThan Method ..... 30
Use the Model ..... 30
Training ..... 30
Prediction ..... 31
Online Learning ..... 32
**Chapter 4** Digression: Priors ..... 34
How Much Does the Initial Prior Matter? ..... 34
Conjugate Priors ..... 34
**Chapter 5** CyclingTime3 Application: Mixture Models ..... 36
CyclistMixedBase Class ..... 37
CreateModel Method ..... 38
SetModelData Method ..... 39
CyclistMixedTraining Class ..... 40
InferModelData Method ..... 42
CyclistMixedPrediction Class ..... 42
CreateModel Method ..... 42
InferTomorrowsTime Method ..... 43

Use the Model ..... 43
**Chapter 6** Digression: Discrete Distributions and the Inference Engine ..... 46
Discrete Distributions ..... 46
Bernoulli Distribution ..... 46
Discrete Distribution ..... 48
How the Inference Engine Works ..... 48
**Chapter 7** CyclingTime4 Application: Model Comparison ..... 50
How to Compute Evidence with Infer.NET ..... 51
CyclistWithEvidence Class ..... 52
CyclistMixedWithEvidence Class ..... 53
Computing Evidence ..... 54
**Chapter 8** CyclingTime5 Application: A Model for Two Cyclists ..... 56
TwoCyclistsTraining Class ..... 56
CreateModel Method ..... 57
SetModelData Method ..... 57
InferModelData Method ..... 57
TwoCyclistsPrediction Class ..... 58
CreateModel Method ..... 58
SetModelData and InferTomorrowsTime Methods ..... 59
InferTimeDifference and InferCyclist1IsFaster Methods ..... 59
Use the Model ..... 60
**Chapter 9** Digression: Functions of Random Variables ..... 62
Resources ..... 64
Appendix ..... 66
A: Terminology ..... 66
B: System Requirements and Installation ..... 68
C: How to Build and Run Infer.NET Applications ..... 69

# Introduction

This document provides a detailed, sample-based introduction to writing probabilistic programs with Microsoft® Infer.NET. If you are unfamiliar with probabilistic programming, we recommend that you first read "An Introduction to Infer.NET," which is listed in Resources, for a discussion of the basic concepts of probabilistic programming and introduction to the Infer.NET platform.

The core of this document walks you through the code of a series of increasingly sophisticated Infer.NET applications. The walkthroughs describe how the applications use the core features of Infer.NET programming and include some discussion of the conceptual and mathematical underpinnings.

**Walkthrough Scenario.** All the applications discussed in this document are based on the following scenario:

- You and several co-workers bicycle to work every day.
- An individual cyclist's travel time varies randomly from day to day, so its value is uncertain.
- The travel time's uncertainty is represented by a probability distribution that defines the average travel time and how much it varies.
- The application will learn this distribution from several observed travel times and use that knowledge to make predictions about future travel times.

**Walkthrough Samples.** The walkthroughs cover the following sample applications:

- CyclingTime1 learns a single cyclist's travel time distribution, and uses that information to predict future travel times.
- CyclingTime2 is a restructured version of CyclingTime1, which introduces a standard practice for implementing models that is used in the subsequent applications.
- CyclingTime3 allows for the possibility of an unexpected event, and models travel time as a mixture of two distributions.
- CyclingTime4 shows how to use evidence to select the best model.
- CyclingTime5 adds a second cyclist, and describes how to construct more complex models.

**About the Samples.** The samples are all contained in a single Microsoft Visual Studio®-based solution, InferNET101, which can be found in the Examples folder of the Infer.NET source tree. The solution consists of the following:

- InferNet101.cs
  A simple console application that runs the five samples in order.
- RunCyclingSamples.cs
  A set of five static methods named RunCyclingTime*N*, one for each sample. The methods contain the corresponding sample's core Infer.NET code, which creates and trains the model, makes predictions, and so on.
- CyclingTime*N*, where *N* ranges from 2 to 5
  The Infer.NET model for CyclingTime1 is implemented in RunCyclingSamples.cs. The final four samples implement their models as separate classes, which are in the corresponding CyclingTime*N*.cs file.

For general directions on how to build and run Infer.NET applications, see Appendix C.

**Prerequisites.** This document assumes that you are familiar with the basic concepts and terminology of probabilistic programming, as described in "An Introduction to Infer.NET", which is listed in "Resources" later in this document.

You should also have at least the following:

- Basic C# programming skills.

  This prerequisite is especially important because otherwise you might have difficulty distinguishing between standard C# code and Infer.NET-specific code.
- Familiarity with using Microsoft Visual Studio to program Microsoft .NET Framework applications with C#.
- Familiarity with basic statistics.

  Familiarity with Bayesian inference is helpful but not required.

**Terminology.** For definitions of terminology that is specific to Infer.NET and is used in this document, see Appendix A.

**Installation.** For system requirements and installation directions see Appendix B.

# Infer.NET Basics

To help you get oriented before starting on the first application, this section briefly describes the basic structure and key elements of the CyclingTime*N* samples. The terms and concepts are discussed in more detail as the document proceeds.

## Create a Model

An Infer.NET application is built around a probabilistic model, which defines the random variables and how they are related.

A random variable is essentially an extension of a standard type such as **double** or **int**, which allows the type to have uncertain values. Infer.NET represents random variables as instances of the **Variable<T>** class, which is in the **Microsoft.ML.Probabilistic.Models** namespace. **T** is called the variable's *domain type*:

- Discrete random variables have a specified set of possible values and a **bool** or **int** domain type.
- Continuous random variables have a range of possible values and a **double** domain type.

A random variable is *defined* by a *probability distribution*—commonly abbreviated to *distribution*—which assigns probabilities to the variable's possible values. Initially, a random variable is defined by a *prior distribution*—commonly abbreviated to *prior*—which represents your understanding of the variable's value before making any observations.

**Note:** Creating a random variable can involve as many as three separate steps. For example:

1. C# declaration: `Variable<bool> x;`
2. C# definition: `x = Variable<bool>.New();`
3. Statistical or model definition: `x = Variable.Random<bool>(someDist);`

In the language of statistical inference, "definition" or "define" refers to the variable's distribution, and that is how the term is generally used in this paper. However, in cases where Steps 2 and 3 are separate statements, we clarify the distinction by using "statistical definition" or "statistically define" for Step 3.

There are a variety of ways to define relationships between random variables, which are discussed later in this document.

## Observe Random Variables

You can perform computations on the model at this point, but they usually aren't very interesting. The results merely reflect the values you chose for the priors. To learn something new, you observe one or more of the model's random variables by assigning values to their **ObservedValue** properties. At this point, the variables are no longer random; they are effectively standard types with a fixed value.

## Infer Posteriors

Before you make any observations, the model's priors directly or indirectly define a particular random variable's distribution. After making some observations on one or more other variables, you know more about that variable's value so it has a new distribution called the *posterior distribution*—commonly abbreviated to *posterior*. A posterior incorporates the information from the prior and the observations, and represents your new and presumably improved knowledge of the variable's value.

You compute posteriors by using an instance of the Infer.NET inference engine, which performs all the numerical heavy lifting. The inference engine is implemented as the **InferenceEngine** class in the **Microsoft.ML.Probabilistic.Models** namespace.

The inference engine computes the posterior distribution for a specified random variable by "summing out" the effect of the model's other random variables. In general, this type of distribution is called the variable's *marginal distribution*, which is commonly abbreviated to *marginal*. When you query the inference engine for a random variable's marginal after observing one or more of the model's other random variables, the marginal that the engine returns is a posterior—an update of the variable's prior, conditioned by the new information from the observations. For a more detailed discussion of marginals, priors, and posteriors, see "An Introduction to Infer.NET."

## Use the Posteriors

You can use posteriors for a variety of purposes. One obvious use is to predict the future behaviour of the variable. However, with only a few observations, the posterior might not accurately reflect the variable's real behaviour and the predictions might not be very accurate. You can improve your understanding by making additional observations, and incorporating that information into the variable's distribution, as follows:

1. Use the posterior as the variable's new prior.
2. Make some additional observations.
3. Compute a new posterior.

The new posterior incorporates the initial prior and all observations. You can continue this process indefinitely. After a sufficient number of observations, the posterior should better reflect the variable's real value.

# What's Next?

A distribution is controlled by one or more parameters. Sometimes you can make reasonable *a priori* estimates of the parameter values, but they rarely match subsequent observations exactly. Sometimes you have little or no prior knowledge. In other words, the parameter values are uncertain.

From a Bayesian perspective everything, including distribution parameters, is uncertain and can be treated as a random variable. You assign a prior to the variable

that reflects your initial understanding—or lack of understanding—of the parameter's value and use observations and probabilistic programming to learn distributions over the value. This general approach is a form of *parameter learning*, which is introduced by CyclingTime1 and is used by all the walkthroughs in this example.

**Chapter 1**

# CyclingTime1 Application: Basic Parameter Learning

The section demonstrates the basics of how to use Infer.NET to implement parameter learning. CyclingTime1 is a simple application that learns an individual cyclist's travel time distribution based on three days of observed travel times. This process is sometimes referred to as "training the model." CyclingTime1 then uses the trained model to predict tomorrow's travel time.

## CyclingTime1 Model

A cyclist's travel time varies from day to day with the individual times distributed about some average value. You can represent these travel times statistically as a set of random samples from a distribution that matches the data. CyclingTime1 uses a Gaussian distribution to represent the travel time distribution, and characterizes the distribution by the following two parameters.

- The distribution's mean is the cyclist's average travel time.
- The distribution's precision determines how much the travel time varies from one day to the next, and reflects factors such as the day to day variations in traffic conditions.

  It is convenient to think of precision as a measure of the traffic "noise" with greater precision corresponding to less day-to-day variation.

**Note:** You might be more familiar with the standard deviation, σ, as a measure of the width of a Gaussian distribution. It is often more mathematically convenient to use one of the following related values:

- Variance: $\sigma^2$
- Precision: $1/\sigma^2$

For a more detailed discussion, see "Gaussian Distribution" in Chapter 2, later in this document.

Because the mean and precision are both uncertain and continuous, CyclingTime1 represents them by **double** random variables.

- *averageTime* represents the mean, and its uncertainty is represented by another Gaussian distribution.
- *trafficNoise* represents the precision and its uncertainty is represented by a Gamma distribution.

The Gaussian and Gamma distributions and the reasons for using them are discussed later.

CyclingTime1 represents the travel times that are observed by using three random variables, *travelTimeMonday*, *travelTimeTuesday*, and *travelTimeWednesday*. All three variables are assumed to be drawn from the same Gaussian travel time distribution.

The final piece of the puzzle is to define the relationship between *averageTime*, *trafficNoise,* and the three *travelTimeX* random variables. Probabilistic programming typically uses a *generative model* to define the relationships between the random variables. A generative model describes how the observed data is generated from the underlying model parameters. Although it is sometimes helpful to think of a generative model as a set of cause-effect relationships, generative models are not necessarily causal.

For CyclingTime1, the generative process is:

1. Sample the *averageTime* and *trafficNoise* random variables to generate mean and precision values for this cyclist.
2. Create a Gaussian distribution over travel times by using the mean and precision values from Step 1.
3. Sample the Gaussian distribution from Step 2 to generate the travel time value *travelTimeMonday*.
4. Repeat step 3 for *travelTimeTuesday* and *travelTimeWednesday*.

Figure 1 shows a graphical representation of the CyclingTime1 model, called a *factor graph*. Factor graphs are discussed in more detail later, but briefly:

- Ellipses indicate random variables.
- Shaded ellipses indicate observed variables.
- Filled squares indicate factors, which represent the associated variable's distribution.
- Arrows show the direction of the generative process.

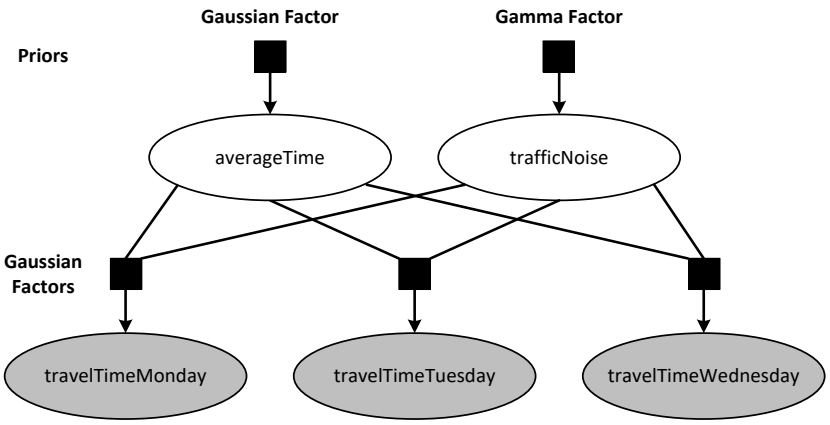


**Figure 1. CyclingTime1 factor graph**

Although the generative process has a well-defined direction, inference can work in any direction. You observe one or more random variables, and the inference engine uses the observations and the model to infer the distributions of the remaining random variables. In Figure 1, the training data represented by the three *travelTimesXYZ* variables is observed, and the inference engine uses the observations and model to infer *averageTime* and *trafficNoise*. This is opposite to the model's generative direction, so in this case inference propagates information backwards from the observations.

There actually two versions of the CyclingTime1 model, which correspond to whether we want to learn the parameters or to predict tomorrow's time. Figure 1 shows the model for training—that is, learning the parameters. The prediction model has the same structure as the training model but uses the computed *averageTime* and *trafficNoise* values to infer *tomorrowsTime*. In this case, the inference engine propagates information from the parameters forwards.

You will see later how these two tasks can be implemented with a single model–the only difference being which variables are inferred. However, to make the code more explicitly correspond to the two tasks we use an explicit '*tomorrowsTime*' prediction variable in the prediction model.

# CyclingTime1 Application

The following example shows the complete CyclingTime1 source, which is annotated for later reference. The **using** statements are for namespaces that used by most Infer.NET applications. It is implemented as the *RunCyclingTime1* static method in RunCyclingSamples.cs.

**Listing 1: CyclingTime1**

```
using System;
using Microsoft.ML.Probabilistic.Models;
using Microsoft.ML.Probabilistic.Distributions;

public static void RunCyclingTime1()
{
  //[1] The model
```

```
  Variable<double> averageTime = Variable.GaussianFromMeanAndPrecision(15, 0.01);
  Variable<double> trafficNoise = Variable.GammaFromShapeAndScale(2.0, 0.5);

  Variable<double> travelTimeMonday =
              Variable.GaussianFromMeanAndPrecision(averageTime, trafficNoise);
  Variable<double> travelTimeTuesday =
              Variable.GaussianFromMeanAndPrecision(averageTime, trafficNoise);
  Variable<double> travelTimeWednesday =
              Variable.GaussianFromMeanAndPrecision(averageTime, trafficNoise);

  //[2] Train the model
  travelTimeMonday.ObservedValue = 13;
  travelTimeTuesday.ObservedValue = 17;
  travelTimeWednesday.ObservedValue = 16;

  InferenceEngine engine = new InferenceEngine();

  Gaussian averageTimePosterior = engine.Infer<Gaussian>(averageTime);
  Gamma trafficNoisePosterior = engine.Infer<Gamma>(trafficNoise);

  Console.WriteLine("averageTimePosterior: " + averageTimePosterior);
  Console.WriteLine("trafficNoisePosterior: " + trafficNoisePosterior);

  //[3] Make predictions

  //Add a prediction variable and retrain the model
  Variable<double> tomorrowsTime =
             Variable.GaussianFromMeanAndPrecision(averageTime, trafficNoise);

  Gaussian tomorrowsTimeDist = engine.Infer<Gaussian>(tomorrowsTime);
  double tomorrowsMean = tomorrowsTimeDist.GetMean();
  double tomorrowsStdDev = Math.Sqrt(tomorrowsTimeDist.GetVariance());

  // Write out the results.
  Console.WriteLine("Tomorrows predicted time: {0:f2} plus or minus {1:f2}",
                    tomorrowsMean, tomorrowsStdDev);

  // Ask other questions of the model
  double probTripTakesLessThan18Minutes = engine.Infer<Bernoulli>(tomorrowsTime <
18.0).GetProbTrue();
  Console.WriteLine("Probability that the trip takes less than 18 min: {0:f2}",
                    probTripTakesLessThan18Minutes);
}
```

The following sections describe how CyclingTimes1 is implemented and are keyed to the numbered comments.

**Note:** The syntax of Infer.NET code can sometimes appear complex. Although it is useful to think of Infer.NET as a language for probabilistic programming, the examples are C# code that calls into the Infer.NET application programming interface (API). The advantage of an API is that you can mix probabilistic and non-probabilistic code, which allows you to embed Infer.NET directly into applications. As an API, Infer.NET is restricted to using the constructs that are provided by C# and other high-level .NET languages and makes heavy use of .NET generics. When you are learning Infer.NET, it is important to focus on the underlying principles, and not get distracted by the syntax.

# [1] Create the Model

The initial part of CyclingTime1 defines the training model.

## Create *averageTime* and *TrafficNoise*, and Specify Initial Priors

The first step in the training phase is to create random variables to represent *averageTime* and *trafficNoise*. Both variables have a continuous range of possible values, so they are represented by **double** random variables. Before running inference, you must specify the variables' priors.

A prior represents your best understanding of a random variable before observing one or more of the model's other random variables. After making the observations, you can use the prior, the model, and the observations to compute the variable's posterior, which represents your improved understanding of the variable's behaviour. You can then use the posterior to predict the variable's future behaviour, or as the prior for the next set of observations, and so on.

One obvious issue is what to use for the initial prior's parameter values. In many cases, you can assume that distribution parameters are based on a general understanding of the variable's behaviour. For example, an experienced cyclist should be able to estimate how long a particular trip should take and how much the time will vary from day to day. Such an estimate is probably not exact, but it's likely to be fairly close and you specify a precision value that appropriately represents the uncertainty. The initial prior is an assumption, which might seem a bit fuzzy, but keep in mind that all forms of statistical analysis make assumptions at some level. With Bayesian inference, assumptions are explicit, and they are handled in a principled way from that point on.

To specify an initial prior, define the associated random variable by using the appropriate distribution with a reasonable set of parameter values. The simplest way to handle this task is to use one or more of the **Variable** class's static factory methods. If the variable is scalar—as opposed to an array of variables—you can typically create the random variable and provide the statistical definition with a single method call.

The following example shows how CyclingTime1 uses factory methods to create *averageTime* and *trafficNoise*, and give them their initial priors.

```
Variable<double> averageTime = Variable.GaussianFromMeanAndPrecision(15, 0.01);
Variable<double> trafficNoise = Variable.GammaFromShapeAndScale(2.0, 0.5);
```

CyclingTime1 uses the following parameter values for the *averageTime* initial prior:

- The mean is set to 15, based on general cycling experience.

  A cyclist usually has a fairly good idea how long a particular route will take. The specified mean reflects that informed understanding and represents a valid if somewhat uncertain understanding of the variable.

- The precision is set to 0.01.

  The mean is basically an educated guess, and the precision value represents how precise the cyclist believes that guess to be. Here, the small precision indicates a large initial uncertainty.

The *trafficNoise* parameters are set to similarly reasonable values. The reasons for choosing these particular distributions and the meaning of the *trafficNoise* parameters are discussed later in this document.

As a practical matter, the choice of an initial prior is significant only if you have a small number of observations. As the number of observations increases, the influence of the initial prior on the posterior declines. With enough observations—often a relatively low number—the posterior distribution is effectively determined by the observations. In fact, applications often just use a "neutral" initial prior—such as assigning equal probabilities to all possible values—and let the observations determine the correct distribution.

### Create and Define the Travel Times

Each observed value is represented by a random variable. Each travel time must therefore be represented by a **double** random variable—a **Variable<double>** object—that is governed by a continuous distribution. For this scenario, a Gaussian distribution is the obvious choice.

The final step in creating the training model is to define the three variables that are to be observed:

```
Variable<double> travelTimeMonday =
        Variable.GaussianFromMeanAndPrecision(averageTime, trafficNoise);
Variable<double> travelTimeTuesday =
        Variable.GaussianFromMeanAndPrecision(averageTime, trafficNoise);
Variable<double> travelTimeWednesday =
        Variable.GaussianFromMeanAndPrecision(averageTime, trafficNoise);
```

Defining the three variables by using *averageTime* and *trafficNoise* as parameters provides the link between the variables that is shown Figure 1.

## [2] Train the Model

To complete the training process, CyclingTime1 observes the training data and infers posteriors for *averageTime and trafficNoise*. The posteriors essentially summarize the results of the training process in a form that can be used for further computations, such as predicting future travel times, or as priors for additional training.

You observe data by assigning a value to the random variable's **ObservedValue** property. CyclingTime1 observes the training data as follows:

```
travelTimeMonday.ObservedValue = 13;
travelTimeTuesday.ObservedValue = 17;
travelTimeWednesday.ObservedValue = 16;
```

At this point, the values of the three travel time variables are fixed, and the variables are no longer random.

All the details are in place now, but so far, Infer.NET has simply used the modelling code to construct an internal representation of the model. No computation takes place until you query the inference engine for a random variable's marginal.

To compute posteriors, CyclingTime1 creates an instance of the inference engine and queries it for the *averageTime* and *trafficNoise* marginals. To query for a random

variable's marginal, you pass the variable to the **InferenceEngine.Infer<T>** method, where **T** is the variable's distribution type.

When you call **Infer<T>**, the inference engine:

1. Uses a model compiler to generate C# code, based on the model and the inference algorithm, to compute the requested marginal.
2. Uses the .NET C# compiler to compile the generated code to an executable.
3. Runs the executable to compute the requested marginal.
4. Returns the marginal to the application as a distribution object.

If you made observations before running the query, the engine uses the observations, the prior, and the model to infer a marginal for the specified random variable that is consistent with the training data and the prior. The returned marginals are therefore posteriors, by definition.

CyclingTime1 queries for the *travelTimes* marginals as follows:

```
InferenceEngine engine = new InferenceEngine();
Gaussian averageTimePosterior = engine.Infer<Gaussian>(averageTime);
Gamma trafficNoisePosterior = engine.Infer<Gamma>(trafficNoise);
```

In this case, the inference engine uses the default inference algorithm, which is expectation propagation. You can also specify other algorithms, as discussed later.

The inference engine returns the inferred posteriors**,** *averageTimePosterior* and *trafficNoisePosterior*, and CyclingTime1 prints the results, as follows:

```
averageTimePosterior: Gaussian(15.33, 1.32)
trafficNoisePosterior: Gamma(2.242, 0.2445)[mean=0.5482]
```

If you just consider the means of these two posterior distributions, the mean of the average travel time is 15.33 and the mean of the traffic noise is 0.5482. However both the average travel time and the traffic noise have uncertainty, as reflected by the full distributions. The uncertainty in these estimates will decrease with more observations.

## [3] Use the Trained Model for Prediction

The first step in predicting a travel time from the trained model is to define a prediction model. CyclingTime1 predicts a single travel time, which can be represented by a single ordinary random variable, *tomorrowsTime,* as follows:

```
Variable<double> tomorrowsTime =
    Variable.GaussianFromMeanAndPrecision(averageTime, trafficNoise);
```

To connect *tomorrowsTime* to the trained model, CyclingTime1 statistically defines *tomorrowsTime* by using a Gaussian distribution with:

- The distribution's mean parameter set to *averageTime*.
- The distribution's precision parameter set to the *trafficNoise*.

The prediction model is effectively the training model with an additional random variable.

Predicting tomorrows travel time is straightforward: query the inference engine for the *tomorrowsTime* marginal.

```
Gaussian tomorrowsTimeDist = engine.Infer<Gaussian>(tomorrowsTime);
double tomorrowsMean = tomorrowsTimeDist.GetMean();
double tomorrowsStdDev = Math.Sqrt(tomorrowsTimeDist.GetVariance());

Console.WriteLine("Tomorrows predicted time: {0:f2} plus or minus {1:f2}",
                  tomorrowsMean, tomorrowsStdDev);
double probTripTakesLessThan18Minutes =
            engine.Infer<Bernoulli>(tomorrowsTime < 18.0).GetProbTrue();
Console.WriteLine("Probability that the trip takes less than 18 min: {0:f2}",
                  probTripTakesLessThan18Minutes);
```

When you query for the *tomorrowsTime* marginal, the inference engine:

1. Compiles the prediction model.

   Adding *tomorrowsTime* changes the original training model, so the model must be compiled before running the computations.

2. Uses the observed values for *travelTimeMonday*, *travelTimeTuesday*, and *travelTimeWednesday* to the train the model.

   *averageTime* and *trafficNoise* are now defined by their posteriors.

3. Uses the trained model to compute the *tomorrowsTime* posterior.

This procedure performs the entire computation from the beginning, including the computations that were performed earlier for the training model. Chapter 3 introduces a more efficient way to handle this process.

The remaining code shows how to use the Gaussian object's methods plus some utility methods to obtain various types of information, as follows:

- **Gaussian.GetMean** returns the distribution's mean.
- **Gaussian.GetVariance** returns the distribution's variance.

  CyclingTime1 converts the variance to the equivalent standard deviation, which is a little easier to visualize.

The results are as follows:

```
Tomorrows predicted time: 15.33 plus or minus 2.15
```

The *tomorrowsTime* prediction is a distribution, not just a number, so you can use it to ask some relatively sophisticated questions. The final line queries the model for the probability that tomorrow's trip takes less than 18 minutes.

The results are as follows:

```
Probability that the trip takes less than 18 min: 0.89
```

**Chapter 2**

# Digressions: Gaussian and Gamma Distributions, and Factor Graphs

## Gaussian and Gamma Distributions

CyclingTime1 uses two continuous distributions: the Gaussian and Gamma distributions. This section provides some details.

### Gaussian Distribution

The Gaussian distribution—sometimes referred to as a normal distribution—is a continuous distribution that defines a classic bell curve. The mathematical definition of a Gaussian distribution for variable *x* is as follows:

$$\mathcal{N}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left\{\frac{1}{2\sigma^2}(x-\mu)^2\right\}$$

Where:

- *x* represents a continuous range of values from which a sample of the distribution can be drawn.

  Formally, x ranges from [-∞, ∞], but as a practical matter $\mathcal{N}$ is effectively zero except near its mean value (x = μ), so Gaussians are often used to represent variables with more limited ranges.

- $\mathcal{N}$ is the probability density for a given value of *x*.

  The probability density is the relative probability that a sample will be drawn for a particular value of x. If point A's probability density is twice point B's probability density, then point A is twice as likely to be sampled.

The distribution is governed by two parameters:

- μ is the mean value of x, and is also the maximum value of $\mathcal{N}$.
- σ is a measure of the distribution's width known as the "standard deviation."

Instead of σ, width is often expressed in terms of two related parameters, which are typically more mathematically convenient:

- The variance, which is $\sigma^2$.

- The precision, which is $1/\sigma^2$.

The exponential function's prefix is a normalization constant, which ensures that the integral of the distribution over its range is 1.0.

Figure 2 shows a graph of a Gaussian for typical values of μ and σ.

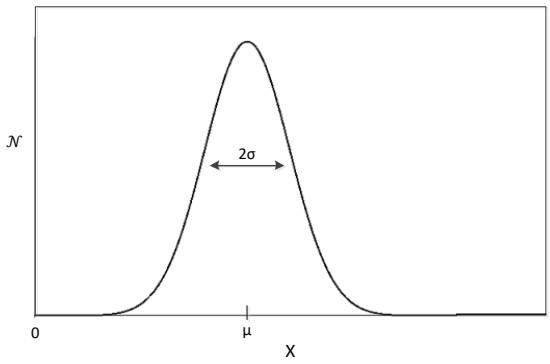


**Figure 2. Gaussian distribution**

For more information about Gaussian distributions, see "Gaussian Distribution" or "Pattern Recognition and Machine Learning" in "Resources."

## Gamma Distribution

The Gamma distribution can be used for any continuous random variable that has a range of [0, ∞]. With probabilistic programming, the Gamma distribution is most commonly used to define random variables such as *trafficNoise* that represent a Gaussian distribution's precision parameter.

The mathematical definition of a Gamma distribution for variable *x* is as follows:

$$P(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k \Gamma(k)}$$

Where:

- *x* represents a continuous range of values in the range [0, ∞] from which a sample of the distribution can be drawn.
- $P$ is the probability density for a given value of *x*.
- $\Gamma$ is a gamma function. If k is an integer, $\Gamma(k) = (k\text{-}1)!$ .

The distribution is governed by two parameters, both of which are positive:

- k is called the *shape* parameter.
- θ is called the *scale* parameter.

The Gamma distribution can also be defined by *shape* (α) and *rate* (β) parameters, where α = k and β = 1/θ.

Figure 3 shows a Gamma distribution for some representative shape and scale values.

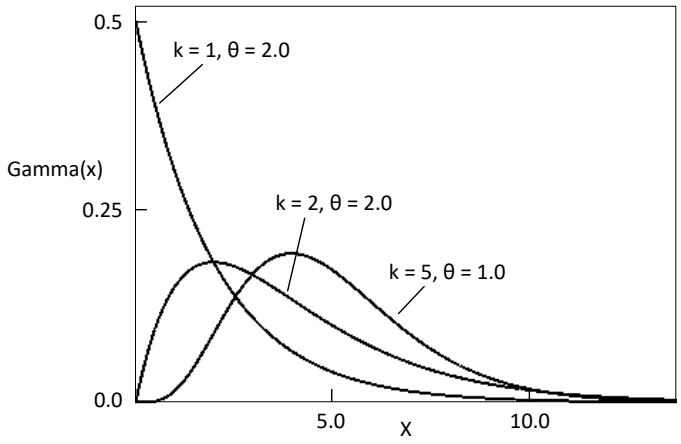


**Figure 3. Gamma distribution**

For more details, see "Gamma Distribution" in "Resources."

# Factors and Factor Graphs

Probabilistic programs are typically built around a generative model that describes the relationships between the system's random variables and so defines the joint probability distribution over these variables. For more discussion of joint probability distributions, see "An Introduction to Infer.NET."

The richest and most flexible ways to represent models are mathematical or programmatic, and that is the only way that some models can be expressed. However, many models can be represented graphically, typically as a factor graph or as a related graph called a *Bayesian network*. Graphs provide a visual representation of the dependency structure among the variables, and can be useful for constructing and understanding probabilistic models and implementing the associated code.

For more discussion of graphical models, see Chapter 8 of "Pattern Recognition and Machine Learning."

This document uses factor graphs, which represent the factorization of the model's joint probability distribution; the model is the product of all the factors. Factor graphs are a natural way to represent complex conditional relationships between variables and are straightforward to represent with Infer.NET code.

Graphically, factor graphs consist of three node types:

- Open circles or ovals to represent random variables.
- Filled circles to represent observed variables.
- Filled squares to represent factors.

By convention, generation starts at the top of the graph and proceeds downward. The nodes are linked by *edges*, which are represented by lines. To show the generative direction the edges that originate from the bottom of a factor are arrows that point to the associated random variable node. Factors also usually have one or

more parameters, such as a Gaussian factor's mean and precision values. The random variables that represent the parameters are connected by edges to the factor's top.

A *stochastic factor* such as a Gaussian, represents the distribution of the random variable that the edge points to, conditioned by the factor's input variables. Other types of factors serve other purposes:

- Deterministic factors represent operations such as summing two existing random variables to produce a new random variable.
- Constraint factors constrain existing random variables in various ways, such as requiring a variable to take only positive values.
- Unary factors do not have edges connected to their tops and typically represent distribution factors with fixed parameter values.

For more discussion of factors and how they are related to distributions, see "Factors and Constraints" in "Resources."

Figure 1 is actually a somewhat simplified version of the CyclingTime1 training graph. Figure 4 shows the complete graph, which explicitly shows the unary factors that represent the *averageTime* and *trafficNoise* priors.

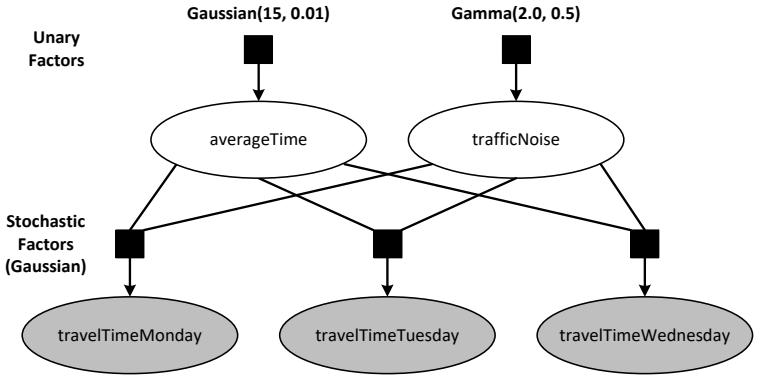


**Figure 4. Complete CyclingTime1 training graph**

Figure 4 contains the following variables and factors:

- The two variables at the top of the graph, *averageTime* and *trafficNoise*, are random variables that represent the Gaussian distribution's mean and precision parameters.

  The associated Gaussian and Gamma unary factors represent the *averageTime* and *trafficNoise* variables' priors, respectively.

- The three variable at the bottom of the graph are the observed travel time variables, which represent the cyclist's travel time data.

  The Gaussian factors in the centre of the graph represent the distribution that defines the three travel time variables. The factor produces a Gaussian distribution based on *averageTime* and *trafficNoise*.

You can construct much more complex and sophisticated factor graphs, especially for systems with many random variables, and you can map them fairly directly to equivalent Infer.NET code. One of prime benefits of Infer.NET is that it provides

applications with a straightforward way to construct complex and sophisticated models from simple building blocks. The inference engine then handles the task of performing inference computations on the complex model.

For more discussion of factor graphs, see Chapter 9, “Digression: Functions of Random Variables,” later in this document.

**Chapter 3**

# CyclingTime2 Application: Restructuring CyclingTime1

CyclingTime1 demonstrated the basics of parameter learning. Although the application's simple structure is useful for introducing basic concepts, it can't be readily extended to handle more sophisticated scenarios. In particular, CyclingTime1 implemented everything in a single method, which is not ideal for production-level applications.

The preferred approach—which is used by CyclingTime2—is to encapsulate the modelling code in a separate class. In some cases, it is helpful to implement separate classes for the training and prediction models. In others, you can use a single class, but the training and prediction models should be separate class instances, so that the inference engine does not have to repeatedly recompile the model.

CyclingTime2 extends CyclingTime1 as follows:

- It encapsulates the training and prediction models in separate classes.

  *RunCyclingTime2* then uses instances of these classes to train the model and implement online learning.

- It introduces the use of random variable arrays for observed variables.

  Random variable arrays are much more efficient than using a separate **Variable<T>** object for each observed value.

- It introduces a different way to handle priors that is useful for more sophisticated learning models.

  A simple example is given at the end of this chapter.

The CyclingTime2 classes are implemented in CyclingTime2.cs. *RunCyclingTime2* is implemented as a static method in RunCyclingSamples.cs. Figure 5 shows the factor graph for the underlying generative model.

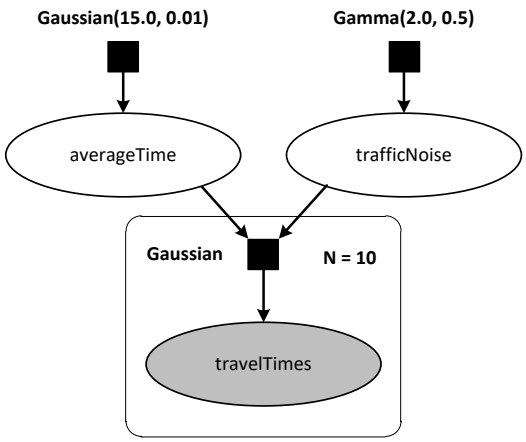


**Figure 5. CyclingTime2 factor graph**

This graph is essentially similar to Figure 1. However, CyclingTime1 used a separate **Variable<T>** object for each observed value, each of which had its own node. This approach is not practical for more than a few values. CyclingTime2 uses an array of random variables, *travelTimes*, to represent the 10 observed values. The rectangle around the *travelTimes* node is called a plate, and is basically a compact way to replicate model structure in a graph.

**Note:** For brevity, this section contains only edited excerpts of the key parts of CyclingTime2.

# CyclistBase Class

CyclingTime2 implements the training and prediction models as separate classes. However, training and prediction models typically share at least some code. To avoid implementing the same code twice, the common parts of the CyclingTime2 models are implemented in a base class, *CyclistBase*. The two model classes then inherit from *CyclistBase*, and implement the code that is specific to training or prediction. The class structure is more complicated than strictly necessary for CyclingTime2, but it is useful for the more sophisticated applications described later in this document.

## CyclistBase Fields

*CyclistBase* has the following basic structure.

```
public class CyclistBase
{
  public InferenceEngine InferenceEngine;

  protected Variable<double> AverageTime;
  protected Variable<double> TrafficNoise;
  protected Variable<Gaussian> AverageTimePrior;
  protected Variable<Gamma> TrafficNoisePrior;

  public virtual void CreateModel() {...}

  public virtual void SetModelData(ModelData priors) {...}
```

```
}
```

The fields serve the following purposes:

- *AverageTime* and *TrafficNoise* are random variables that serve the same purpose as in CyclingTime1.
- *AverageTimePrior* and *TrafficNoisePrior* are distributions that represent the priors for the random variables.
- *InferenceEngine* represents an instance of the inference engine.

*AverageTimePrior* and *TrafficNoisePrior* are distributions, but they are typed as **Variable<T>** objects over their respective distribution types rather than as distribution types. Distributions are not random variables. However, the **Variable<T>** type is not used only for random variables; it can also be used for values that you want to be able to change at run-time without requiring the inference engine to recompile the model. Using **Variable<T>** to hold the *averageTime* and *trafficNoise* priors is more computationally efficient and supports a more flexible approach to handling priors. For more details, see the following section, "CreateModel Method."

Most of the variables are protected because they are used only by the two derived classes. The exception is *InferenceEngine*. By default, each instance of the model has its own instance of the engine. However applications that use multiple instances of the model—such as applications that handle multiple cyclists—could end up with a large number of engine instances, even though a single instance is usually sufficient. Making *InferenceEngine* public allows a parent application to assign the same inference engine instance to each model, and use it for all computations.

*CyclistBase* has two instance methods. *RunCyclingTime2* calls these methods in the following order, after creating an instance of the training or prediction classes:

1. *RunCyclingTime2* calls *CreateModel* to create the training or prediction model.
2. *RunCyclingTime2* calls *SetModelData* to specify the model's priors.

The training and prediction models implement additional methods to run various queries, based on the model and the observed values.

## CreateModel Method

*RunCyclingTime2* calls *CreateModel* to create the model. *CyclistBase.CreateModel* implements the modelling code that is common to both training and prediction as follows:

```
public virtual void CreateModel()
{
  AverageTimePrior = Variable.New<Gaussian>();
  TrafficNoisePrior = Variable.New<Gamma>();
  AverageTime = Variable.Random<double, Gaussian>(AverageTimePrior);
  TrafficNoise = Variable.Random<double, Gamma>(TrafficNoisePrior);
  if (InferenceEngine == null)
  {
    InferenceEngine = new InferenceEngine();
  }
}
```

The *CyclistTraining* and *CyclistPrediction* implementations override *CreateModel* and add code as appropriate for their specific model.

CyclingTime1 took the simplest approach to handling priors: use a factory method to create and define the random variable. For example, CyclingTime1 assigns a fixed prior to *averageTime* as follows:

```
Variable<double> averageTime = Variable.GaussianFromMeanAndPrecision(15, 0.01);
```

CyclingTime2 uses a more sophisticated and flexible approach to handling priors. The priors are represented by **Variable<T>** objects, which are basically containers for the distributions. For example, the *AverageTime* prior is represented by a **Variable<Gaussian>** random variable, *AverageTimePrior*.

*CreateModel* creates *AverageTimePrior* by using the static **Variable.New<T>** method, which is a static factory method that creates a new **Variable<T>** object with a specified domain type but no statistical definition. This method is useful if you want to create a variable object and statistically define it later, or simply assign a value to the variable's **ObservedValue** property.

*CreateModel* then statistically defines *AverageTime* by calling the static **Variable.Random** method. **Random** is a unary factor which defines a random variable by using a specified distribution object or a **Variable<T>** object that represents a distribution object. CyclingTime2 packages the prior as a **Variable<T>** object, so it uses the **Random<T1, T2>** overload, where **T1** specifies the *AverageTime* variable's domain type and **T2** specifies its distribution type. In this example, **Random<T1, T2>**creates a **double** random variable and statistically defines it by using the Gaussian distribution that is represented by *AverageTimePrior*.

*TrafficNoise* is created and defined in the same way.

At this point, the actual prior distribution has not been specified. That can be done later—at any time before querying the inference engine—by assigning an appropriate **Gaussian** object to the *AverageTimePrior* variable's **ObservedValue** property. To specify another prior as the variable's distribution, just assign the distribution object to **ObservedValue**.

This approach has two advantages over the one used by CyclingTime1:

- It simplifies implementing learning models with multiple sets of observations.

  Instead of creating a new variable for each set of observations, CyclingTime2 uses *AverageTime* throughout the application, and updates the prior by assigning each new posterior to the *AverageTimePrior* variable's **ObservedValue** property.

- It improves performance.

  If you change the model, the inference engine must recompile it before running a query, which can take a significant amount of time. However, assigning a new value to an **ObservedValue** property doesn't change the model, even if the value is a reference type rather than a value type. For all queries after the first one, the engine just runs the existing compiled model with new observed values.

Finally, *CreateModel* creates an instance of the inference engine, if one hasn't already been assigned by the parent application.

## SetModelData Method

*CreateModel* defines the model, but doesn't assign any values to the model data. After creating the model, *RunCyclingTime2* calls *SetModelData* to specify the model data. For CyclingTime2, the model data consists of the priors, but other applications might use different data. The base class's *SetModelData* method assigns the specified distributions to the *AverageTimePrior* and *TrafficNoisePrior* variables' **ObservedValue** properties. The method is **virtual**, which allows derived classes to override it and specify different data, as required.

```
public virtual void SetModelData(ModelData priors)
{
  AverageTimePrior.ObservedValue = priors.AverageTimeDist;
  TrafficNoisePrior.ObservedValue = priors.TrafficNoiseDist;
}
```

For convenience, the model data is packaged in a private *ModelData* structure.

```
public struct ModelData
{
  public Gaussian AverageTimeDist;
  public Gamma TrafficNoiseDist;

  public ModelData(Gaussian mean, Gamma precision)
  {
    AverageTimeDist = mean;
    TrafficNoiseDist = precision;
  }
}
```

# CyclistTraining Class

The *CyclistTraining* class inherits from *CyclistBase* and implements the training model. The following example shows the class structure:

```
public class CyclistTraining : CyclistBase
{
  protected VariableArray<double> TravelTimes;
  protected Variable<int> NumTrips;

  public override void CreateModel()
  {...  }

  public ModelData InferModelData(double[] trainingData)
  {...}
}
```

The fields represent the following:

- *TravelTimes* represents an array of training data.

  The data array is in the form of a **VariableArray<T>** object, which is discussed later in "CreateModel Method."

- *NumTrips* represents the length of the training data array.

The reason for making *NumTrips* a **Variable<T>** object is explained later in "CreateModel Method."

The fields are **protected** rather than **private**, so that derived classes can access them.

*CyclistTraining* has two methods:

- *CreateModel* overrides the base method and implements the training-specific parts of the model.
- *InferModelData* takes the training data, runs the queries, and returns the posteriors.

*CyclistTraining* has no model data apart from the *AverageTime* and *TrafficNoise* priors, so it does not override *SetModelData*.

## CreateModel Method

RunCyclingTime2 calls *CyclistTraining.CreateModel* to create the training model, as follows:

```
public override void CreateModel()
{
  base.CreateModel();
  NumTrips = Variable.New<int>();
  Range tripRange = new Range(NumTrips);
  TravelTimes = Variable.Array<double>(tripRange);
  using (Variable.ForEach(tripRange))
  {
    TravelTimes[tripRange] =
           Variable.GaussianFromMeanAndPrecision(AverageTime, TrafficNoise);
  }
}
```

*CreateModel* calls the base method to create the common parts of the model and then implements the training-specific modelling code.

*NumTrips* is a **Variable<int>** type that represents the number of elements in the training data array. *CreateModel* uses **New** to create the *NumTrips* variable, and then uses *NumTrips* to initialize *tripRange*. *NumTrips* doesn't have a value at this point; it is specified later by assigning a value to the **ObservedValue** property.

Because CyclingTime2 has ten observations, it uses an array of random variables to represent the observed values instead of separate **Variable<double>** objects. You could package the set of objects as a .NET array, but the inference engine doesn't handle such arrays efficiently. A better approach is to use a **VariableArray<T>** object, which represents an array of random variables in a way that can be handled efficiently by the inference engine.

Some explanatory details:

- **VariableArray<T>** is an indexed type, so it can be used much like a standard .NET array.

  However, you must use a **Microsoft.ML.Probabilistic.Models.Range** object as the index instead of the usual integer.

- The **Range** object is initialised with the array length.

  In this case, the array length is determined by *NumTrips*. The *NumTrips* value hasn't been specified at this point, but you can initialize **Range** in this way as long as you specify the *NumTrips* value before querying the inference engine.

- To create a **VariableArray<T>** object, call the static factory method **Variable.Array<T>**, and pass it the associated **Range** object.

  **Variable.Array<T>** creates a **VariableArray<T>** object, but the variables are not statistically defined. That task must be handled separately.

**Note:** Although *NumTrips* is a **Variable<int>** object, it actually represents a well-defined value—the number of array elements. Making it a **Variable<int>** instead of an **int** allows CyclingTime2 to efficiently handle multiple training sessions with data arrays of different lengths. If *NumTrips* were an **int**, CyclingTime2 would have to create a new **Range** object each time the array length changes, which would change the model and force the model compiler to recompile it. By using a **Variable<int>** object, the only change from one session to the next is the *NumTrips* **ObservedValue** property, and recompilation isn't necessary.

The final step in creating the training model is to statistically define the elements of the *travelTimes* array. Because *travelTimes* is not a standard array, you can't use **for** or **foreach** to step through the array. Instead, Infer.NET provides a static **Variable.ForEach** method which serves the same purpose. **ForEach** takes a specified **Range** object and iterates over the range. The **using** statement (a standard C# construct) defines the **ForEach** block's scope, and can contain an arbitrary number of statements.

## InferModelData Method

*RunCyclingTime2* calls *InferModelData* to infer the *AverageTime* and *TrafficNoise* posteriors based on the priors that are specified earlier in *SetModelData* and on a training data array.

```
public ModelData InferModelData(double[] trainingData)
{
  ModelData posteriors;

  NumTrips.ObservedValue = trainingData.Length;
  TravelTimes.ObservedValue = trainingData;
  posteriors.AverageTimeDist = InferenceEngine.Infer<Gaussian>(AverageTime);
  posteriors.TrafficNoiseDist = InferenceEngine.Infer<Gamma>(TrafficNoise);
  return posteriors;
}
```

*InferModelData* assigns the length of the training data array to the *NumTrips* variable's **ObservedValue** property. It then assigns the training data to the *TravelTimes* array's **ObservedValue** property and calls the inference engine to infer the posteriors. *InferModelData* then returns the posteriors, packaged as a *ModelData* structure.

# CyclistPrediction Class

The *CyclistPrediction* class implements the prediction model. It inherits from *CyclistBase* and has the following class structure:

```
public class CyclistPrediction: CyclistBase
{
  private Gaussian tomorrowsTimeDist;
  public Variable<double> TomorrowsTime;

  public override void CreateModel() {...}

  public Gaussian InferTomorrowsTime() {...}

  public Bernoulli InferProbabilityTimeLessThan(double time)
  {...}
}
```

The fields represent the following:

- *tomorrowsTimeDist* represents the predicted travel time distribution.
- *TomorrowsTime* is a random variable that represents the predicted travel time.

  This field could be private for the purposes of CyclingTime2. However, a public field is useful for constructing models that involve more than one cyclist, as discussed later in Chapter 8, “CyclingTime5 Application: A Model for Two Cyclists.”

*CyclistPrediction* has three methods:

- *CreateModel* overrides the base method and implements the prediction-specific parts of the model.
- *InferTomorrowsTime* queries the inference engine for tomorrow’s predicted time distribution, based on the trained model.
- *InferProbabilityTimeLessThan* infers the probability that tomorrow’s time is less than the specified value.

*CyclistPrediction* handles the model data in the same way as *CyclistTraining*, so it does not override *SetModelData*.

## CreateModel Method

*RunCyclingTime2* calls *CyclistPrediction.CreateModel* to create the prediction model.

```
public override void CreateModel()
{
  base.CreateModel();
  TomorrowsTime =
     Variable.GaussianFromMeanAndPrecision(AverageTime, TrafficNoise);
}
```

*CreateModel* calls the base method to create the common parts of the model. It then implements the prediction-specific modelling code, which is identical to the code from CyclingTime1.

## InferTomorrowsTime Method

*RunCyclingTime2* calls *SetModelData* to specify the prediction model's priors. It then calls *InferTomorrowsTime* to obtain the predicted distribution for tomorrow's travel time. These priors are presumably the posteriors that were obtained from the most recent training session.

```
public Gaussian InferTomorrowsTime()
{
  tomorrowsTimeDist = InferenceEngine.Infer<Gaussian>(TomorrowsTime);
  return tomorrowsTimeDist;
}
```

*InferTomorrowsTime* infers the marginal based on the priors that *RunCyclingTime2* specified when it called *SetModelData*, and then returns the distribution to the caller.

The inference computation in *InferTomorrowsTime* is more efficient than the corresponding computation in CyclingTime1, which had to retrain the model from the beginning before computing tomorrow's predicted time. CyclingTime2 computes *tomorrowsTimeDist* by using the posteriors from the training model as priors. These distributions essentially represent the training results, so there is no need to repeat the training computation.

## InferProbabilityTimeLessThan Method

*InferProbabilityTimeLessThan* infers the probability that tomorrows travel time is less than a specified time.

```
public Bernoulli InferProbabilityTimeLessThan(double time)
{
  return  InferenceEngine.Infer<Bernoulli>(TomorrowsTime < time);
}
```

This code performs the same task as the corresponding code in CyclingTime1. *InferProbabilityTimeLessThan* returns a Bernoulli distribution, which is characterized by a single parameter that specifies the probability that the variable is **true**. In this case, **true** corresponds to the probability that tomorrows travel time is less than the specified value. The Bernoulli distribution is discussed in detail in "Discrete Distributions" in Chapter 6, later in this document.

# Use the Model

This section discusses how *RunCyclingTime2* uses the classes discussed in the preceding sections to train the model and predict tomorrow's travel time.

## Training

The first part of *RunCyclingTime2* creates and trains the model, as shown in the following excerpt.

**Listing 2. Training the Model**

```
double[] trainingData = new double[] { 13, 17, 16, 12, 13, 12, 14, 18, 16, 16 };
ModelData initPriors = new ModelData(
```

```
    Gaussian.FromMeanAndPrecision(1.0, 0.01),
    Gamma.FromShapeAndScale(2.0, 0.5));

//Train the model
CyclistTraining cyclistTraining = new CyclistTraining();
cyclistTraining.CreateModel();
cyclistTraining.SetModelData(initPriors);

ModelData posteriors1 = cyclistTraining.InferModelData(trainingData);

... //Print the training results
```

*RunCyclingTime2* defines initial priors for *AverageTime* and *TrafficNoise*, and packages them as a *ModelData* structure.

*RunCyclingTime2* creates a new *CyclistTraining* object to represent the training model. It calls *CreateModel* to create the training model and passes the initial priors to *SetModelData*. Finally, *RunCyclingTime2* passes the training data to *InferModelData*, which returns the posteriors.

The results are:

```
Average travel time = Gaussian(14.65, 0.4459)
Traffic noise = Gamma(5.33, 0.05399)[mean=0.2878]
```

## Prediction

*RunCyclingTime2* next uses the posteriors from the first training session to predict tomorrow's results.

**Listing 3. Predicting results**

```
CyclistPrediction cyclistPrediction = new CyclistPrediction();
cyclistPrediction.CreateModel();
cyclistPrediction.SetModelData(posteriors1);

Gaussian tomorrowsTimeDist = cyclistPrediction.InferTomorrowsTime();

double tomorrowsMean = tomorrowsTimeDist.GetMean();
double tomorrowsStdDev = Math.Sqrt(tomorrowsTimeDist.GetVariance());

Console.WriteLine("Tomorrows average time: {0:f2}", tomorrowsMean);
Console.WriteLine("Tomorrows standard deviation: {0:f2}", tomorrowsStdDev);
Console.WriteLine("Probability that tomorrow's time is < 18 min: {0}",
                  cyclistPrediction.InferProbabilityTimeLessThan(18.0));
```

The procedure is similar to the training phase. *RunCyclingTime2* creates a new *CyclistPrediction* object to represent the training model. It calls *CreateModel* to create the training model and passes the posteriors from the training phase to *SetModelData*. Finally, *RunCyclingTime2* calls *InferTomorrowsTime* and *InferProbabilityTimeLessThan* to obtain the predicted results, as follows:

```
Tomorrow's average time: 14.65
Tomorrow's standard deviation: 2.17
Probability that tomorrow's time is < 18 min: Bernoulli(0.9383)
```

## Online Learning

CyclingTime1 is limited to a single training phase and a single set of predictions. Probabilistic programs often use a process known as *online learning*, which allows an application to learn model parameters rapidly based on some initial data, and then continue to learn incrementally as more data comes in. The basic process is:

1. Start with initial priors that are typically chosen to have very broad distributions.
2. Collect some data and compute posteriors.
3. Update the model's priors to reflect the new data by replacing them with posteriors from Step 2.
4. Use the updated model to run predictions.
5. Repeat Steps 2 to 4 as many times as required, perhaps indefinitely.

As you continue the cycle of collecting data and updating priors, the model parameters should approach the actual values, and the model's predictive ability should improve. In addition, if external conditions change, online training allows the model to adapt to those changes.

It is important to note that if you are prepared to keep the data around, you can always retrain your model from scratch using the original priors and all the data; this is the fundamental difference between offline learning and online learning.

So far, CyclingTime2 is just a more sophisticated and efficient version of CyclingTime1, with some additional training data. However, the approach used by CyclingTime2 is much more flexible. This section shows how to use the *CyclistTraining* and *CyclistPrediction* classes to implement online learning.

CyclingTime2 has already completed one training phase and used the results for prediction. By implementing a second training and prediction phase, you can incrementally improve the accuracy of the *AverageTime* and *TrafficNoise* distributions. The basic approach can be easily extended to handle as many additional training sessions as required.

The following example shows how RunCyclingTime2 implements the second training session.

```
double[] trainingData2 = new double[] { 17, 19, 18, 21, 15 };

cyclistTraining.SetModelData(posteriors1);
ModelData posteriors2 = cyclistTraining.InferModelData(trainingData2);

//Print results

cyclistPrediction.SetModelData(posteriors2);

tomorrowsTimeDist = cyclistPrediction.InferTomorrowsTime();
tomorrowsMean = tomorrowsTimeDist.GetMean();
tomorrowsStdDev = Math.Sqrt(tomorrowsTimeDist.GetVariance());

//Print results
```

The second training data set is for five days. *CyclistTraining* can handle data sets of arbitrary length, so a training session can span any convenient time window, even daily.

An instance of *CyclistTraining* already exists, and the model was created during the first training session, so subsequent training sessions require only two lines of code:

1. Call *CyclistTraining.SetModelData* to update the training model's priors with the posteriors from the previous training session.
2. Call *CyclistTraining.InferModelData* to infer new posteriors.

The new posteriors now incorporate the initial priors and training data plus additional information from the second set of training data. As a practical matter, the influence of the initial priors is negligible by now, and the posteriors reflect the training data.

*RunCyclingTime2* calls *CyclistPrediction.SetModelData* to update the prediction model's priors with the new posteriors and calls the two *CyclistPrediction* inference methods, to predict the next day's travel time.

This procedure demonstrates the advantage of using **Variable<T>** objects to represent priors. When you update the priors, *SetModelData* simply assigns the specified distributions to the *AverageTimePrior* and *TrafficNoisePrior* variables' **ObservedValue** properties. Because only **ObservedValue** properties have changed between the first and second phases, the training and prediction models are unchanged and don't have to be recompiled. The inference engine can simply run the compiled models from the first phase with the new training data and priors.

The results for the second training and prediction phase are:

```
Average travel time = Gaussian(15.61, 0.3409)
Traffic noise = Gamma(7.244, 0.02498)[mean=0.1809]
Tomorrows average time: 15.61
Tomorrows standard deviation: 2.60
Probability that tomorrow's time is < 18 min: Bernoulli(0.8213)
```

**Chapter 4**

# Digression: Priors

## How Much Does the Initial Prior Matter?

A posterior reflects the influence of both the prior and the observations. How many observations does it take before the influence of the prior on the trained distribution is negligible? What if the prior doesn't match the observations well at all?

The answers to these questions depend on the particular model and prior, but you can get some idea by playing with *averageTime* prior's parameters. CyclingTime2 uses a prior with a nominal average time of 1.0, which is not a good estimate of the value of 15.61 that was inferred after training. If you change the prior's average time to 15.0 (a much more accurate prior), the trained value is 15.65, almost indistinguishable from the value when the prior was set to 1.0. For CyclingTime2 at least, fifteen observations are enough to effectively override the influence of the initial prior.

## Conjugate Priors

Why do CyclingTime1 and CyclingTime2 use a Gamma prior for *TrafficNoise*? Why not just use a Gaussian for both parameters? More generally, if you use a random variable to represent a distribution parameter, which distribution should you use to define that random variable?

For a factor representing a distribution, if a distribution parameter's posterior is from the same distribution family as the parameter's prior—and the other parameters are fixed—then the prior is referred to as a conjugate prior. Conjugate priors can simplify the process of computing posteriors—and some inference calculations require them—so conjugate priors are the natural way to represent a parameter's uncertainty for computational purposes.

For the CyclingTime1 example, the parent distribution is a Gaussian which has two parameters. Each of the parameters requires a different conjugate prior:

- The distribution's mean is represented by a **double** random variable that is defined by another Gaussian distribution.

- The distribution's precision is represented by a **double** random variable that is defined by a Gamma distribution.

Other ways of defining a Gaussian require different conjugate priors, and other distributions have their own conjugate priors. For example, the conjugate prior for the probability vector of a Discrete distribution is the Dirichlet distribution. For more information, see "Conjugate prior" in "Resources."

**Chapter 5**

# CyclingTime3 Application: Mixture Models

The scenario for CyclingTime1 and CyclingTime2 assumed that the distribution of travel times could be represented by a single Gaussian distribution. This is a common assumption, but data isn't necessarily that cooperative.

For example, any cyclist knows that you occasionally encounter extraordinary circumstances, such as a flat tire or a road closed by construction, which add a substantial amount to your usual travel time. A single Gaussian might be a reasonable model for the ordinary trips, but the data for the preceding scenario might look more like the thick line in Figure 6.

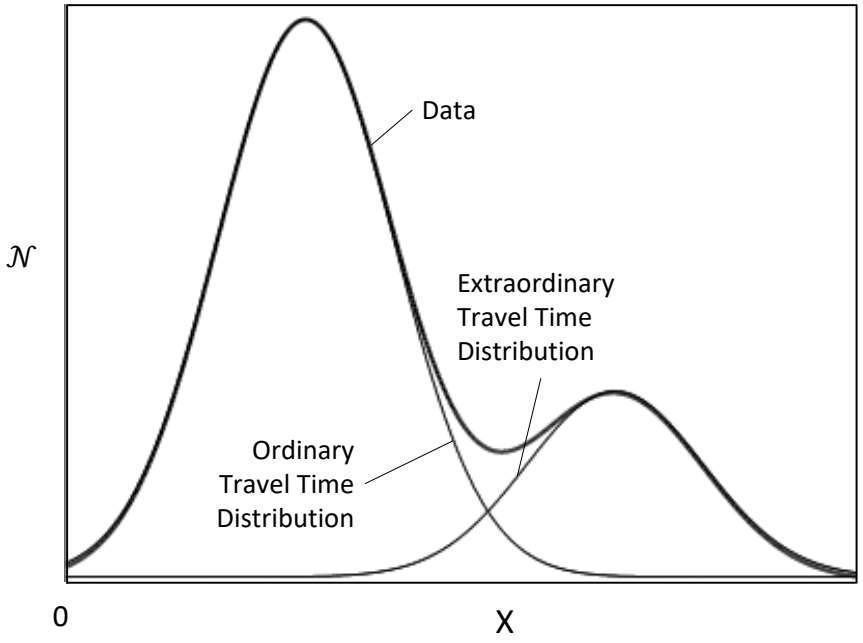


**Figure 6. Bimodal travel time distribution**

The effect of the extraordinary travel times in this example is a data set with two peaks—one corresponding to normal travel, and one corresponding to abnormal travel. Representing this data set by a single Gaussian merges the effects of both types of travel into a single relatively simple model, which might not be a very accurate. Another, and often better approach, is to represent such data as a mixture

of several Gaussians. Mixture models have the further advantage of being less sensitive to data outliers than a single distribution.

To specify such a model, you don't need to know the relative frequency of each type of travel but you must specify the number of Gaussians in the mixture. In this case, two Gaussians is the obvious choice, but the optimum number isn't always that apparent. As discussed in Chapter 7, Bayesian inference provides a way to determine the optimum number of Gaussians in the mixture by using *model evidence*.

The CyclingTime3 application extends the CyclingTime2 models to handle extraordinary events, as follows:

- Most trips are ordinary, with a mean travel time of approximately 15 - 16 minutes.
- Five to ten per cent are extraordinary, and require an additional ten to fifteen minutes.

CyclingTime3 handles this scenario by representing the system as a mixture of two Gaussians. For convenience, the Gaussian that corresponds to the ordinary travel times is called the ordinary component and the Gaussian that corresponds to the extraordinary travel times is called the extraordinary component.

CyclingTime3 is generally similar to CyclingTime2, as follows:

- The training and prediction models are implemented by the *CyclistMixedTraining* and *CyclistMixedPrediction* classes.
- *CyclistMixed*Training and *CyclistMixedPrediction* inherit from a base class, *CyclistMixedBase*, which implements the common parts of the models.
- The basic class structure is identical to the corresponding classes from CyclingTime2.

  The differences are limited to the details of the modelling code.
- A *RunCyclingTime3* static method creates and trains the training model and then uses the results and the prediction model to predict tomorrow's travel time.

For brevity, CyclingTime3 uses only a single training and prediction phase, but can easily be extended to multiple phases.

The CyclingTime3 classes are implemented in CyclingTime3.cs. *RunCyclingTime3* is implemented as a static method in RunCyclingSamples.cs.

# CyclistMixedBase Class

*CyclistMixedBase* has the following fields, most of which are random variables that are shared by the training and prediction classes.

**Listing 4.: CyclistMixed global variables**

```
protected InferenceEngine InferenceEngine;

protected int NumComponents;

protected VariableArray<Gaussian> AverageTimePriors;
```

```
protected VariableArray<double> AverageTime;

protected VariableArray<Gamma> TrafficNoisePriors;
protected VariableArray<double> TrafficNoise;

protected Variable<Dirichlet> MixingPrior;
protected Variable<Vector> MixingCoefficients;
```

The first four random variables serve the same purpose as the corresponding variables in CyclingTime2. However, a mixture model requires separate *AverageTime* and *TrafficNoise* parameters for each mixture component, so there are two of each. For computational convenience, the two *AverageTime* and *TrafficNoise* values are packaged as variable arrays, as are the corresponding priors. *MixingPrior* and *MixingCoefficients* are related to the mixture model, and are discussed later.

## CreateModel Method

*CyclistMixedBase.CreateModel* creates and defines the variables that are common to both models. It is similar in principle to *CyclistBase.CreateModel*, but the details are a bit more complicated.

```
public virtual void CreateModel()
{
  InferenceEngine = new InferenceEngine(new VariationalMessagePassing());

  NumComponents = 2;
  Range ComponentRange = new Range(NumComponents);

  AverageTimePriors = Variable.Array<Gaussian>(ComponentRange);
  TrafficNoisePriors = Variable.Array<Gamma>(ComponentRange);
  AverageTime = Variable.Array<double>(ComponentRange);
  TrafficNoise = Variable.Array<double>(ComponentRange);

  using (Variable.ForEach(ComponentRange))
  {
    AverageTime[ComponentRange] =
             Variable.Random<double, Gaussian>(AverageTimePriors[ComponentRange]);
    TrafficNoise[ComponentRange] =
             Variable.Random<double, Gamma>(TrafficNoisePriors[ComponentRange]);
  }

  //Mixing coefficients
  MixingPrior = Variable.New<Dirichlet>();
  MixingCoefficients = Variable<Vector>.Random(MixingPrior);
  MixingCoefficients.SetValueRange(ComponentRange);
}
```

The inference engine supports several inference algorithms. The default algorithm, which is used by the previous examples, is expectation propagation. *CreateModel* instead specifies the variational message-passing algorithm for the inference engine. Variational message passing is often a better choice for mixture models, and using expectation propagation for CyclingTime3 causes numerical problems in the inference computation, giving rise to *improper messages*. For more discussion of inference algorithms and improper messages, see “How the Inference Engine Works” later in this document.

The **Range** object, *ComponentRange,* is initialized to the number of components (2). It defines the range for several component-related **VariableArray<T>** objects.

For convenience the *AverageTime* and *TrafficNoise* random variables and their priors are packaged as two-element arrays. *CyclistMixedBase* then defines *AverageTime* and *TrafficNoise* by using **Variable.ForEach** to iterate over the two mixture components and define each element of *AverageTime* and *TrafficNoise* with the appropriate prior.

The final set of random variables is required to define the mixture itself. Mixture components typically do not have equal influence on the overall distribution, so the relative proportion of each component in the mixture is defined by a pair of *mixing coefficients*, each of which specifies the proportion of one of the components.

*CyclistMixedBase* represents the mixing coefficients by a **Vector** random variable, *MixingCoefficients*. The **Vector** type—defined by Infer.NET in the **Microsoft.ML.Probabilistic.Math** namespace—contains a set of **double** values, two in this case. *MixingCoefficients* is defined by a **Dirichlet** prior, *MixingPrior,* which is represented by a **Variable<T>** object so that it can be assigned a value later.

Briefly, a Dirichlet distribution is a distribution over a probability vector, which is a vector whose elements add up to 1.0. For example, a sample from such a distribution of dimension two, such as the one used by CyclingTime3, might be (0.25, 0.75). For more discussion of Dirichlet distributions see “Discrete Distributions” in Chapter 6.

The final line calls **Variable.SetValueRange** to explicitly specify the *MixingCoefficients* range, because the Infer.NET model compiler cannot always deduce the range of values that an integer random variable can take.

## SetModelData Method

*RunCyclingTime3* calls *SetModelData* to specify priors for *AverageTime*, *TrafficNoise*, and *MixingCoefficients*.

```
public virtual void SetModelData(ModelDataMixed modelData)
{
  AverageTimePriors.ObservedValue = modelData.AverageTimeDist;
  TrafficNoisePriors.ObservedValue = modelData.TrafficNoiseDist;
  MixingPrior.ObservedValue = modelData.MixingDist;
}
```

For convenience, the model data is packaged as a *ModelDataMixed* structure.

```
public struct ModelDataMixed
{
  public Gaussian[] AverageTimeDist;
  public Gamma[] TrafficNoiseDist;
  public Dirichlet MixingDist;
}
```

# CyclistMixedTraining Class

The *CyclistMixedTraining* class inherits from *CyclistMixedBase* and implements the training model.

The following example shows the class structure and fields:

```
public class CyclistMixedTraining : CyclistMixedBase
{
  protected Variable<int> NumTrips;
  protected VariableArray<double> TravelTimes;
  protected VariableArray<int> ComponentIndices;

  public override void CreateModel() {...}
  public ModelDataMixed InferModelData(double[] trainingData)
  {...}
}
```

The *NumTrips* and *TravelTimes* fields serve the same purpose as they do in *CyclistTraining*. *ComponentIndices* is a new **int** random variable array with the same data range as *TravelTimes*. It specifies the index of the mixture component that generated the corresponding travel time.

The generative process is as follows:

1) For each mixture component, sample AverageTime and TrafficNoise from their respective priors.
2) Sample the mixing coefficients from the mixing prior.
3) For each trip, do the following:
    a. Sample the component index to determine which mixing component generates the travel time.
    b. Sample the travel time value from a Gaussian distribution whose mean and precision are the AverageTime and TrafficNoise values for that component.

*CyclistMixedTraining.CreateModel* implements this model, as shown in the following example.

```
public override void CreateModel()
{
  base.CreateModel();

  NumTrips = Variable.New<int>();
  Range tripRange = new Range(NumTrips);
  TravelTimes = Variable.Array<double>(tripRange);
  ComponentIndices = Variable.Array<int>(tripRange);

  using (Variable.ForEach(tripRange))
  {
    ComponentIndices[tripRange] =
              Variable.Discrete(MixingCoefficients);
    using (Variable.Switch(ComponentIndices[tripRange]))
    {
      TravelTimes[tripRange].SetTo(
        Variable.GaussianFromMeanAndPrecision(
          AverageTime[ComponentIndices[tripRange]],
          TrafficNoise[ComponentIndices[tripRange]]));
```

```
    }
  }
}
```

**CreateModel** calls the base class method to create and define the common variables. It then creates the field variables, and a **Range** variable that is initialized by *NumTrips*. The core of the training model is in the **using** blocks.

The outer **using** block uses **Variable.ForEach** to iterate through the elements of the *TravelTimes* and *ComponentIndices* arrays. For each iteration, it defines the corresponding *ComponentIndices* element by using a **Discrete** distribution whose parameters are specified by *MixingCoefficients*. The *MixingCoefficients* determine the probability that each of the two mixture components produces the data value.

The inner **using** block implements the *branches* that define the mixture. With standard branches, such as **if-else if**-**else**, one branch executes and the others do not. With probabilistic programming, instead of all-or-nothing, each branch executes with a specified probability, which determines the proportion that the branch contributes to the model.

The probability that each branch executes is specified by a discrete random variable called a *condition*. If, for example, the condition has equal probabilities for all possible values, each branch is activated with equal probability and contributes equally to the model.

CyclingTime3, implements branches by using the **Variable.Switch** method, which represents multiple branches in the generated code, one for each of the condition's possible values. Each branch executes the code in the **using** block, as follows:

- The probability of each of the condition's possible values determines the percentage of the time that the associated branch executes.
- The code is essentially the same for each branch, except that the code in each branch uses the condition's possible value for that particular branch.

For more discussion of branches, see "Branching on Variables to Create Mixture Models" in "Resources".

For *CyclistMixed*:

- The **Switch** condition is the currently selected element of *ComponentIndices*, which generates two branches, one for each mixture component.
- The probability that each branch executes is determined by the mixing coefficients.
- The code in each branch defines the currently selected *TravelTimes* element by using a Gaussian distribution characterized by *AverageTimePrior* and *TrafficNoisePrior* variables for the associated component.

**Note:** *CyclistMixed* uses **Variable.SetTo** to define *TravelTimes* rather than the '=' operator. This is not strictly required for this example, but it is useful to get into the habit of using **SetTo** because it is required in one or two situations . For example, **SetTo** is required if you provide the statistical definition of a scalar random variable in both branches of an **If**/**IfNot** construct. Because Infer.NET cannot overload the C# '='

operator, the definition in the **IfNot** branch will override the definition in the **If** branch.

## InferModelData Method

*InferModelData* infers posteriors based on the priors specified earlier in *SetModelData* and a training data array.

```
public ModelDataMixed InferModelData(double[] trainingData) //Training model
{
  ModelDataMixed posteriors;

  NumTrips.ObservedValue = trainingData.Length;
  TravelTimes.ObservedValue = trainingData;

  posteriors.AverageTimeDist = InferenceEngine.Infer<Gaussian[]>(AverageTime);
  posteriors.TrafficNoiseDist = InferenceEngine.Infer<Gamma[]>(TrafficNoise);
  posteriors.MixingDist = InferenceEngine.Infer<Dirichlet>(MixingCoefficients);

  return posteriors;
}
```

*RunCyclingTime3* passes in a **double** array that contains the training data. *InferModelData* sets *NumTrips* to the array length and assigns the training data to *TravelTimes*. It then infers posteriors for *AverageTime*, *TrafficNoise*, and *MixingCoefficients*, and returns them to *RunCyclingTime3* as a *ModelDataMixed* structure.

# CyclistMixedPrediction Class

The *CyclistMixedPrediction* class implements the prediction model. The following example shows the class structure. It is nearly identical to *CyclistPrediction* except that, for simplicity, there is no *InferProbabilityTimeLessThan* method.

```
public class CyclistMixedPrediction: CyclistMixedBase
{
  private Gaussian TomorrowsTimeDist;
  private Variable<double> TomorrowsTime;

  public override void CreateModel() {...  }
  public Gaussian InferTomorrowsTime() {...}
}
```

*CyclistMixedPrediction* inherits from *CyclistMixedBase* and the fields and methods serve the same purpose as the equivalent fields and methods from *CyclistPrediction*. The differences are in the implementation details.

## CreateModel Method

RunCyclingTime3 calls *CyclistPrediction.CreateModel* to create the prediction model.

```
public override void CreateModel()
{
      base.CreateModel();
```

```
        Variable<int> componentIndex = Variable.Discrete(MixingCoefficients);
        TomorrowsTime = Variable.New<double>();

        using (Variable.Switch(componentIndex))
        {
                TomorrowsTime.SetTo(
                                Variable.GaussianFromMeanAndPrecision(
                                AverageTime[componentIndex],
                                TrafficNoise[componentIndex]));
        }
}
```

The model predicts a single travel time, so there is a single *component index* variable, *componentIndex*. It is defined by a two-element Discrete distribution which is characterized by the mixing coefficients, presumably the posterior determined by the training phase. The model then uses **Variable.Switch** to define *TomorrowsTime* by using a mixture of the two components, with the proportions defined by *MixingCoefficients*.

## InferTomorrowsTime Method

*RunCyclingTime3* calls the *InferTomorrowsTime* method to obtain the predicted distribution for tomorrow's time. The implementation is identical to *CyclistPrediction.InferTomorrowsTime*.

```
public Gaussian InferTomorrowsTime()
{
  TomorrowsTimeDist = InferenceEngine.Infer<Gaussian>(TomorrowsTime);
  return TomorrowsTimeDist;
}
```

# Use the Model

*RunCyclingTime3* creates an instance of *CyclistMixed*, trains the model, and then uses the trained model to predict tomorrow's travel time.

```
static void RunCyclingTime3(string[] args)
{
  ModelDataMixed initPriors;

  double[] trainingData =
              new double[]{ 13, 17, 16, 12, 13, 12, 14, 18, 16, 16, 27, 32 };
  initPriors.AverageTimeDist = new Gaussian[] { new Gaussian(15.0, 100.0), //O
                                                new Gaussian(30.0, 100.0) }; //E
  initPriors.TrafficNoiseDist = new Gamma[] { new Gamma(2.0, 0.5), //O
                                              new Gamma(2.0, 0.5) };//E
  initPriors.MixingDist = new Dirichlet(1, 1);

  CyclistMixedTraining cyclistMixedTraining = new CyclistMixedTraining();
  cyclistMixedTraining.CreateModel();
  cyclistMixedTraining.SetModelData(initPriors);

  ModelDataMixed posteriors = cyclistMixedTraining.InferModelData(trainingData);
```

```
  //Print results

  CyclistMixedPrediction cyclistMixedPrediction = new CyclistMixedPrediction();
  cyclistMixedPrediction.CreateModel();
  cyclistMixedPrediction.SetModelData(posteriors);

  Gaussian tomorrowsTime = cyclistMixedPrediction.InferTomorrowsTime();

  double tomorrowsMean = tomorrowsTime.GetMean();
  double tomorrowsStdDev = Math.Sqrt(tomorrowsTime.GetVariance());

  //Print results
}
```

The training data set is identical to that used by CyclingTime2, except for two additional times at the end of the array that represent extraordinary events.

*RunCyclingTime3* creates the following initial priors:

- The ordinary mixture component has the same initial priors that were used by CyclingTime2.

  They are labelled O in the example.

- The extraordinary mixture component sets the mean travel time to 30, based on the estimate that extraordinary events add around 15 minutes to a trip.

  The extraordinary priors are labelled E in the example. The *AverageTime* initial prior's precision value and the *TrafficNoise* initial prior are set to the same values as used for the ordinary component.

- The initial prior for *MixingDist* is a Dirichlet(1, 1) distribution.

  As discussed later in "Discrete Distributions" in Chapter 6, this distribution corresponds to an equal proportion of both components with a relatively large uncertainty.

**Important:** If you can't estimate initial priors for the mixture components, you typically set them to the same values. However, if you use identical priors for all mixture components you must initialise the model to break its symmetry, which is beyond the scope of this document. See "Tutorial 6: Mixture of Gaussians" in "Resources" for a discussion of how to handle this case.

*RunCyclingTime3* creates an instance of *CyclingMixedTraining*, calls *CreateModel* to create the model, and calls *SetModelData* to specify the priors. *RunCyclingTime3* then calls *InferModelData* to infer posteriors for the model's means, precisions, and mixing coefficients and prints the following results:

```
Average time distribution 1 = Gaussian(14.7, 0.3533)
Average time distribution 2 = Gaussian(29.51, 1.618)
Noise distribution 1 = Gamma(7, 0.0403)[mean=0.2821]
Noise distribution 2 = Gamma(3, 0.1013)[mean=0.304]
Mixing coefficient distribution = Dirichlet(11 3)
```

Distribution 1 corresponds to ordinary travel times and distribution 2 to extraordinary travel times.

After training the model, *RunCyclingTime3* creates an instance of *CyclistMixedPrediction*, creates the model, and sets the priors to the just-computed posteriors. The results are:

```
Tomorrow's expected time: 17.03
Tomorrow's standard deviation: 5.71
```

The mean predicted travel time and the standard deviation are both larger than for CyclingTime2, which reflects the inclusion of extraordinary events in the model.

Chapter 6

# Digression: Discrete Distributions and the Inference Engine

## Discrete Distributions

CyclingTime1 and CyclingTime2 were based on Gaussian and Gamma distributions. They are called continuous distributions because they define probabilities—more precisely, probability densities—for random variables that have a continuous range of possible values. However, some random variables have a specified set of possible values. For example, a random variable that represents the outcome of throwing a standard die would have six possible values.

Random variables with an enumerable set of possible values are defined by *discrete distributions*, which assign a probability to each possible value with the constraint that the probabilities must sum to 1.0. The two most commonly used distributions are Bernoulli and Discrete (sometimes called Categorical), which are briefly discussed in this section.

### Bernoulli Distribution

The Bernoulli distribution is used for variables that have two possible values. Strictly, the Bernoulli distribution is used to define **bool** random variables, but you can generalize it to any pair of possible values by mapping one value to **true** and the other to **false**.

The Bernoulli distribution has a single parameter, *probabilityTrue*, which specifies the probability that the random variable is **true**. For a fixed distribution, you set the parameter to an appropriate value between 0 and 1.0. For example, the following code uses the **Variable.Bernoulli** factory method to create and define the *willBuy* random variable. The variable has a 60% probability of being **true**.

```
Variable<bool> willBuy = Variable.Bernoulli(0.6);
```

The conjugate prior for the Bernoulli distribution's *probabilityTrue* parameter is a Beta distribution, which is a continuous distribution on [0, 1]. It represents the probability that *probabilityTrue* has each of the values from that range.

The Beta distribution's shape is controlled by two parameters, *a* and *b*. The ratio of *a* and *b* controls the location of the peak of the curve:

- If $b/a = 1$, the peak is at x = 0.5.

  If $a = b = 1$, the distribution is flat.
- If $b/a > 1$, the peak is between x = 0 and x = 0.5.

  The larger the ratio, the closer the peak is to 0.
- If $b/a < 1$, the peak is between x = 0.5 and x = 1.0.

  The smaller the ratio, the closer the peak is to 1.0.

The distribution's width is given by $\sqrt{ab}/(a+b)$, so larger values of $a + b$ define a narrower distribution.

Figure 7 shows Beta distributions for some representative values of *a* and *b*.

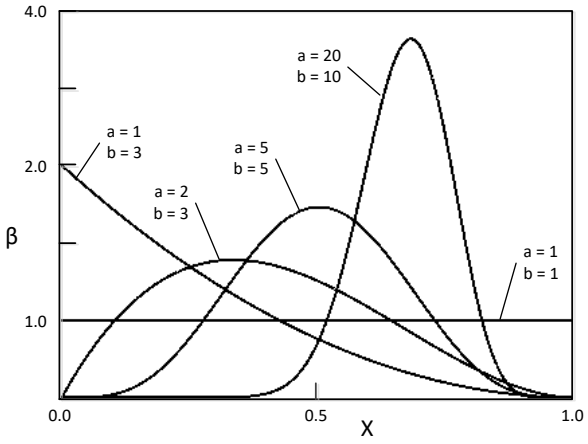


**Figure 7. Beta distribution**

A convenient way to think of *a* and *b* is that they are *pseudo counts* that represent the results of a series of trials:

- *a* is the number of **true** results and b is the number of **false** results.
- If $a = b$, **true** and **false** are equally probable.

  If $a = b = 1$, all possible parameter values have the same probability. For larger values of *a* and *b*, the distribution is peaked, with a mean probability of 0.5.
- If a > b, **true** is more likely, and if a < b, **false** is more likely.
- A larger value of $a + b$ corresponds to more trials, which yields a narrower distribution and a more accurate parameter estimate.

## Discrete Distribution

Variables with two or more possible values are defined by a **Discrete** distribution, which can represent any number of possible values. A **Discrete** distribution has the following characteristics, assuming N possible values:

- The possible values are integers over the range [0, N-1].

  Discrete distributions are thus associated with **Variable<int>** random variables.
- N is a positive integer (not zero).
- Each possible value is associated with a probability in the range [0.0, 1.0].

  The probabilities must sum to 1.0.

The conjugate prior of a **Discrete** distribution's probability vector is a **Dirichlet** distribution, which is a generalization of the **Beta** distribution to any number of possible values. **Dirichlet** distributions are defined by a set of parameters, one for each possible value. Each parameter can be treated as a pseudo-count, much like the Beta distribution parameters:

- The relative values of the pseudo counts define the relative probabilities of each possible value.
- The sum of the pseudo counts defines the variance.

  A larger sum corresponds to a smaller variance and a more accurate estimate.

# How the Inference Engine Works

The code required to infer a posterior distribution is often very complicated and difficult to implement correctly. With Infer.NET, applications don't have to implement this code. Instead, they call the inference engine, which uses the priors, model, and observations to compute the requested marginal "behind the scenes". This section demystifies that process a bit by explaining in general terms how the inference engine works.

The inference engine handles the computation by passing messages between the code that represents different parts of the graph. Figure 8 shows the CyclingTime1 training model, with arrows indicating the message direction.

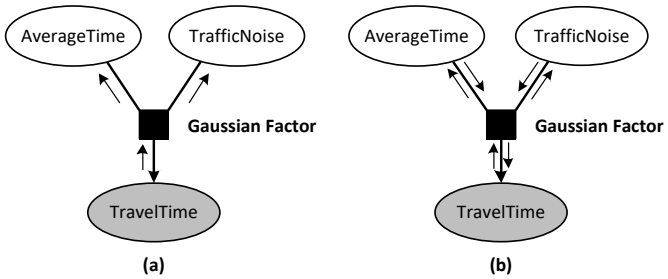


**Figure 8. Message passing**

When you impose a condition on one of the nodes, such as assigning the training data to *TravelTimes*, the inference engine must modify the other nodes' distributions to reflect the change. To do so, *TravelTimes* passes a message to the Gaussian factor, as shown in Figure 8a. The factor then computes two new messages that pass up the graph to *AverageTime* and *TrafficNoise*, which recompute their distributions using the new information.

If this were an exact computation, the process would be finished at this point. However, most inference computations are approximate, so *AverageTime* and *TrafficNoise* are not exact. Instead, they pass messages back to the Gaussian factor. The Gaussian factor passes a message back to *TravelTimes*, and so on, as shown in Figure 8b. The process continues repeating until the engine arrives at a stable set of distributions.

The message passing mechanism doesn't have a preferred direction, which allows the engine to reason backwards as well as forwards. A better way to think about inference is that the engine takes the model, the priors, and any observed data and comes up with a consistent set of distributions for the variables, regardless of where the observed variables and the requested marginal are located on the graph.

The exact details of the messages depend on the particular inference algorithm that is used in the model. The Infer.NET inference engine currently supports three inference algorithms, each of which is implemented by a separate class:

- Expectation propagation, which is the default algorithm (**Microsoft.ML.Probabilistic.ExpectationPropagation**).
- Variational message passing (**Microsoft.ML.Probabilistic.VariationalMessagePassing**).
- Gibbs sampling (**Microsoft.ML.Probabilistic.GibbsSampling**).

Each algorithm has different characteristics, so you choose the one that is best suited for your particular model. For more information, see "Running inference."

**Caution:** The model does not depend on the particular algorithm that you use for inference, so in principle, you can switch algorithms without touching the modelling code. However, the different algorithms don't necessarily support every possible distribution or factor, so some models are compatible only with certain algorithms. For details, see "List of Factors and Constraints" in "Resources."

Another issue that can arise is that sometimes your choice of priors, data, and so on can cause some algorithms to generate *improper messages*, which causes the inference engine to throw an **ImproperDistributionException**. For example, a message with a negative Gaussian precision value would be improper. Sometimes you can fix the problem by using a different inference algorithm. For example, CyclingTime3 generates improper messages with the expectation propagation algorithm, but runs correctly with variational message passing.

For more information about the inference engine, see the "Running Inference" in "Resources."

**Chapter 7**

# CyclingTime4 Application: Model Comparison

There are now two models to represent cyclists' travel time, one based on a single Gaussian and the other based on a mixture of two Gaussians. Why not a model based on a mixture of three Gaussians, or twenty or thirty Gaussians? Or is a single Gaussian sufficient? How do you pick the best model?

In general, a complex model with more adjustable parameters represents a particular data set more accurately than a simpler model with fewer parameters. The more interesting question is whether the more complex model provides a *better* representation than a simpler model. Models that become too good at fitting a particular data set won't be useful in practice since they won't necessarily fit new data well.

For example, when you fit a polynomial to a set of data points, you can always get an exact fit by adding enough elements to the polynomial. However, a model that exactly hits every data point typically swings wildly in between and so won't match new data well. This phenomenon is known as *overfitting*, and might look something like Model A in Figure 9.

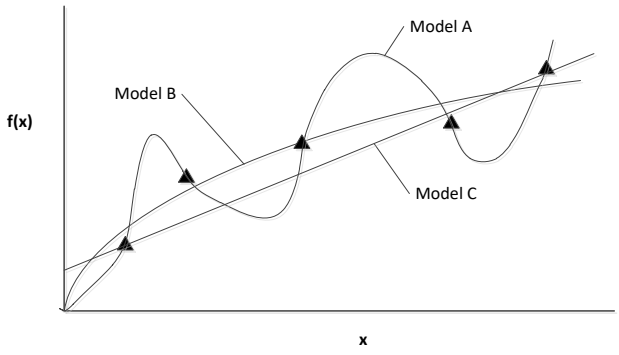


**Figure 9. Fitting and overfitting**

Model A might be more accurate in the sense that it fits the data better than the other two models, but it's not very plausible or useful. If you use Model A to predict a value that is between the original points, the result will probably not be very close to the actual value. Model B fits the data almost as well, but provides much better predictions. Model C is even simpler, but doesn't fit the data or predict future values

as well as Model B. In practice, some complexity is useful, but eventually you reach a point of diminishing returns beyond which additional complexity is detrimental.

The best model is therefore one that fits the data reasonably well without being overly complex. However, visually comparing the models to the data is subjective and not very practical for any but the simplest scenarios. What you need is an *Occam's razor*, which quantitatively evaluates model quality and determines the optimal trade-off between accuracy and complexity. Such a mechanism is in fact an integral part of Bayesian inference, which includes a robust way to evaluate model quality called *model evidence*, which is often abbreviated to *evidence*.

Evidence is essentially a measure of how well a training model predicts the training data, penalised by the probability of the training model itself—more complex models are less probable. Evidence automatically incorporates the trade-off between accuracy and complexity:

- Overly simplistic models have low evidence values because they don't fit data very well.
- Overly complex models have low evidence values because the probability of the particular parameterisation which gives the good fit is very unlikely.

The optimal model is somewhere in between, and indicated by the maximum evidence value. For a more thorough discussion of evidence, see "Bayesian Interpolation" in "Resources."

The CyclingTime4 application uses evidence to assess whether the CyclingTime2 or CyclingTime3 model best represents the CyclingTime3 training data.

## How to Compute Evidence with Infer.NET

With Infer.NET, evidence is represented by a **bool** random variable. The basic procedure is shown schematically in the following example.

```
Variable<bool> Evidence = Variable.Bernoulli(0.5);
using (Variable.If(Evidence))
{
  //Implement the training model to be evaluated
}
  //Observe the training data
  //Query the inference engine for model posteriors
  //Query the inference engine for the evidence distribution
```

Infer.NET defines two-branch models by using **Variable.If** and **Variable.IfNot**, which are the Infer.NET equivalent of **if-else**. The condition is a **bool** random variable. The probability that the condition is true determines the proportion of the **If** branch in the mixture, and the remainder of the mixture is the **IfNot** branch.

To evaluate a model's evidence, you use **If**/**IfNot** as shown in the example, which create a mixture of two models.

- The model that you want to evaluate, which is represented by the **If** branch.
- An "empty" model, which is effectively represented by the missing **IfNot** branch.

The *Evidence* variable is the condition that controls the proportions in the mixture. Its initial prior is usually set to Bernoulli(0.5). To determine the actual evidence distribution, observe the data and compute the *Evidence* posterior.

**Note.** To evaluate evidence in this way:

- All modelling code must be in the **If** block.

  You can declare random variables outside the block, but you must create and define them inside the block.
- The code that queries the inference engine must be outside the **If** block. It is good programming practice to also observe data outside the **If** block, but not required.

The following sections describe how CyclingTime4 uses evidence to compare the CyclingTime2 and CyclingTime3 models:

- The CyclingTime2 model is based on a single Gaussian and is represented by the *CyclistTraining* class.
- The CyclingTime3 model is based on a mixture of two Gaussians and is represented by the *CyclistMixedTraining* class.
- A *RunCyclingTime4 static* method creates and trains the models and displays the results.

CyclingTime4 uses both classes without modification, so they are not discussed here. For simplicity, CyclingTime4 does not do any prediction, so it does not use *CyclistPrediction* and *CyclistMixedPrediction*.

To compute evidence, CyclingTime4 implements a pair of lightweight classes, *CyclistWithEvidence* and *CyclistMixedWithEvidence*, which inherit from *CyclistTraining* and *CyclistMixedTraining*, respectively. The RunCyclingTime4 method creates the models and determines the results.

The CyclingTime4 classes are implemented in CyclingTime4.cs. *RunCyclingTime4* is implemented as a static method in RunCyclingSamples.cs.

## CyclistWithEvidence Class

The *CyclistWithEvidence* class evaluates evidence for the *CyclistTraining* model.

```
public class CyclistWithEvidence : CyclistTraining
{
  protected Variable<bool> Evidence;

  public override void CreateModel()
  {
    Evidence = Variable.Bernoulli(0.5);
    using (Variable.If(Evidence))
    {
      base.CreateModel();
    }
  }
```

```
public double InferEvidence(double[] trainingData)
  {
    double logEvidence;
    ModelData posteriors = base.InferModelData(trainingData);
    logEvidence = InferenceEngine.Infer<Bernoulli>(Evidence).LogOdds;

    return logEvidence;
  }
}
```

*CyclistWithEvidence* inherits from *CyclistTraining*, and has two methods, *CreateModel* and *InferEvidence*.

*CreateModel* implements the evidence block. Within the block, it calls the corresponding base class method to create the model.

*InferEvidence* takes the training data and calls the base class *InferModelData* method to infer the posteriors. It then queries the inference engine for the *Evidence* distribution and returns the posterior's LogOdds value. Evidence values tend to be quite small, so it is usually easier to compare the logarithms than to compare the values themselves.

## CyclistMixedWithEvidence Class

The *CyclistMixedWithEvidence* class evaluates evidence for the *CyclistMixedTraining* model.

```
public class CyclistMixedWithEvidence : CyclistMixedTraining
{
  protected Variable<bool> Evidence;

  public override void CreateModel()
  {
    Evidence = Variable.Bernoulli(0.5);
    using (Variable.If(Evidence))
    {
      base.CreateModel();
    }
  }

  public double InferEvidence(double[] trainingData)
  {
    double logEvidence;
    ModelDataMixed posteriors = base.InferModelData(trainingData);
    logEvidence = InferenceEngine.Infer<Bernoulli>(Evidence).LogOdds;

    return logEvidence;
  }
}
```

The class is nearly identical to *CyclistWithEvidence* and works in the same way. The main difference is that *CyclistMixedWithEvidence* inherits from *CyclistMixedTraining*, which means that the *base.CreateModel* call in *CreateModel* creates the CyclingTime3 model.

# Computing Evidence

*RunCyclingTime4* computes the evidence for the two models and prints the results. Both models use the CyclingTime3 training data set and initial priors.

The first part of *RunCyclingTime4* computes the evidence for the CyclingTime2 model, as shown in the following example.

```
public static void RunCyclingTime4()
{
  double[] trainingData =
              new double[] { 13, 17, 16, 12, 13, 12, 14, 18, 16, 16, 27, 32};

  ModelData initPriors = new ModelData(
    Gaussian.FromMeanAndPrecision(15.0, 0.01),
    Gamma.FromShapeAndScale(2.0, 0.5));
  CyclistWithEvidence cyclistWithEvidence = new CyclistWithEvidence();
  cyclistWithEvidence.CreateModel();
  cyclistWithEvidence.SetModelData(initPriors);

  double logEvidence = cyclistWithEvidence.InferEvidence(trainingData);
  ...
}
```

*RunCyclingTime4* creates an instance of *CyclistWithEvidence*, calls *CreateModel* to create the model, and calls *SetModelData* to specify the initial priors. It then passes the training data to *InferEvidence*, which returns the log of the evidence value.

The second part of *RunCyclingTime4* computes the evidence for the CyclingTime3 model.

```
static void RunCyclingTime4 ( )
{
  ...
  ModelDataMixed initPriorsMixed;
  initPriorsMixed.AverageTimeDist = new Gaussian[] {new Gaussian(15.0, 100),
                                                    new Gaussian(30.0, 100) };
  initPriorsMixed.TrafficNoiseDist = new Gamma[] {new Gamma(2.0, 0.5),
                                                  new Gamma(2.0, 0.5) };
  initPriorsMixed.MixingDist = new Dirichlet(1, 1);

  CyclistMixedWithEvidence cyclistMixedWithEvidence =
          new CyclistMixedWithEvidence();
  cyclistMixedWithEvidence.CreateModel();
  cyclistMixedWithEvidence.SetModelData(initPriorsMixed);

  double logEvidenceMixed = cyclistMixedWithEvidence.InferEvidence(trainingData);

  //Display results
}
```

*RunCyclingTime4* handles the CyclingTime3 evidence computation in essentially the same way as it did for CyclingTime2 and then prints the results, as follows:

```
Log evidence for single Gaussian: -45.80
Log evidence for mixture of two Gaussians: -40.98
```

The evidence values indicate that the mixture model is probably better.

**Note:** Evidence values are usually quite small, especially for large amounts of data. This is why the example uses the logarithm of the evidence value rather than the value itself. What matters is the relative size of the two values, not their absolute values. It is sometimes useful to normalize the log evidence by dividing the value by the number of data points.

**Chapter 8**

# CyclingTime5 Application: A Model for Two Cyclists

The CyclingTime1 - CyclingTime4 examples are all based on a single cyclist's travel time. However, the original scenario was for multiple cyclists, so you need a way to represent each cyclist, handle their data, and compare their performance.

To construct a model for multiple cyclists, you must represent each cyclist by a random variable, and use the techniques discussed in the previous walkthroughs to train the models and compute posteriors. However, you can also use Infer.NET operators to make more complex predictions, such as estimating the probability that one cyclist will have a faster time.

This section is a walkthrough of the CyclingTimes5 sample. It implements a model for two cyclists, and uses the trained model to make some predictions about their relative performance. For the model's factor graph, see Chapter 9.

CyclingTimes5 has three basic components:

- *CyclistTraining* and *CyclistPrediction* classes, instances of which represent each cyclist.

  These are the same classes that were introduced in CyclingTime2, and are used here without modification. An advantage of the programming pattern introduced in CyclingTime2 is that it can be easily extended to handle any number of cyclists.
- A pair of classes, *TwoCyclistsTraining* and *TwoCyclistsPrediction*, which implement the overall training and prediction models, including the code to predict the two cyclists' relative performance.
- A *RunCyclingTime5* method, which uses instances of *TwoCyclistsTraining* and *TwoCyclistsPrediction* to train the model and make predictions.

The CyclingTime5 classes are implemented in CyclingTime5.cs. *RunCyclingTime5* is implemented as a static method in RunCyclingSamples.cs.

## TwoCyclistsTraining Class

The *TwoCyclistsTraining* class implements the overall training model. The basic class structure is similar to *CyclistsTraining*, as shown in the following example:

```
public class TwoCyclistsTraining
{
  private CyclistTraining cyclist1, cyclist2;

  public void CreateModel() {...}

  public void SetModelData(ModelData modelData) {...}

  public ModelData[] InferModelData(double[] trainingData1,
                                    double[] trainingData2) {...}
```

The training model trains the models for each cyclist. The training is handled by the *CyclistTraining* class, as discussed earlier, so *TwoCyclistsTraining* is basically a wrapper over a pair of *CyclistTraining* objects.

## CreateModel Method

The *CreateModel* method creates an instance of *CyclistTraining* for each cyclist and calls their *CreateModel* methods to create the training model.

```
public void CreateModel()
{
  cyclist1 = new CyclistTraining();
  cyclist1.CreateModel();
  cyclist2 = new CyclistTraining();
  cyclist2.CreateModel();
}
```

## SetModelData Method

The *SetModelData* method passes initial priors to the *CyclistTraining.SetModelData* methods.

```
public void SetModelData(ModelData modelData)
{
  cyclist1.SetModelData(modelData);
  cyclist2.SetModelData(modelData);
}
```

For simplicity, *TwoCyclistsTraining* uses the same priors for each model.

## InferModelData Method

The *InferModelData* method determines the posteriors for both *CyclistTraining* models by passing the appropriate training data array to each *CyclistTraining.InferModelData* method.

```
public ModelData[] InferModelData(double[] trainingData1,
                                  double[] trainingData2)
{
  ModelData[] posteriors = new ModelData[2];

  posteriors[0] = cyclist1.InferModelData(trainingData1);
  posteriors[1] = cyclist2.InferModelData(trainingData2);

  return posteriors;
}
```

*InferModelData* returns the posteriors for the two models as an array of *ModelData* structures.

# TwoCyclistsPrediction Class

The *TwoCyclistsPrediction* class implements the overall prediction model. The following example shows the class structure:

```
public class TwoCyclistsPrediction
{
  private CyclistPrediction cyclist1, cyclist2;
  private Variable<double> TimeDifference;
  private Variable<bool> Cyclist1IsFaster;
  private InferenceEngine CommonEngine;

  public void CreateModel() {...}

  public void SetModelData(ModelData[] modelData) {...}

  public Gaussian[] InferTomorrowsTime() {...}

  public Gaussian InferTimeDifference() {...}

  public Bernoulli InferCyclist1IsFaster() {...}
}
```

*TwoCyclistsPrediction* is similar to *CyclistPrediction*. However, it has some additional fields and methods to support predicting the relative performance of the two cyclists, which are discussed in the following sections.

## CreateModel Method

The *CreateModel* method creates the overall model.

```
public void CreateModel()
{
  CommonEngine = new InferenceEngine();

  cyclist1 = new CyclistPrediction() {InferenceEngine = CommonEngine};
  cyclist1.CreateModel();
  cyclist2 = new CyclistPrediction() {InferenceEngine = CommonEngine};
  cyclist2.CreateModel();

  TimeDifference = cyclist1.TomorrowsTime - cyclist2.TomorrowsTime;
  Cyclist1IsFaster = cyclist1.TomorrowsTime < cyclist2.TomorrowsTime;
}
```

*CreateModel* creates an instance of the inference engine, *CommonEngine*, which is used to infer all predicted values. By using a common inference engine, the model is compiled only once, which provides significantly better performance than using three separate engines, each of which would have to compile the model.

The first part of *CreateModel* creates a *CyclistPrediction* instance for each cyclist. It assigns *CommonEngine* to their *InferenceEngine* fields, which ensures that all

prediction-related queries use the same engine. *CreateModel* then calls the *CyclistPrediction.CreateModel* methods to create models for each cyclist.

The final two lines use the *TomorrowsTime* results from the individual models to implement the following predictions about the cyclists' relative performance.

- *TimeDifference* is a **double** random variable that represents the difference between tomorrows predicted travel time for Cyclist 1 and Cyclist 2.
- *Cyclist1IsFaster* is a **bool** random variable that represents the probability that tomorrow, Cyclist 1 is faster than Cyclist 2.

These predictions are implemented by using the '-' (subtraction) and '<' (less-than) operators, respectively. Infer.NET overloads these operators—along with other standard mathematical and logical operators—to perform similar operations on random variables. Infer.NET operators are discussed in more detail in Chapter 9.

## SetModelData and InferTomorrowsTime Methods

These two methods are basically wrappers for the corresponding methods on the two *CyclistPrediction* objects.

```
public void SetModelData(ModelData[] modelData)
{
  cyclist1.SetModelData(modelData[0]);
  cyclist2.SetModelData(modelData[1]);
}
```

```
public Gaussian[] InferTomorrowsTime()
{
  Gaussian[] tomorrowsTime = new Gaussian[2];

  tomorrowsTime[0] = cyclist1.InferTomorrowsTime();
  tomorrowsTime[1] = cyclist2.InferTomorrowsTime();
  return tomorrowsTime;
}
```

The methods pass the appropriate information to the *CyclistPrediction* methods and return the results to *Main*.

## InferTimeDifference and InferCyclist1IsFaster Methods

These methods infer tomorrow's relative performance.

```
public Gaussian InferTimeDifference()
{
  return CommonEngine.Infer<Gaussian>(TimeDifference);
}

public Bernoulli InferCyclist1IsFaster()
{
  return CommonEngine.Infer<Bernoulli>(Cyclist1IsFaster);
}
```

Both methods query for predicted distributions for their respective variables.

- *InferTimeDifference* queries for the *TimeDifference* marginal, and returns the resulting Gaussian distribution.

- *InferCyclist1IsFaster* queries for the *Cyclist1IsFaster* marginal, and returns the resulting Bernoulli distribution.

## Use the Model

The first part of *RunCyclingTime5* creates and trains a *TwoCyclistsTraining* model, and prints the results.

```
static void RunCyclingTime5 (string[] args)
{
  double[] trainingData1 =
          new double[]{ 13, 17, 16, 12, 13, 12, 14, 18, 16, 16, 27, 32 };
  double[] trainingData2 =
          new double[] { 16, 18, 21, 15, 17, 22, 28, 16, 19, 33, 20, 31 };
  ModelData initPriors = new ModelData(new Gaussian(15.0, 100.0),
                                       new Gamma(2.0, 0.5));

  //Train the model
  TwoCyclistsTraining cyclistsTraining = new TwoCyclistsTraining();
  cyclistsTraining.CreateModel();
  cyclistsTraining.SetModelData(initPriors);

  ModelData[] posteriors1 = cyclistsTraining.InferModelData(trainingData1,
trainingData2);

  //Print results
  ...
}
```

*RunCyclingTime5* creates an instance of *TwoCyclistsTraining*, calls *CreateModel* to create the model, and then passes the initial mean and precision prior to *SetModelData*. Cyclist 1 uses the same training data as CyclingTime3 and CyclingTime4, and Cyclist 2 uses a new data set.

*RunCyclingTime5* then calls *InferModelData* to infer posteriors for the cyclists' *AverageTime* and *TrafficNoise* distributions and prints the following results:

```
Cyclist 1 average travel time: Gaussian(17.12, 2.741)
Cyclist 1 traffic noise: Gamma(6.712, 0.00536)[mean=0.03597]
Cyclist 2 average travel time: Gaussian(21.19, 2.722)
Cyclist 2 traffic noise: Gamma(6.4, 0.00577)[mean=0.03693]
```

*RunCyclingTime5* then uses these posteriors to make several predictions as follows:

```
static void RunCyclingTime5 (string[] args)
{
  ...
  TwoCyclistsPrediction cyclistsPrediction = new TwoCyclistsPrediction();
  cyclistsPrediction.CreateModel();
  cyclistsPrediction.SetModelData(posteriors1);

  Gaussian[] posteriors2 = cyclistsPrediction.InferTomorrowsTime();

//Print results

  Gaussian timeDifference = cyclistsPrediction.InferTimeDifference();
  Bernoulli cyclist1IsFaster = cyclistsPrediction.InferCyclist1IsFaster();
```

```
//Print results
}
```

*RunCyclingTime5* creates an instance of *TwoCyclistsPrediction*, calls *CreateModel* to create the prediction model, and passes the *AverageTime* and *TrafficNoise* posteriors to *SetModelData*. It then calls the three prediction methods, with the following results:

```
Cyclist1 tomorrow's travel time: Gaussian(17.12, 35.4)
Cyclist2 tomorrow's travel time: Gaussian(21.19, 34.81)
Time difference: Gaussian(-4.075, 70.22)
Probability that cyclist 1 is faster: Bernoulli(0.6866)
```

The mean predicted time difference shows Cyclist 1 arriving in roughly 17 minutes, well before Cyclist 2, who arrives in about 21 minutes. However, that prediction isn't a firm number; it's the mean value of a distribution that has a relatively large variance. There's a reasonable probability that Cyclist 1 is actually slower than Cyclist 2. A different and perhaps more useful way to evaluate the odds is to look at the probability that Cyclist 1 is faster. Even though the mean time difference is relatively large, the large variance means that the probability that Cyclist 1 is faster is only 69%.

**Chapter 9**

# Digression: Functions of Random Variables

Machine learning applications often must ask questions such as "what is the probability that a random variable is larger than a specified value." With multivariate models, interesting questions often involve two or more variables, such as:

- Which random variable is larger?
- What is the probability that both variables are true?
- What is the sum or difference of two random variables?

You implement such questions in a model by using Infer.NET factor functions, usually abbreviated to just *factor*. Factors are functions that can take and return random variables. Infer.NET supports a wide range of factors, including:

- Mathematical operations such as **Variable<T>.Addition**, which perform operations on individual random variables or pairs of random variables.

  For convenience Infer.NET overrides standard .NET mathematical operators, such as +, *, -, and / to map them to the equivalent Infer.NET factors.
- Boolean and comparison factors perform logical or comparison operations on individual random variables and pairs of random variables.

  For convenience, Infer.NET overrides standard logical operators, such as &, |, >, ==, !, and < to map them to the equivalent Infer.NET factors.
- Linear Algebra factors perform matrix operations on pairs of random variables, including matrix-matrix, matrix-vector and vector-vector (inner product) multiplication.

The Infer.NET factors act on random variables rather than specific values, and produce another random variable. For example, CyclingTime5 uses a subtraction (-) factor to define the difference between the travel times. The resulting *TimeDifference* random variable's distribution specifies the probabilities for the possible differences. For details about the available factors, see "Factors and Constraints" in "Resources."

When using a factor to create a random variable, Infer.NET associates the factor with the resulting **Variable<T>** instance. For example, Figure 10 shows a factor graph for the *TwoCyclist* time difference prediction model.

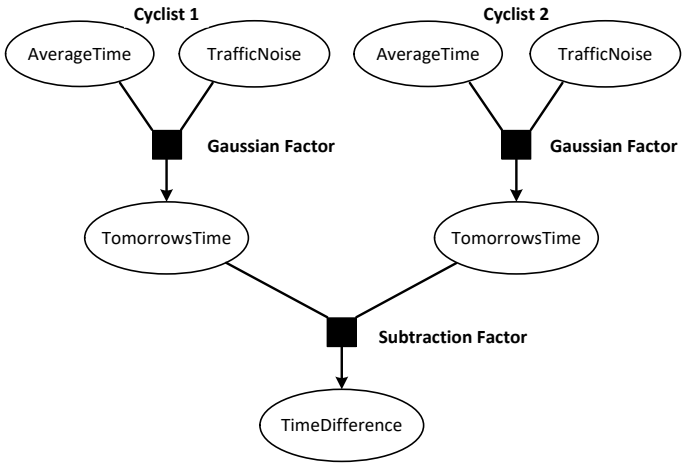


**Figure 10. Factor graph to predict the time difference**

The upper part of the model is implemented by the *Cyclist* class, as discussed earlier. The subtraction factor then operates on the *TomorrowsTime* variables to produce the *TimeDifference* variable. In this case, the variable is approximated by a Gaussian that specifies the distribution over time differences. You could further refine this model by using additional factors to evaluate the probability that Cyclist 1 is faster by more than one minute, and so on.

**Note:** The factor graph for the model that predicts the probability that Cyclist 1 is faster is nearly identical to Figure 10. Just replace the subtraction operator with a less-than (<) factor and the *TimeDifference* random variable with *Cyclist1IsFaster*.

# Resources

This section provides links to additional information about Infer.NET and related topics.

## Infer.NET

**An Introduction to Infer.NET**
https://dotnet.github.io/infer/InferNet_Intro.pdf

**Applying functions and operators to variables**
https://dotnet.github.io/infer/userguide/Applying%20functions%20and%20operators%20to%20variables.html

**Branching on variables to create mixture models**
https://dotnet.github.io/infer/userguide/Branching%20on%20variables%20to%20create%20mixture%20models.html

**Factors and Constraints**
https://dotnet.github.io/infer/userguide/Factors%20and%20Constraints.html

**Infer.NET**
https://dotnet.github.io/infer

**Infer.NET User Guide**
https://dotnet.github.io/infer/userguide

**List of factors and constraints**
https://dotnet.github.io/infer/userguide/list%20of%20factors%20and%20constraints.html

**Running inference**
https://dotnet.github.io/infer/userguide/Running%20inference.html

**Tutorials and Examples**
https://dotnet.github.io/infer/userguide/Infer.NET%20tutorials%20and%20examples.html

**Tutorial 6: Mixture of Gaussians**
https://dotnet.github.io/infer/userguide/Mixture%20of%20Gaussians%20tutorial.html

**Working with arrays and ranges**
https://dotnet.github.io/infer/userguide/Arrays%20and%20ranges.html

**Working with different inference algorithms**
https://dotnet.github.io/infer/userguide/Working%20with%20different%20inference%20algorithms.html

## General Background

**A family of algorithms for approximate Bayesian inference (describes expectation propagation)**
http://research.microsoft.com/en-us/um/people/minka/papers/ep/

**Bayesian inference**
http://en.wikipedia.org/wiki/Bayesian_inference

**Bayesian Interpolation**
http://www.cs.uwaterloo.ca/~mannr/cs886-w10/mackay-bayesian.pdf

**Conjugate prior**
http://en.wikipedia.org/wiki/Conjugate_prior

**Gibbs sampling**
http://en.wikipedia.org/wiki/Gibbs_sampling

**Information Theory, Inference, and Learning Algorithms**
http://www.inference.phy.cam.ac.uk/mackay/itila/book.html

**Pattern Recognition and Machine Learning**
https://www.springer.com/us/book/9780387310732

**Variational message passing**
http://en.wikipedia.org/wiki/Variational_message_passing

## Distributions

**Bernoulli distribution**
http://en.wikipedia.org/wiki/Bernoulli_distribution

**Beta distribution**
http://en.wikipedia.org/wiki/Beta_distribution

**Dirichlet distribution**
http://en.wikipedia.org/wiki/Dirichlet_distribution

**Gamma distribution**
http://en.wikipedia.org/wiki/Gamma_distribution

**Gaussian distribution**
http://en.wikipedia.org/wiki/Gaussian_distribution

**Poisson distribution**
http://en.wikipedia.org/wiki/Poisson_distribution

**Multivariate normal distribution (vector Gaussian)**
http://en.wikipedia.org/wiki/Multivariate_normal_distribution

**Wishart distribution**
http://en.wikipedia.org/wiki/Wishart_distribution

# Appendix

## A: Terminology

This appendix defines terminology specific to Infer.NET that is used in this document.

**Bayesian inference**
An approach to statistical analysis based on Bayes theorem that starts with prior beliefs and uses observations to update those beliefs in a principled way. Bayes theorem is straightforward in concept but the practical computations for all but a small number of cases are intractable. Bayesian inference is therefore done using either sampling techniques or advanced approximation techniques.

**Bernoulli distribution**
A distribution that specifies the probability of two possible outcomes, true or false.

**Beta distribution**
A continuous distribution over the range [0 ,1]. It is the form of the conjugate prior of the parameter of a Bernoulli distribution.

**conjugate prior**
A prior for a variable in a factor is conjugate if its posterior has the same form as the prior.

**continuous distribution**
A distribution with continuous range of possible values.

**Dirichlet distribution**
An extension of the Beta distribution to handle any number of possible values

**discrete distribution**
A distribution with an enumerable set of possible values.

**distribution**
The probability associated with each of a random variable's possible values.

**domain type**
The type of a random variable's possible values. The most common domain types are **bool**, **int**, and **double**.

**evidence**
A measure of how well a model fits the observed data. Evidence balances between fit and model complexity in a principled way based on Bayes theorem.

**factor**
Defines a relationship between two or more random variables. A complex distribution can be built up as a product of factors.

**Gaussian distribution**
A classic “bell curve” distribution, commonly used to define random variables with continuous distributions.

**Gamma distribution**
A continuous distribution on [0, 1]. It is the form of the conjugate prior of the precision parameter of a Gaussian factor.

**inference engine**
Computes a requested posterior marginal, based on a model that defines the system’s joint probability, the priors, and any observations.

**joint probability**
The probability that a set of random variables has specified values. For example, given a set of three Boolean variables, A, B, and C, the probability A and B are true, and C is false is a joint probability.

**machine learning**
Techniques that allow applications to modify their behaviour, based on user interaction.

**marginal probability**
Defines a random variable’s probability distribution after the other random variables have been “summed out”. For example, given a set of three Boolean variables, A, B, and C, the probability that A is true, regardless of the values of B or C is a marginal probability.

**normal distribution**
Another name for a Gaussian distribution.

**observation**
A particular value that is assigned to a random variable. An observation effectively turns a random variable into a normal variable with a well-defined value.

**posterior**
A distribution that represents your understanding of a random variable after making one or more observations.

**prior**
A distribution that represents your prior understanding of a random variable.

**probability density function (pdf)**
A function that describes the relative probabilities associated with a continuous range of possible values. A pdf is scaled so that it integrates across all its values to 1.

**random variable**
A variable whose value is uncertain, and has a set or range of possible values, each of which has an associated probability.

# B: System Requirements and Installation

See the README at https://github.com/dotnet/infer

## Infer.NET Files

The Infer.NET src folder contains the subfolders shown in the following list.

| Subfolder | Description |
|---|---|
| Compiler/bin | Contains debug and release versions of the Infer.NET DLLs, debug (PDB) files, and XML files that contain documentation comments. |
| Learners | Contains Visual Studio projects and solution for complete machine learning applications including classification and recommendation. |
| Examples | Contains Visual Studio projects and solution for the sample applications and examples described in the Infer.NET user guide. It also includes the Visual Studio project for the examples in this document. |
| Runtime | Contains selected parts of the Infer.NET source tree. In particular, this folder includes source code for the distributions, factors, and message operators from the Infer.NET runtime library. |

For more information about the samples and learners, see https://dotnet.github.io/infer

# C: How to Build and Run Infer.NET Applications

The following procedure describes the basics of how to implement an Infer.NET application.

### To Implement an Infer.NET Application

1. Open Microsoft Visual Studio and create a new .NET project.
2. Add references to Microsoft.ML.Probabilistic.Compiler.dll and Microsoft.ML.Probabilistic.dll. This is easiest to do by referencing the Compiler NuGet package, which requires no installation.
3. Add appropriate **using** declarations to your source files.

   The following **using** declarations are used by most Infer.NET applications.

```
using Microsoft.ML.Probabilistic.Models;
using Microsoft.ML.Probabilistic.Distributions;
```