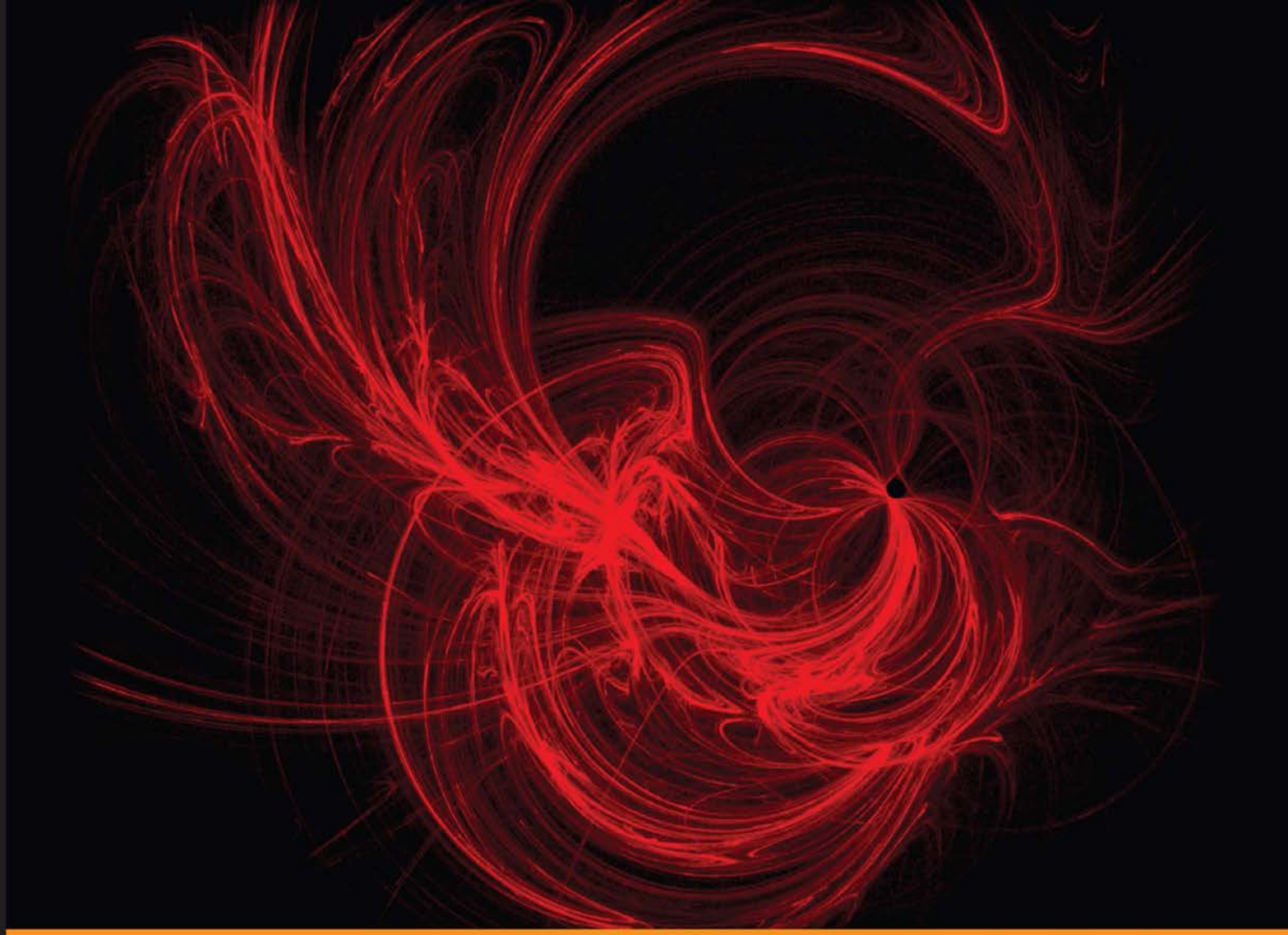


Community Experience Distilled

# Exploring Data with RapidMiner

Explore, understand, and prepare real data using RapidMiner's practical tips and tricks

Andrew Chisholm

[PACKT] PUBLISHING open source* community experience distilled

# Exploring Data with RapidMiner

Explore, understand, and prepare real data
using RapidMiner's practical tips and tricks

**Andrew Chisholm**

[PACKT] PUBLISHING
open source
community experience distilled

BIRMINGHAM - MUMBAI

# Exploring Data with RapidMiner



First published: November 2013

Production Reference: 1181113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-933-8

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

# Credits


**Author**
Andrew Chisholm

**Reviewer**
Venkatesh Umaashankar
Ingo Mierswa

**Acquisition Editor**
Pramila Balan

**Commissioning Editor**
Poonam Jain

**Technical Editors**
Pragnesh Bilimoria
Arwa Manasawala
Anand Singh

**Copy Editor**
Alisha Aranha
Roshni Banerjee
Brandt D'Mello
Mradula Hegde
Dipti Kapadia
Kirti Pai

**Project Coordinator**
Suraj Bist

**Proofreader**
Maria Gould

**Indexer**
Rekha Nair

**Graphics**
Ronak Dhruv

**Production Coordinator**
Pooja Chiplunkar

**Cover Work**
Pooja Chiplunkar

# About the Author

**Andrew Chisholm** completed his degree in Physics from Oxford University nearly thirty years ago. This coincided with the growth in software engineering and it led him to a career in the IT industry. For the last decade he has been very involved in mobile telecommunications, where he is currently a product manager for a market-leading test and monitoring solution used by many mobile operators worldwide.

Throughout his career, he has always maintained an active interest in all aspects of data. In particular, he has always enjoyed finding ways to extract value from data and presenting this in compelling ways to help others meet their objectives. Recently, he completed a Master's in Data Mining and Business Intelligence with first class honors. He is a certified RapidMiner expert and has been using this product to solve real problems for several years. He maintains a blog where he shares some miscellaneous helpful advice on how to get the best out of RapidMiner.

He approaches problems from a practical perspective and has a great deal of relevant hands-on experience with real data. This book draws this experience together in the context of exploring data—the first and most important step in a data mining process.

He has published conference papers relating to unsupervised clustering and cluster validity measures and contributed a chapter called *Visualizing cluster validity measures* to an upcoming book entitled *RapidMiner: Use Cases and Business Analytics Applications*, Chapman & Hall/CRC

---

I would like to thank my family, and in particular my wife Jennie for putting up with me while I wrote this book.

---

# About the Reviewer

**Venkatesh Umaashankar** is an analytics professional with a rich experience in implementing data mining and machine learning systems. His main areas of interest are machine learning and big data. He is also an avid learner and follower of new developments in the field of machine learning and its practical application. He blogs about machine learning at `http://intelligencemining.blogspot.com`.

# www.PacktPub.com

## Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.


`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

## Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

# Table of Contents


**Preface** 1

**Chapter 1: Setting the Scene** 7

- **A process framework** 8
- **Data volume and velocity** 10
- **Data variety, formats, and meanings** 11
- **Missing data** 12
- **Cleaning data** 12
- **Visualizing data** 13
- **Resource constraints** 13
- **Terminology** 14
- **Accompanying material** 15
- **Summary** 16

**Chapter 2: Loading Data** 17

- **Reading files** 17
  - Alternative delimiters 20
  - Reading complete lines 21
  - Reading large numbers of attributes 21
  - Splitting files into smaller pieces 23
- **Databases** 25
  - The Read Database operator 25
  - Large datasets 27
- **Using macros** 27
- **Summary** 28

**Chapter 3: Visualizing Data** **29**
**Getting started** **29**
**Statistical summaries** **30**
**Relationships between attributes** **32**
Scatter plots 32
Scatter 3D color 34
Parallel and deviation 35
Quartile color 38
**Time series data** **39**
Plotting series 39
Using the survey plotter 42
**Relations between examples** **43**
Using histograms 44
Using block plots 45
**Summary** **47**
**Chapter 4: Parsing and Converting Attributes** **49**
**Generating attributes** **50**
Date functions 51
Regular expression functions 53
Generating extracts 54
Regular expressions 54
XPath 57
**Renaming attributes** **59**
Searching and replacing attribute values 59
Using the Map operator 59
Using the Replace operator 60
Using the Replace (Dictionary) operator 60
**Summary** **62**
**Chapter 5: Outliers** **63**
**Manual inspection** **63**
Increasing the data volume 68
Rules for handling outliers 68
**Automated detection of example outliers** **69**
The Detect Outlier (Distances) operator 69
The Detect Outlier (Densities) operator 73
The Detect Outlier (LOF) operator 74
The Detect Outliers (COF) operator 75
**Summary** **76**

**Chapter 6: Missing Values** **77**
**Missing or empty?** **77**
**Types of missing data** **78**
Missing completely at random 78
Missing at random 78
Not missing at random 79
**Categorizing missing data** **79**
Finding MCAR data 83
Finding MAR data 85
Finding NMAR data 86
A cautionary note 87
**Effect of missing data** **88**
**Options for handling missing data** **88**
Returning to the root cause 89
Ignoring it 89
Manual editing 89
Deletion of examples 90
Deletion of attributes 90
Imputation with single values 90
Modeling 91
**Summary** **91**
**Chapter 7: Transforming Data** **93**
**Creating new attributes** **94**
**Aggregation** **98**
**Using pivoting** **100**
**Using de-pivoting** **102**
**Summary** **106**
**Chapter 8: Reducing Data Size** **107**
**Removing examples using sampling** **107**
**Removing attributes** **108**
Removing useless attributes 109
Weighting attributes 111
Selecting attributes using models 114
**Summary** **119**

**Chapter 9: Resource Constraints 121**
**Measuring and estimating performance 121**
Measuring performance 122
**Adding memory 129**
**Parallel processing 130**
**Restructuring processes 131**
**Summary 131**
**Chapter 10: Debugging 133**
**Breakpoints in RapidMiner Studio 133**
**Logging data in RapidMiner Studio 134**
**RapidMiner Studio console printing 135**
**Groovy scripts 136**
Outputting macros example 137
Console logging with Groovy 137
**Regex tools 138**
**Using XPath effectively 138**
**Summary 139**
**Chapter 11: Taking Stock 141**
**Exploring new techniques 142**
Time series 142
Web mining 142
Using R 142
Java or Groovy 142
Third-party components 143
RapidMiner Server 143
**Where to go next 143**
**Index 145**

# Preface

This book is a practical guide to exploring data using RapidMiner Studio. Something like 80 percent of a data mining or predictive analytics project is spent importing, cleaning, visualizing, restructuring, and summarizing data in order to understand it. This book focuses on this vital aspect and gives practical advice using RapidMiner Studio to help with the process.

A number of techniques are illustrated and it is the nature of exploratory data analysis that they can be re-used and modified in different places. By drawing these techniques together into a context, the reader will get a better sense of how RapidMiner Studio can be used in general and gain more confidence to use it.

## What this book covers

*Chapter 1*, *Setting the Scene*, describes the main challenges when mining real data. These challenges arise because data is big and, in the real world, it is unstructured, difficult to visualize, and time consuming to bring order to.

*Chapter 2*, *Loading Data*, describes the different ways of loading data into RapidMiner Studio and the advanced techniques sometimes needed to transform raw unstructured data into a common format.

*Chapter 3*, *Visualizing Data*, describes the visualization techniques available in RapidMiner Studio to help make sense of data.

*Chapter 4*, *Parsing and Converting Attributes*, explains that data is rarely in precisely the right format and, therefore, needs to be parsed to extract specific information or converted into a different representation.

*Chapter 5*, *Outliers*, explains that real data contains values that do not seem to fit the rest of the data. There are many reasons for this and it is important to have a strategy for identifying and dealing with them, otherwise model accuracy risks can be severely compromised.

*Chapter 6*, *Missing Values*, explains that real data inevitably contains missing values. Simple deletion of rows containing missing values can quickly lead to a significant reduction in the performance of a data mining algorithm. Much better techniques exist.

*Chapter 7*, *Transforming Data*, covers techniques to restructure the data into new representations that can assist its exploration and understanding.

*Chapter 8*, *Reducing Data Size*, explains that reducing the number of rows will generally speed up processing but will reduce accuracy. Balancing this is important for large datasets. Reducing the number of columns of data can often improve model accuracy and for large datasets it is doubly valuable as it can speed up processing in general.

*Chapter 9*, *Resource Constraints*, explains that processing large amounts of data requires a lot of physical processing power and memory, to say nothing of the amount of time. This chapter gives some techniques to help measure process performance. Sometimes, it is not possible to process the data using available resources and in this situation, some techniques can be adopted to persuade the process to complete.

*Chapter 10*, *Debugging*, explains that when something goes wrong, it can be frustrating and time consuming to detect and resolve the problem. This chapter gives some generic methods for making this process a bit easier.

*Chapter 11*, *Taking Stock*, explains that having reached this point, the reader will have a greater visibility of the possibilities to process, clean, and explore data as part of the data mining process. This will be a stepping stone to more complex data mining techniques.

# What you need for this book

You will need some basic previous exposure to RapidMiner.The latest version of RapidMiner is now RapidMiner Studio which adds some templating features to help analysts get going more quickly as well as some changes to the look and feel of the GUI in general. This book uses the latest version and it is assumed that you have installed it so that you can download and try the example processes that are worked through in the text.

# Who this book is for

This book is for data analysts with some experience of RapidMiner who wish to use it to explore real data as part of an overall data mining or business intelligence objective. It is very likely that the analyst may have spent some initial time on mining data but could not get the results they wanted. This book gives some real examples and helps to build a context in which data exploration can be done.

# Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "To identify missing attributes, the `Filter Examples` operator can be used."

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The **featureNames** attribute shows the attributes that were used to create the various performance measurements."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

# Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

# Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

## Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output.You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/9338OS_Images.pdf`.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

# Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

# Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

# 1
# Setting the Scene

You have data, you know it has hidden value, and you want to mine it. The problem is you're a bit stuck.

The data you have could be anything and you have a lot of it. It is probably from where you work, and you are probably very knowledgeable about how it is gathered, how to interpret it, and what it means. You may also know a domain expert to whom you can turn for additional expertise.

You also have more than a passing knowledge of data mining and you have spent a short time becoming familiar with **RapidMiner** to perform data mining activities, such as clustering, classification, and regression. You know well that mining data is not just a case of using a spreadsheet to draw a few graphs and pie charts; there is much more.

Given all of this, what is the problem, why are you stuck, and what is this book for?

Simply put, real data is huge, stored in a multitude of formats, contains errors and missing values, and does not yield its secrets willingly. If, like me, your first steps in data mining involved using simple test datasets with a few hundred rows (all with clean data), you will quickly find that 10 million rows of data of dubious quality stored in a database combined with some spreadsheets and sundry files presents a whole new set of problems. In fact, estimates put the proportion of time spent cleaning, understanding, interpreting, and exploring data at something like 80 percent. The remaining 20 percent is the time spent on mining.

The problem restated is that if you don't spend time cleaning, reformatting, restructuring, and generally getting to know your data as part of an exploration, you will remain stuck and will get poor results. If we agree that this is the activity to be done, we come to a basic question: how will we do this?

The answer to this problem for this book is to use RapidMiner, a very powerful and ultimately easy-to-use product. These features, coupled with its open source availability, means it is very widely used. It does have a learning curve that can seem daunting. Be assured, once you have ascended this, the product truly becomes easy to use and lives up to its name.

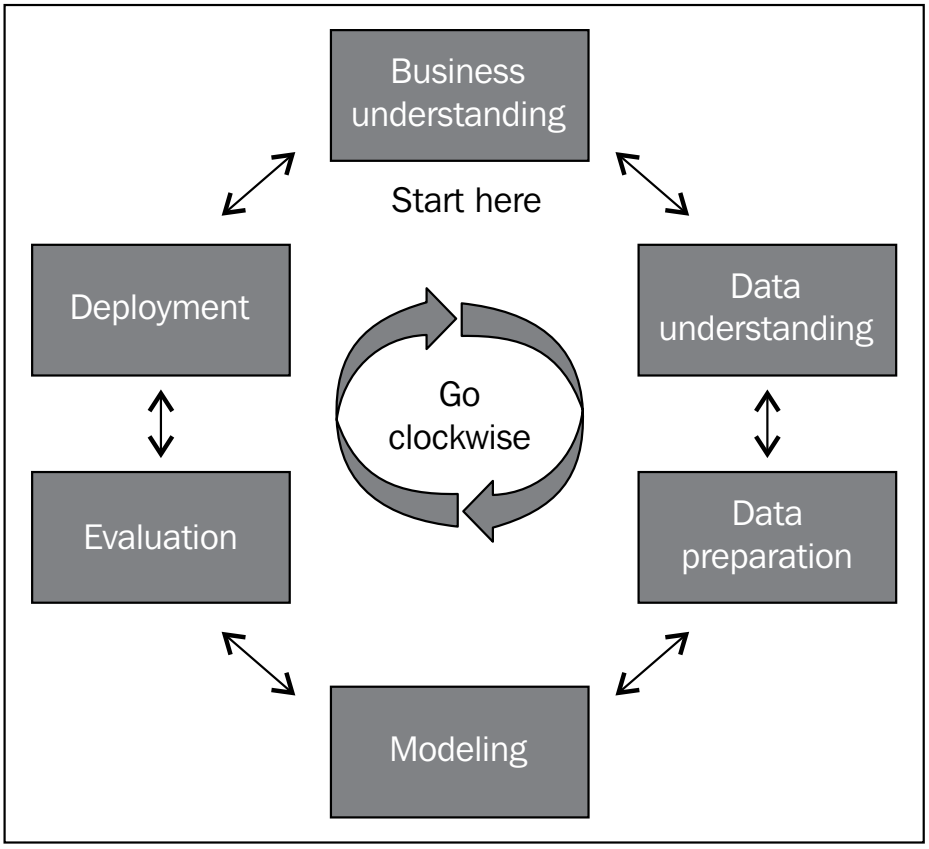
This book is therefore an intermediate-level practical guide to using RapidMiner to explore data and includes techniques to import, visualize, clean, format, and restructure data. This overall objective gives a context in which the various techniques can be considered together. This is helpful because it shows what is possible and makes it easier to modify the techniques for whatever real data is encountered. Hints and tips are provided along the way; in fact, some readers may prefer to use these hints as a starting point.

Having set the scene, let us consider some of the aspects of data exploration raised in this introduction. The following sections explain some of the aspects of data exploration and give references to chapters where these aspects are considered in detail.

# A process framework

It is important to think carefully about the framework within which any data mining investigation is done. A systematic yet simple approach will help results happen and will ensure everyone involved knows what to do and what to expect.

The following diagram shows a simple process framework, derived in part from CRISP-DM (`ftp://ftp.software.ibm.com/software/analytics/spss/documentation/modeler/14.2/en/CRISP_DM.pdf`):




There are six main phases. The process starts with **Business understanding** and the whole process proceeds in a clockwise direction, but it is quite normal to return, at any stage, to the previous phases in an iterative way. Not all the stages are mandatory. It is possible that the business has an objective that is not related to data mining and modeling at all. It might be enough to summarize large volumes of data in some sort of dashboard, so the **Modeling** step would be ignored in this case.

The **Business understanding** phase is the most important phase to get correct. Without clear organizational objectives set by what we might loosely call the business, as well as its continuing involvement, the whole activity is doomed. The output from this phase is considered the criteria for determining success. For the purpose of this book, it is assumed that this critical phase has been started and this clear view exists.

**Data understanding** and **Data preparation** follow **Business understanding**, and these phases involve activities such as importing, extracting, transforming, cleaning, and loading data into new databases and visualizing and generally getting a thorough understanding of what the data is. This book will be concerned with these two phases.

The **Modeling**, **Evaluation,** and **Deployment** phases concern building models to make predictions, testing these with real data, and deploying them in live use. This is the part that most people regard as data mining but it represents 20 percent of the effort. This book does not concern itself with these phases in any detail.

Having said that, it is important to have a view of the **Modeling** phase that will eventually be undertaken because this will impact the data exploration and understanding activity. For example, a predictive analytics project may try to predict the likelihood of a mobile phone customer switching to a competitor based on usage data. This has implications for how the data should be structured. Another example is using online shopping behavior to predict customer purchases, where a market basket analysis would be undertaken. This might require a different structure for the data. Yet another example would be an unsupervised clustering exercise to try and summarize different types of customers, where the aim is to find groups of similar customers. This can sometimes change the focus of the exploration to find relationships between all the attributes of the data.

**Evaluation** is also important because this is where success is estimated. An unbalanced dataset, where there are few examples of the target to be predicted, will have an effect on the validation to be performed. A regression modeling problem, which estimates a numerical result, will also require a different approach to a classification in which nominal values are being predicted.

Having set the scene for what is to be covered, the following sections will give some more detail about what the **Data understanding** and **Data preparation** phases contain, to give a taste of the chapters to come.

# Data volume and velocity

There is no doubt that data is growing. Even a cursory glance at historical trends and future predictions reveals graphs trending sharply upwards for data volumes, data sources, and datatypes as well as for the speed at which data is being created. There are also graphs showing the cost of data storage going down, linked to the increased power and reduced cost of processing, the presence of new devices such as smartphones, and the ability of standard communication networks such as the Internet to make the movement of data easy.

So, there is more and more data being generated by more and more devices and it is becoming easier to move it around.

However, the ability of people to process and understand data remains constant. The net result is a gap in understanding that is getting wider.

For evidence of this, it is interesting to use Google Trends to look for search terms such as data visualization, data understanding, data value, and data cost. All of these have been trending up to a greater or lesser extent since 2007. This points to the concerns that people have which causes them to search for these terms because they are being overwhelmed with data.

Clearly, there is a need for something to help close the understanding gap to make the process of exploring data more efficient. As the first step, therefore, *Chapter 8*, *Reducing Data Size*, and *Chapter 9*, *Resource Constraints*, give some practical advice on determining how long a RapidMiner process will take to run. Some techniques to sample or reduce the size of data are also included to allow results to be obtained within a meaningful time span while understanding the effect on accuracy.

# Data variety, formats, and meanings

For the purpose of this book, data is something that can be processed by a computer. This means that it is probably stored in a file on a disk or in a database or it could be in the computer's memory. Additionally, it might not physically exist until it is asked for. In other words, it could be the response to a web service query, which mashes up data sources to produce a result. Furthermore, some data is available in real time as a result of some external process being asked to gather or generate results.

Having found the data, understanding its format and the fields within it represents a challenge. With the increase of data volume comes an inevitable increase in the formats of data, owing simply to there being more diverse sources of data. User-generated content, mash-ups, and the possibility of defining one's own XML datatypes means that the meaning and interpretation of a field may not be obvious simply by looking at its name.

The obvious example is date formats. The date 1/5/2012 means January 5, 2012 to someone from the US whereas it means May 1, 2012 to someone from the UK. Another example in the context of a measurement of time is where results are recorded in microseconds, labeled as elapsed time, and then interpreted by a person as being in seconds. Yet another example could be a field labeled Item with the value Bat. Is this referring to a small flying mammal or is it something to play cricket with?

To address some aspects of data, *Chapter 2*, *Loading Data*, *Chapter 4*, *Parsing and Converting Attributes*, and *Chapter 7*, *Transforming Data*, take the initial steps to help close the understanding gap mentioned earlier.

# Missing data

Most data has missing values. These arise for many reasons by virtue of errors during the gathering process, deliberate withholding for legitimate or malicious reasons, and simple bugs in the way data is processed. Having a strategy to handle this is very important because some algorithms perform very poorly even with a small percentage of missing data.

On the face of it, missing data is easy to detect, but there is a pitfall for the unwary since a missing value could in fact be a completely legitimate empty value. For example, a commuter train could start at one station and stop at all intermediate stations before reaching a final destination. An express train would not stop at the intermediate stations at all, and there would be no recorded arrival and departure times for these stops. This is not missing data but if it is handled like it is, the data would become unrepresentative and would lead to unpredictable results when used for mining.

That's not all; there are different types of missing data. Some are completely random, while some depend on the other data in complex ways. It is also possible for missing data to be correlated with the data to be predicted. Any strategy for handling missing values has therefore to consider these issues because the simple strategy of deleting records does not only remove precious data but could also bias the results of any data mining activity. The typical starting approach is to fill missing values manually. This is not advisable because it is time consuming, error prone, risks bias, is not repeatable, and does not scale.

What is needed is a systematic method of handling missing values and determining a way to process them automatically with little or no manual intervention. *Chapter 6*, *Missing Values*, takes the first step on this road.

# Cleaning data

It is almost certain that any data encountered in the real world has data quality issues. In simple terms, this means that values are invalid or very different from other values. Of course, it can get more complex than this when it is not at all obvious that a particular value is anomalous. For example, the heights of people could be recorded and the range could be between 1 and 2 meters. If there is data for young children in the sample, lower heights are expected, but isn't a 2-meter five-year-old child an anomaly? It probably is, but anomalies such as these usually occur.

As with missing data, a systematic and automatic approach is required to identify it and deal with it and *Chapter 5*, *Outliers*, gives some details.

# Visualizing data

A picture paints a thousand words, and this is particularly true when trying to understand data and close the understanding gap. Faced with a million rows of data, there is often no better way to view it to understand what quality issues there are, how the attributes within it relate to one another, and whether there are other systematic features that need to be understood and explained.

There are many types of visualizations that can be used and it is also important to combine these with the use of descriptive statistics, such as the mean and standard deviation.

Examples include 2D and 3D scatter plots, density plots, bubble charts, series, surfaces, box plots, and histograms, and it is often important to aggregate data into summaries for presentation because the larger the data gets, the more time it takes to process. Indeed, it becomes mandatory to summarize data as the resource limits of the available computers are reached.

Some initial techniques are given in *Chapter 3*, *Visualizing Data*.

# Resource constraints

There is never enough time and there is never enough money. In other words, there is never enough time to get all the investigation and processing done, both in terms of the capacity of a person to look at the data and understand it as well as in terms of processing power and capacity. To be valuable in the real world, it must be possible to process all the data in a time that meets the requirements set at the outset. Referring back to the overall process, the business objectives must consider and set acceptance criteria for this.

This pervades all aspects of the data mining process from loading data, cleaning it, handling missing values, transforming it for subsequent processing, and performing the classification or clustering process itself.

When faced with huge data that is taking too long to process, there are many techniques that can be used to speed things up and *Chapter 9*, *Resource Constraints*, gives some details. This can start by breaking the process into steps and ensuring that intermediate results are saved. Very often, an initial load of data from a database can dwarf all other activities in terms of elapsed time. It may also be the case that it is simply not possible to load the data at all, making a batch approach necessary.

It is well known that different data mining algorithms perform differently depending on the number of rows of data and the number of attributes per row. One of the outputs from the data preparation phase is a dataset that is capable of being mined. This means that it must be possible for the data to be mined in a reasonable amount of time and so it is important that attention is paid to reducing the size of the data while bearing in mind that any reduction could affect the accuracy of the resulting data mining activity.

Reducing the number of rows by filtering or by aggregation is one method. An alternative method to this is to summarize data into groups. Another approach is to focus on the attributes and remove those that have no effect on the final outcome. It is also possible to transform attributes into their principal components for summarization purposes.

All of this does not help you think any quicker, but by speeding up the intermediate steps, it helps keep the train of thought going as the data is being understood.

# Terminology

The following table contains some common terms that RapidMiner uses:

| Term | Explanation |
|---|---|
| **Process** | A process is an executable unit containing the functionality to be executed. The user creates the process using operators and joins them together in whatever way is required. |
| **Operator** | An operator is a single block of functionality available from the RapidMiner Studio GUI that can be arranged in a process and connected to other processes. Each operator has parameters that can be configured as per the specific requirements of the process. |
| **Macro** | A macro is a global variable that can be set and used by most operators to modify operator behavior. |
| **Repository** | A repository is a location where processes, data, models, and files can be stored and read either from the RapidMiner Studio GUI or from a process. |
| **Example** | An example is a single row of data. |
| **Example set** | This is a set of one or more examples. |
| **Attribute** | An attribute is a column of data. |
| **Type** | This is the type of an attribute. It can be `real`, `integer`, `date_time`, `nominal` (both polynominal and binominal), or `text`. |

| Term | Explanation |
|---|---|
| **Role** | An attribute's role dictates how operators will use the attribute. The most obvious role is regular. The other standard types are known as special attributes and these include `label`, `id`, `cluster`, `prediction`, and `outlier`. It is also possible to set the role of an attribute that is generally ignored by most operators (there are exceptions). |
| **Label** | A label is the target attribute to be predicted in a data mining classification context. This is one of the special role types for an attribute. |
| **ID** | This is a special role that indicates an identifier for an example. Some operators use the ID as part of their operation. |

This table is given here so that readers are aware of the terminology up front and to make it easier to find later.

# Accompanying material

Many RapidMiner processes have been produced for this book, and most are available on the Internet.

Some of the processes contain additional bonus material. Note that, where files need to be accessed, you will have to edit the processes to match the locations of your files.

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

# Summary

So far, we have seen in detail how extracting value from data should be considered an iterative process that consists of a number of phases. It is important to have a clear business objective as well as continued involvement of key people throughout the process. The important point is that the bulk of data mining is about data cleaning, exploration, and understanding, which means it is important to make this clear at the beginning to avoid disappointment.

Having seen some of the aspects of data cleaning, exploring, and understanding, you recognize some of the practical issues you have faced that have prevented you from getting value out of your data.

Without further ado, let's get straight onto the next logical step: importing data. RapidMiner provides many ways to do this and these are covered in *Chapter 2*, *Loading Data*.

# 2
# Loading Data

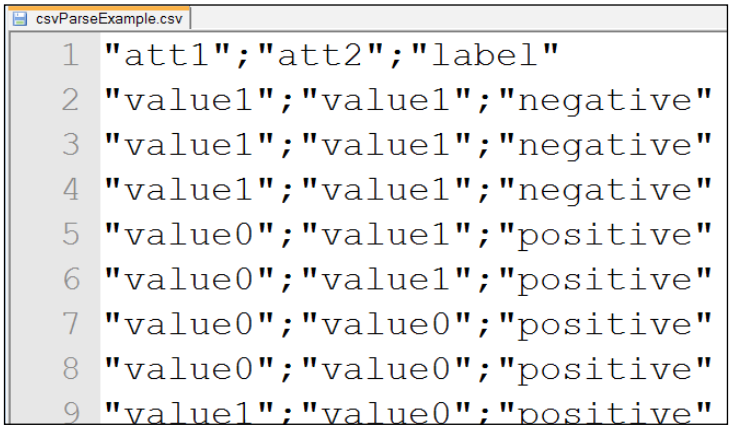
There are many ways of loading data into RapidMiner. This chapter will cover the important ones when reading from files or databases. Keeping in mind that we are exploring data, the chapter focuses on using the features of RapidMiner to work around the typical issues that real data can cause. Some issues can occur without being noticed, so it is particularly important to be vigilant.

## Reading files

The most used operator to import flat files is `Read CSV`. This is a powerful operator that is capable of far more than importing comma-separated values. The operator has a wizard that can be used to take some of the tedious work out of setting up the details of the fields to import, and generally, this works well. However, there are some points to take care of.

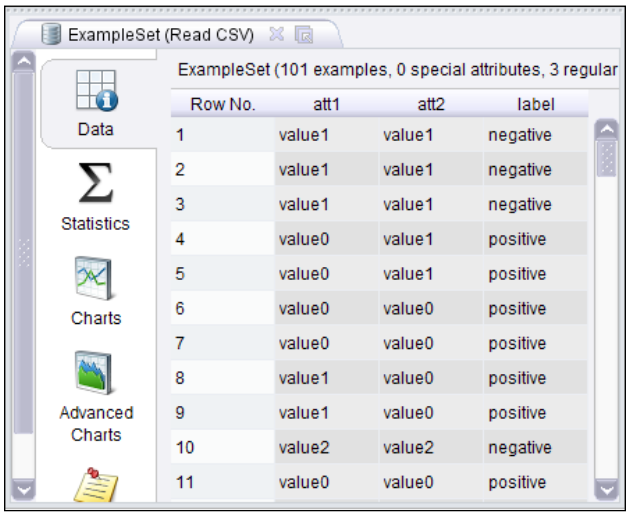
An example process, `ReadCSVGuessing.xml`, is provided with the files that accompany this book. This process creates a CSV file and then reads it back in and can be used in conjunction with the following sections to help understand them more quickly. If this is run once, a CSV file will be created. The process includes a `Read CSV` operator, and by choosing the wizard from the operator's configuration options, it will be easier to follow the next sections.

At the most basic level, the result of running the `Read CSV` operator is an example set based on the contents of the file. The following screenshot shows a subset of the CSV file generated by the `ReadCSVGuessing.xml` process:




Each line consists of three entries separated by semicolons. The first line corresponds to the names of the attributes, and the subsequent lines correspond to the values for the rows. Notice how the fields are surrounded by double quotation marks. All in all, this is a very well behaved file and is easy to read. Real files often have all sorts of issues, which we will cover shortly.

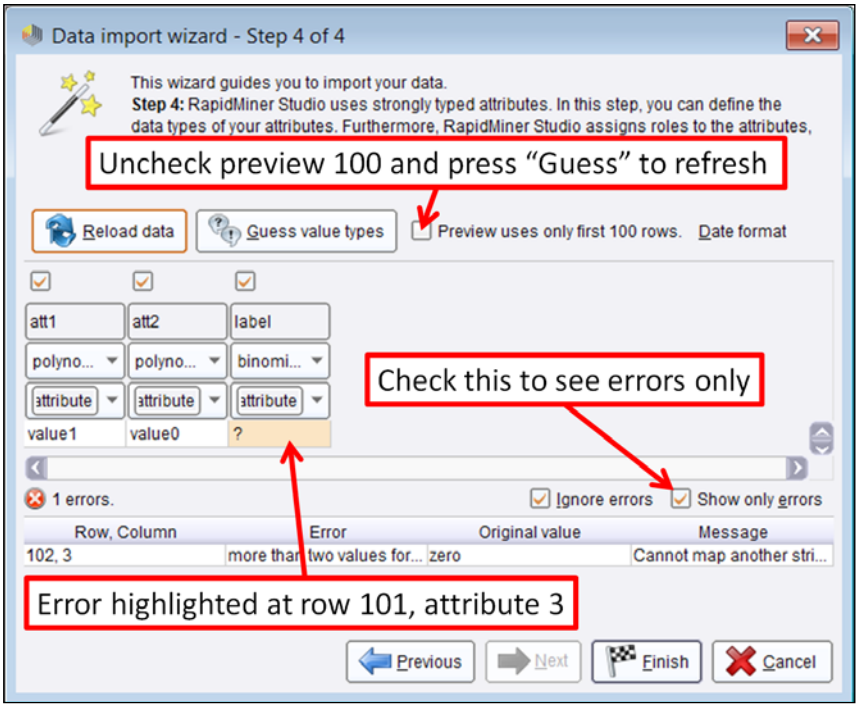
After the file is read, the RapidMiner example set corresponding to this data is displayed as shown in the following screenshot:



This is the first opportunity to explore the data, and an inspection of this process shows how the fields in the CSV file map the examples and attributes in the example set.

As said earlier, we are exploring real data, which is always more challenging. A good first step to deal with this is to use the wizard built into the `Read CSV` operator. This wizard is started by clicking on the **Import Configuration Wizard** button in the parameter view for the operator.

The wizard uses the first 100 rows that it encounters to infer the types of the attributes (this value is configurable from the RapidMiner settings options). If these rows contain two possible values for an attribute, RapidMiner will guess that the type is **binominal**. If the 101st row contains a third possible value, the import will generate an error. Using the `ReadCSVGuessing.xml` process from earlier, deselect the checkbox to preview the first 100 rows and click on the **Reload data** button. Errors should show up as shown in the following diagram:




In this case, changing the type of the last column to **polynominal** and reloading the data will make the error disappear. Guessing value types will set the type to **binominal** again.

If the file is large, the wizard may take some time, so it may be advisable either to run the import and follow this process with the `Filter Examples` operator (for more details, refer to *Chapter 8*, *Reducing Data Size*) to see examples with missing attributes. Or you can split the file into smaller parts as described in the *Splitting files into smaller pieces* section.

# Alternative delimiters

The field delimiter used in the previous example is a semicolon. In real explorations, many different delimiters will be encountered. It is possible to parse these using regular expressions. For example, the default regular expression provided in step 2 of the wizard is as follows:

```
,\s*|;\s*
```

This expression means the following:

*Look for a comma followed by some optional white space or look for a semicolon followed by some optional white space*.

The expression will allow a file containing a mix of commas and semicolons as delimiters to be read although usually only one type of delimiter is used and these are most commonly semicolons or commas.

Another example would be as follows:

```
:{3}
```

This expression will look for three consecutive colons and treat them as the delimiter.

In passing, it is worth mentioning that regular expressions are very commonly used within RapidMiner. At various points, regular expressions will be used in this book, and where possible, an English translation will be given. In addition, refer to *Chapter 10*, *Debugging*, to find a further summary of why these are so important, where they are used and some tools to help.

Delimiter handling can be challenging if the delimiter is included as valid text within some of the values for fields. This particularly affects commas because many numbering schemes use them to delimit numbers. Vigilance is needed to spot these in general, and sometimes the only thing to do is parse the contents outside the wizard as described in the next section.

# Reading complete lines

When using the `Read CSV` wizard to explore real data, you will inevitably encounter a situation where lines are too complex to parse. In such a case, reading each complete line one by one is a sensible course of action as it allows more complex multistep processing to be performed later. In this case, the line terminators are treated as delimiters. To do this, use a regular expression to look for carriage returns and line feeds as follows:

```
\r\n
```

It is generally necessary to deselect the use of quotes to make this expression work.

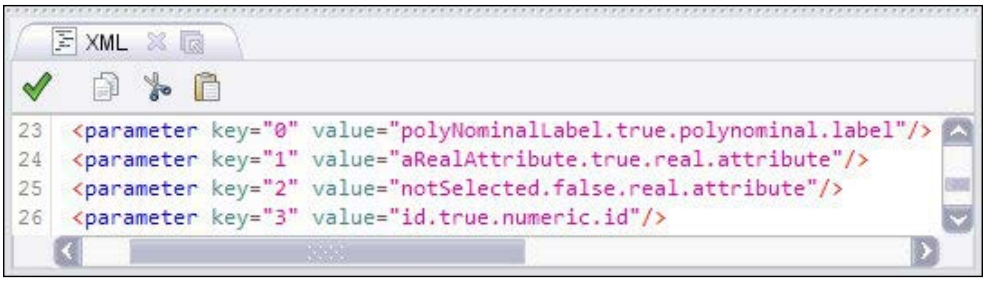
The number of examples will correspond to the number of lines in the file. Some UNIX files may not have both the carriage return and line feed as line terminators. In this case, it is necessary to use either `\r` or `\n` by itself as the regular expression.

# Reading large numbers of attributes

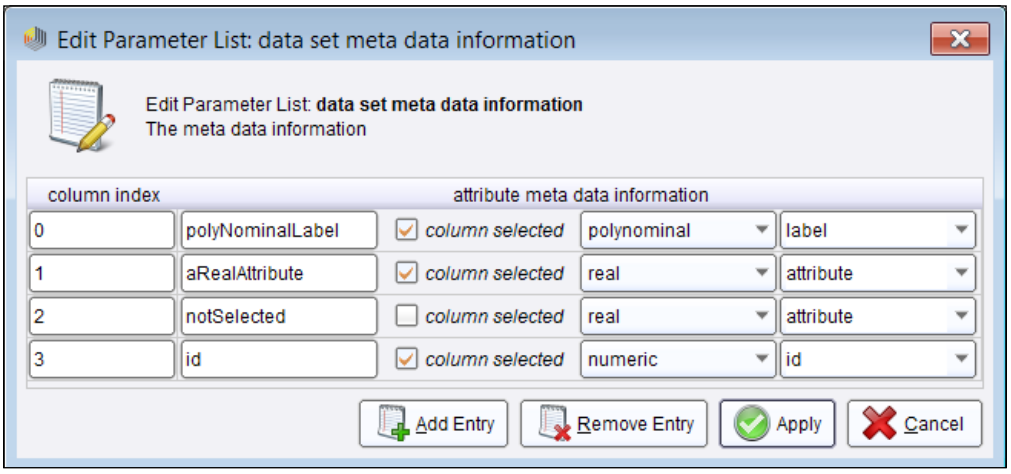
The wizard does a good job of inferring attribute types. But by using the first 100 examples (or whatever the configuration is set to), it can sometimes get this wrong if attributes that are beyond the first 100 have a different type. This particularly affects binominal values, real values, and dates. An incorrect guess can lead to errors when the whole file is read. For example, if the first 100 rows for a particular attribute contained two possible values, the wizard would guess that the type of the attribute is binominal. In fact, if there are more than two possible values, the attribute type should be polynominal. When the wizard encounters the unexpected value, it will—not unreasonably—show an error.

When the data to be read has hundreds of attributes, it can be tedious to use the GUI to correct any incorrect guesses. In this situation, the easiest approach is to edit the XML for the process directly.

For example, the following screenshot shows a small fragment of the XML for a `Read CSV` operator, which can be seen by enabling the **XML** view from the RapidMiner Studio GUI (this process is available as `ReadCSVMetaData.xml` in the files that accompany this book):




The following screenshot shows how this XML is shown in the GUI view. The relationship between the GUI view and the XML should be obvious from this example:



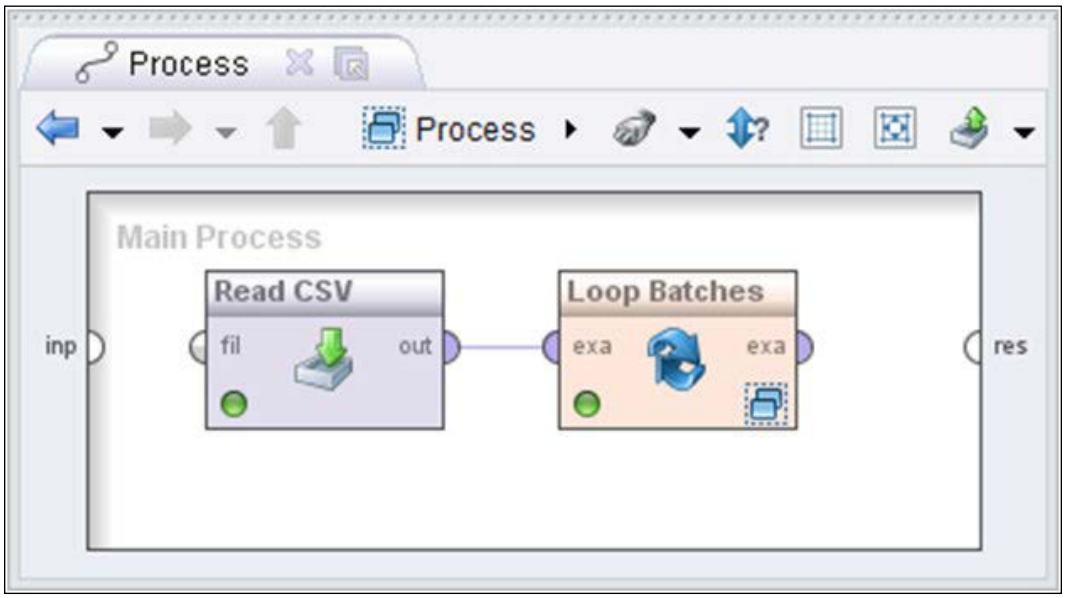

Editing the XML is straightforward and can be done by copying and pasting it into an external editor. When the finished XML is pasted back into the RapidMiner Studio GUI, it can be validated by pressing *F12* or by clicking on the relevant button in the **XML** view.

Generally, for very large imports, that is, those with many attributes, it makes sense to set all the variable types to `text` or `polynominal` and then use various operators to parse, validate, and set the type of the attributes. This usually avoids errors on import and allows focus to be given to processing each attribute in a more controlled way.

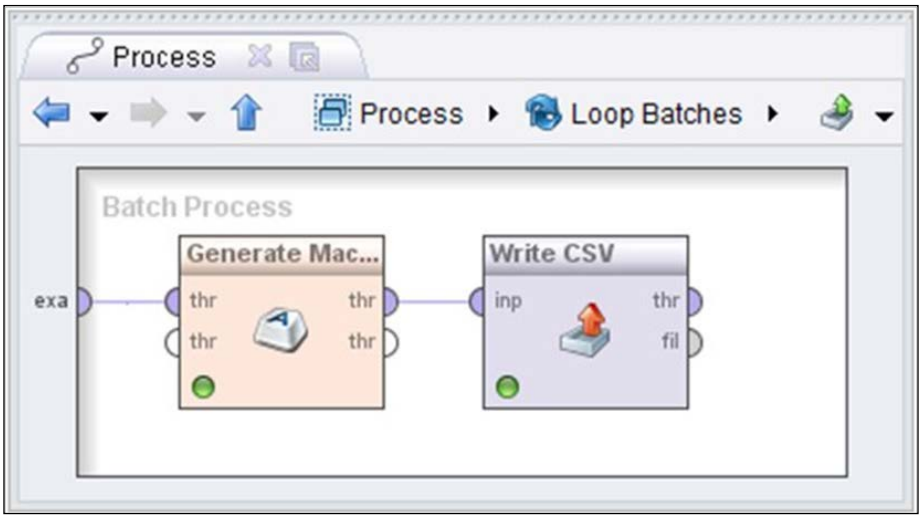
# Splitting files into smaller pieces

Processing a single large file that results in many attributes may exceed available memory, which is ultimately dictated by the computer on which RapidMiner is running. In this situation, it is sometimes possible to split a file into chunks using the capabilities of RapidMiner.

An example process that does this is shown in the following screenshot:



This process reads each line of the entire CSV file to be split into chunks. No processing is performed, thereby avoiding the overhead of having to create many attributes. The following screenshot shows the inner operators within the **Loop Batches** operator:




The **Generate Macro** operator increments a counter that is used to label the different files created with the **Write CSV** operator. Macros are initially defined in the context view. Macros are like global variables that are available to all operators and can be accessed and manipulated in numerous ways. Refer to the upcoming section *Using macros* for more information on macros in general.

The whole process titled `chopFiles.xml` is available from the code that accompanies this book. This file can be imported directly into RapidMiner using **Import Process** from the **File** menu. Alternatively, the XML can be pasted in its entirety into the **XML** view of the GUI. Pressing *F12* or selecting the **Validate Process** option from the **Process** menu will check the validity of the process and load it.

# Databases

RapidMiner is able to read from databases with ease. Most databases—such as MySQL, PostgresSQL, SQL Server, Sybase, Oracle, and Access—are supported. A **Java Database Connectivity** (**JDBC**) driver is generally available. What this means is that it will generally be possible to read data from virtually any database, but it is often the case that some of the specific configuration details can become complex. These configuration details are beyond the scope of this book, because they stray into the specifics of databases rather than RapidMiner. But, generally speaking, it is straightforward to deploy and configure a new driver to connect to virtually any datasource.

# The Read Database operator

The `Read Database` operator is the main operator used to access databases. It is generally a good thing to have data in databases because it eases the exploration process. Databases allow larger datasets to be stored, provide tools and a query language to allow data to be retrieved and updated, and impose a type on attributes that can make validation easier.

Database connections are created from the RapidMiner Studio GUI via the **Tools** menu item. The dialog is straightforward to follow and the test connection button allows a quick confirmation that the connection is working correctly.

It is possible to construct SQL queries using macros. This is vital to allow data to be selected specifically using parameters for the task at hand. The setup of the `Read Database` operator is slightly different from the norm but is straightforward once you have it working.

To illustrate this, a simple working example is given in the following points:

1. First, the database table that will be queried is a simple single table containing lines read from files. The name of the table is `details` and the important columns are as follows:
    - **line**: This corresponds to a line within a file
    - **file**: This corresponds to the file containing the line
    - **source**: This gives the source from which the file was copied
    - **length**: This gives the length of the line

A small fragment of the data would look something like the following screenshot, where the column headings correspond to the fields in the database and the whole table has a name, for example, `details`:

| line | file | source | length |
|---|---|---|---|
| This is a line | file42.txt | Desktop1 | 14 |
| This is some other line | file42.txt | Desktop1 | 23 |
| A short line | file43.txt | Desktop1 | 12 |
| A very long ... Line | file101.txt | Desktop2 | 121 |

2. To select data from this table, a SQL `where` clause might be as follows:

```
Select * from details where length > 50 and source like
'%Desktop2%'
```

3. This query will return the fourth row of the previous table.

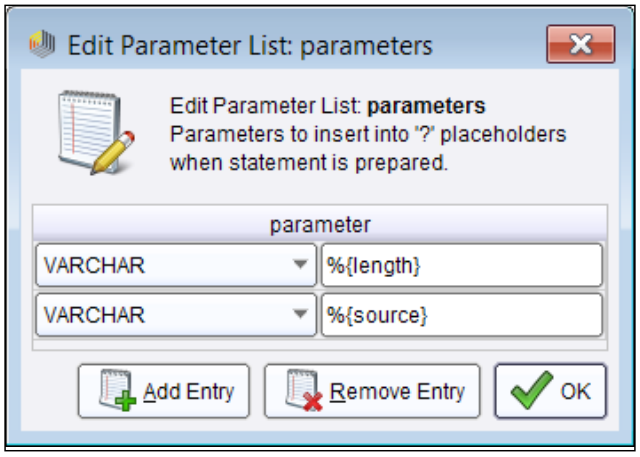
To configure the `Read Database` operator to submit a query like this requires the following steps:

4. First, the **Build SQL Query** dialog is used to enter the following query in the SQL Query box:

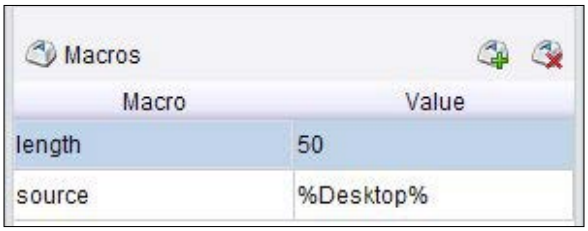
```
Select * from details where length > ? and source like ?
```

   The question marks correspond to the parameters.

5. Next, the **Edit Enumeration** dialog is used to create a mapping between the question marks and some macros, shown in the following screenshot. Note how the macro names `length` and `source` match the names within the query.



6. The final thing to do is to create the two macros (macros are discussed in detail in an upcoming section). This can be done using operators or can be done in the process context. The following figure shows the macros defined in the context view:




Note how the **source** macro value has percent signs around it. This ensures the syntax comes out correctly so that the final issued query is also correct.

Macros are an important part of RapidMiner and are used everywhere. Refer to the *Using macros* section in a moment to get an overview of these important entities.

# Large datasets

When reading large databases using the `Read Database` operator, all the data is read into memory. Clearly, this will cause problems as the data increases.

In my experience, large database imports should try to take advantage of the tools within the database rather than get RapidMiner to perform significant processing. Databases have many powerful tools to allow data to be transformed with views or summary tables.

# Using macros

Macros are like variables that can be passed between operators. They can be used as operator values to control execution. They can also be modified during processing as well as set to specific values based on attribute values or from properties of example sets as a whole. Processes have **Contexts** (use the GUI view menu item to see this view), where macros can be defined. Furthermore, macros can act as parameters to a process when it is run under the control of an external server, for example, RapidAnalytics.

Macros are referred to by name and are enclosed by the reserved characters `%{}`. The following expression shows a macro called `MyMacro` being referred to:

```
%{MyMacro}
```

The percent sign and curly braces delimit the macro, and during the execution of the process, the value of the macro is substituted. How the value is interpreted depends on the context and the value itself. String values are usually interpreted as attribute names, and during execution, the end result will be that the attribute's value will be used. Placing double quotes around the macro reference ensures that it is treated as a string.

Macros are used everywhere. *Chapter 4*, *Parsing and Converting Attributes*, uses them extensively while generating attributes, and this chapter is recommended to get more exposure to their use. This use continues in *Chapter 7*, *Transforming Data*, where more advanced data transformation using macros is performed. *Chapter 9*, *Resource Constraints*, uses them to measure process performance and *Chapter 10*, *Debugging*, mentions using them for debugging.

# Summary

In this chapter we learned how data in files and databases can be imported into RapidMiner. We explored some initial techniques for dealing with real data and the multitude of different formats that will be encountered. Armed with this information and some examples, you should be able to import most data. Of course, no book can ever cover every eventuality, but you should also be able to see how you could adapt the examples to your particular needs in the confident knowledge that a RapidMiner process can be constructed.

This leads to the next chapter, which addresses data visualization using RapidMiner. This is an important part of data mining in general, and in the context of data import, is a key way to determine whether what has been imported makes sense.

# 3
# Visualizing Data

Large amounts of data are difficult to understand; this chapter is about techniques that will help make sense of this data with visualizations. Visualization is vital at every stage of the exploration and mining process, and you will use these techniques time and again. Sometimes, the structure of the data needs to be changed to plot it effectively. This requires the use of RapidMiner operators to generate new attributes or pivot the data in new ways. Some of these techniques are a preview of what we will cover in the next chapter.

## Getting started

Whenever a RapidMiner process runs, the results are presented on the results perspective. This perspective is shown by selecting the appropriate option from the GUI menu or by pressing *F9*. Each result that can be viewed, such as example sets, models, log entries, or weights are displayed as tabs.

Selecting an example set presents a number of possible detailed views, including the Data View, Statistics View, and Charts View. The Data View gives a simple table summary of the example set. The Statistics View gives a statistical summary of the example set and details of the attributes and size of the data. The Charts View provides a large number of possible plotters, and these are selected from the drop-down list at the top left of the view. The RapidMiner Studio Charts view displays a useful thumbnail graphic for each of the possible chart types to make it easy to select the desired chart.

The simplest plot is the scatter plot and this appears as the first option in the drop-down list. This plot allows attributes to be shown on a two-dimensional grid with the x and y axes being determined by attributes from the example set. Individual points can be colored based on an attribute. Despite its simplicity, this plot is very useful to get an initial feel of the data. This is because it gives answers for questions such as: Are there obvious relations between attributes? Are there any points that look like outliers? and so on.

There are many other plotters that are available, and the RapidMiner Studio GUI selects the one that is appropriate to display the example set depending on its characteristics. It is always worth trying other chart types to see if these help to give a better understanding of the data. However, the following sections give some more detail of some specific techniques that can help.

# Statistical summaries

Having promised pictures, we start, however, by never underestimating a simple statistical summary. The Statistics View in the RapidMiner Studio GUI gives such a summary, and this is very useful to get a sense of how big the data is and what its range is. This view is available to show example sets when the Results view is selected. It is always worthwhile to take a careful look at this view to check that the attributes are of the correct type. Numerical attributes should have an average and a standard deviation that looks sensible and nominal values should have a full set of valid values and dates within an expected range. The Statistics View also shows which attributes have missing values.

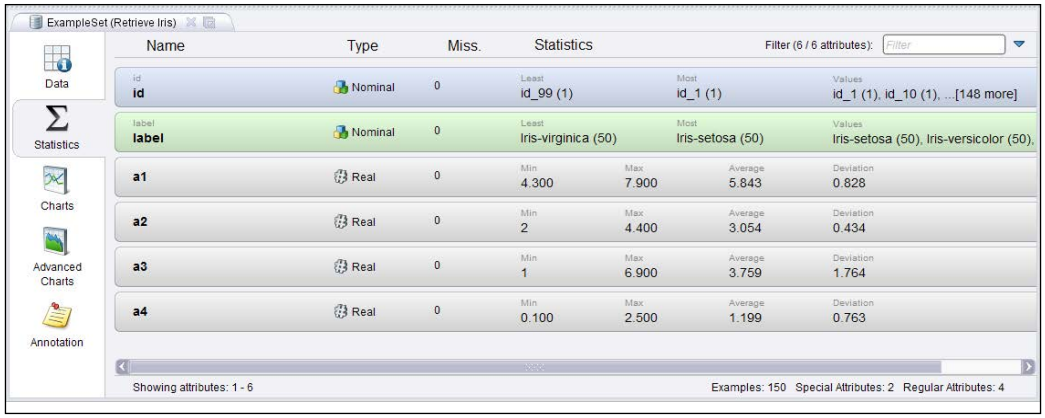
Sensibly, the GUI does not attempt to calculate statistics when there is too much data. In this situation, it is possible to calculate statistics in a process by using the `Extract Macro` operator. This operator is used to set a macro from some aspect of an example set that is being processed. By selecting the macro type to be `statistics`, and selecting one of the possible calculations such as `average`, `count`, `sum`, and so on, a macro value for an attribute can be calculated.

For large numbers of attributes this can be laborious, but it is perfectly possible to create a process using the `Loop Attributes` operator that loops over all attributes and generates a set of macros for each type of statistical measure. Then, these macros can be logged using `Provide Macro` as the `Log Value` operator and can be converted to an example set using `Log to Data`. An example process named `logAttributeDetails.xml` is provided with the files that accompany this book. This process shows the use of these operators to provide a statistical summary of some data that replicates the Statistics View.

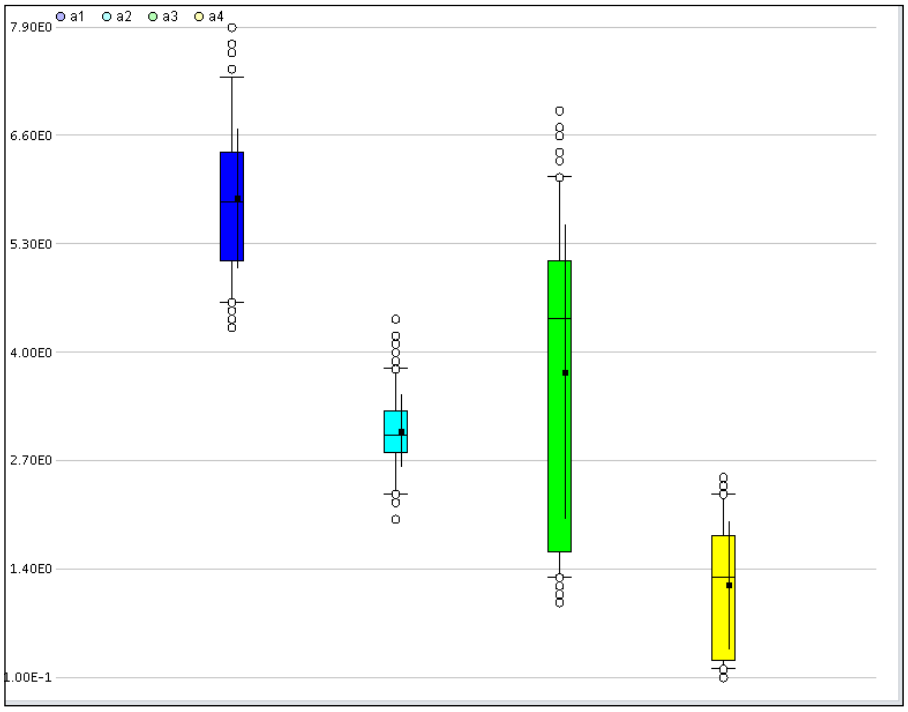
To identify missing attributes, the `Filter Examples` operator can be used. Set the `condition` class for this operator to `missing_attributes` and only examples with at least one missing attribute will be shown. An example of this operator being used is included in the `logAttributeDetails.xml` process.

This chapter is about visualization and one useful plotter to supplement the Statistics view is the quartile plotter, which can be a useful way to summarize numerical attributes in particular.

For example, the following screenshot shows the Statistics for the **Iris** dataset:




The same data plotted using the quartile plotter is shown in the following screenshot:



This plot can be obtained by holding down the *Ctrl* key and selecting multiple attributes. The plot shows the mean with the help of the dot within the colored area and the range of the standard deviation is shown by the vertical line offset toward the right, inside the colored area. The colored area itself represents the 25th to 75th quartiles and the 10th and 90th quartiles are the horizontal lines above and below (outside) the colored area. Finally, the range is represented by the dots at the extremes. Examination of the figures will confirm that the numbers match the screenshot. For example, the maximum for attribute **a3** (the third from the left) is 6.9 and this is shown on the graph as the highest point for that attribute. Similarly, the average for the same attribute is 3.79 and this is represented by the dot within the shaded area.

The metadata gives information about individual attributes in isolation. Another important aspect is to understand how attributes relate to one another. This aspect is covered in the following section.

# Relationships between attributes

The relationships between attributes are important to understand and visualization can help in understanding these relationships. Attributes may be correlated with one another and viewing this may help shed light on the data and new ways to perform further processing to help understand it and make progress towards the overall objective.

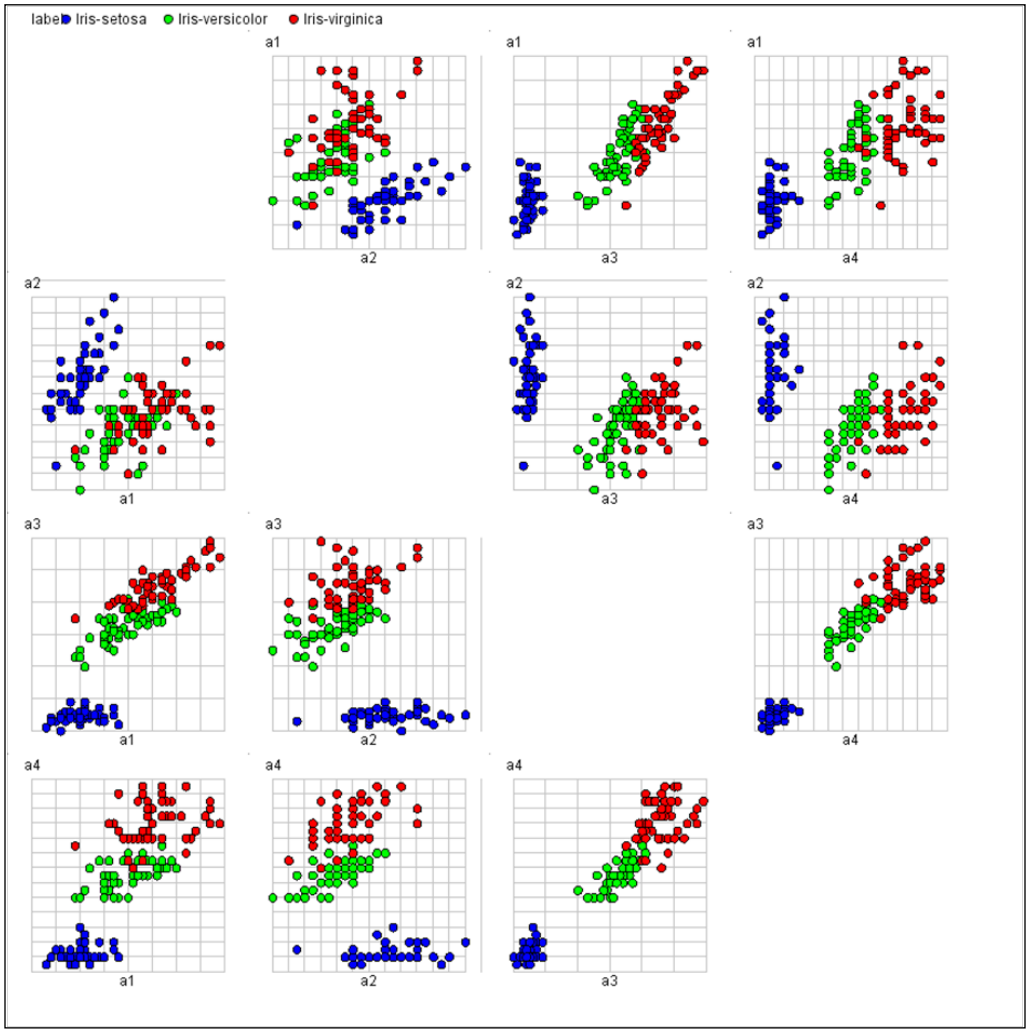
There are many ways to show how attributes relate to one another. These include scatter plots, 3D scatter plots, parallel plots, deviation plots, and quartile plots, which are described in the following sections.

# Scatter plots

To start answering the question about the relationships between attributes and examples, the scatter plot is a quick summary method which has already been mentioned earlier. A good next step is the scatter matrix plotter, which summarizes all possible pair-wise permutations for a given attribute. This is used to determine the color for the points.

An example using the Iris dataset is shown in the next screenshot. The idea is to spot patterns in the data and see if it is possible to explain them. A general rule for classification is to see if groups of colored points representing labeled data can be separated by simple lines; in effect the observer is becoming a support vector classifier. By doing this, we can gain a better understanding of the data.

For example, the upper-right corner of the screenshot shows a graph of **a4** on the x axis and **a1** on the y axis. The points are colored based on the class of the example. The graph shows that there is an approximate correlation between **a1** and **a4**, and that low values favor one class very clearly and higher values favor the others with a reasonably clear threshold. Understanding this and deciding what it means for the data mining task is an important step.



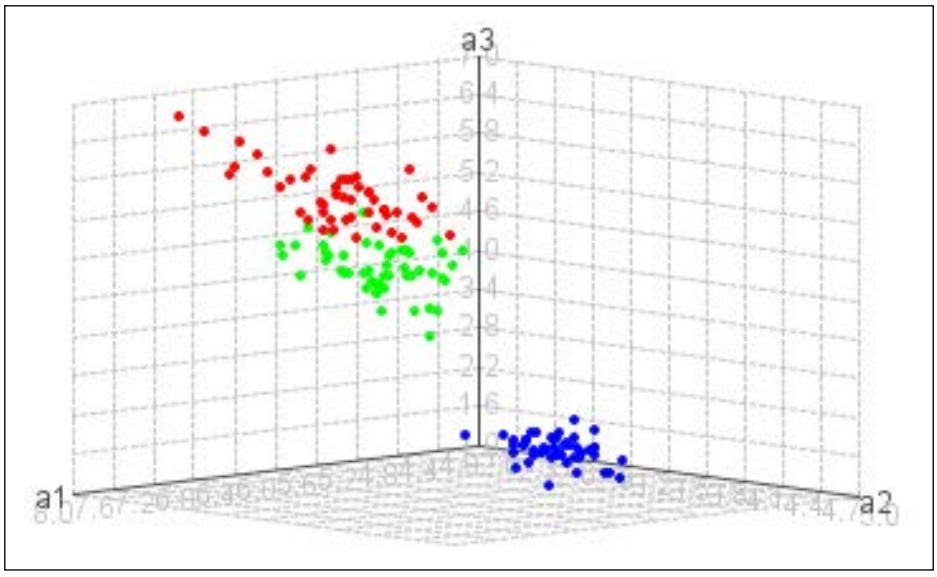
The jitter parameter allows each point to be given a random nudge. This allows points that are very close together to be seen more easily, and this gives a sense of the density of the points in space. The way to understand how jitter works is to imagine that all the points that share the same x and y attribute are stacked one on top of another and you are looking down on them from a great height so only a single point is seen. Applying a small jitter causes the co-located points to become visible.

Of course, real data is never this easy and it can quickly become impossible to see the detail if there are too many attributes. 20 attributes plotted this way would also be difficult to visualize. In these situations, it may be appropriate to select groups of attributes using the `Select Attributes` operator to see how the attributes within these groups interact with one another. For example, if there are 20 attributes, selecting the first 10 with this operator and plotting them using the scatter matrix plotter will show how these 10 attributes relate to one another. The next 10 could then be selected and plotted. Of course, the interactions between groups selected in this way would not be seen, so care should be taken or else the permutations would quickly get out of hand.

Sometimes, however, two dimensions are not enough and so RapidMiner provides us with the scatter 3D color plotter.

# Scatter 3D color

Again, using the Iris dataset, a 3D representation is shown in the following screenshot:



Large datasets can be difficult to view using this plotter because there is a lot to plot and this can be beyond the capabilities of the computer running the GUI. Sampling using the `Sample` operator is one possibility in this situation. A process named `scatterPlotAndSample.xml` is included with the files that accompany this book. This shows the Iris dataset and also a 100,000 point dataset which has been sampled. Comparing the sampled and unsampled data on a plot gives a sense of whether the sampled data is still representative of the unsampled data, and therefore, whether it is useful to help understand the data or not.

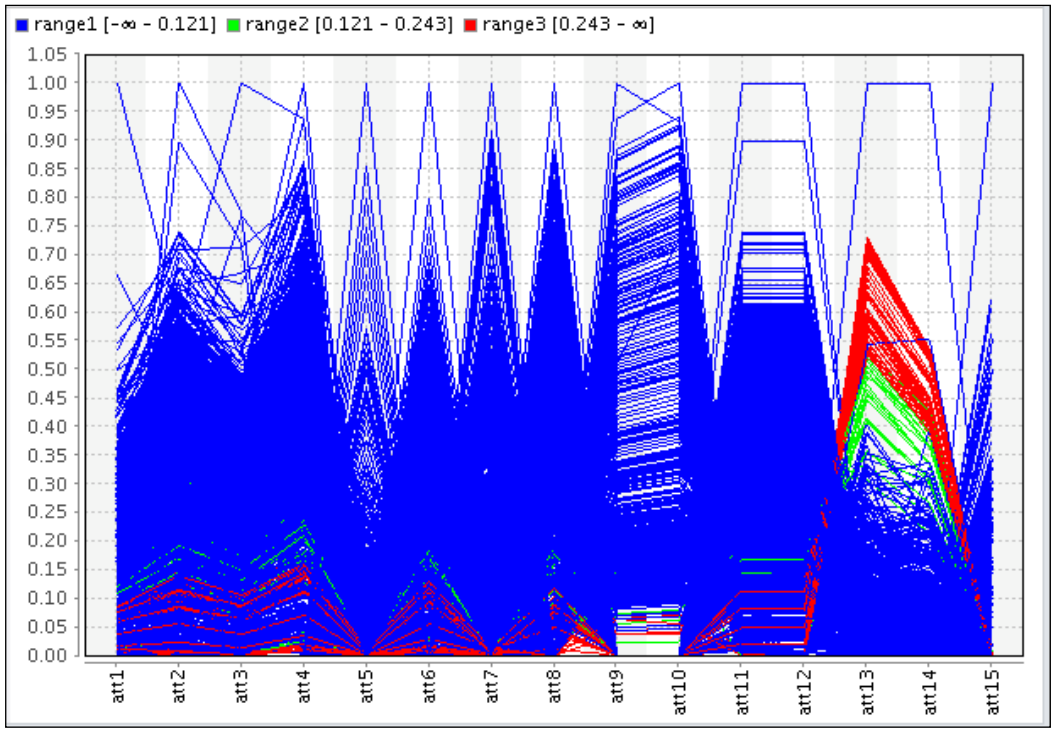
An alternative is to use the parallel and deviation plotters described in the next section.

# Parallel and deviation

The parallel plotter is used to see the relationships between attributes when there are many attributes and examples. The plotter lists each attribute on the x axis and deviation plots, then it plots the value of each attribute for each example. One of the attributes is chosen as the color and the line is colored based on the value of the example within the example set.

Some data is provided along with this book to allow the following illustrations to be recreated. The data is contained in a file called `DataToVisualize.csv` and it can be imported using the `Read CSV` operator. Be sure to set the role for the attribute **att16** to `label`, the attribute `id` to have the role `id`, and the attribute `date` to have the role `date_time` using the `Set Role` operator or by getting the parameters correctly set when importing the CSV file. A sample process to read this CSV file with the correct parameters is provided. It is called `readDataToVisualize.xml`. This process is very straightforward and the resulting example set contains 3,848 examples with 15 regular attributes called **att1** to **att15**, one label attribute called **att16**, an ID attribute, and a date attribute. The label attribute is nominal and has three values based on an original value of **att16**; these are **range1**, **range2**, and **range3**.

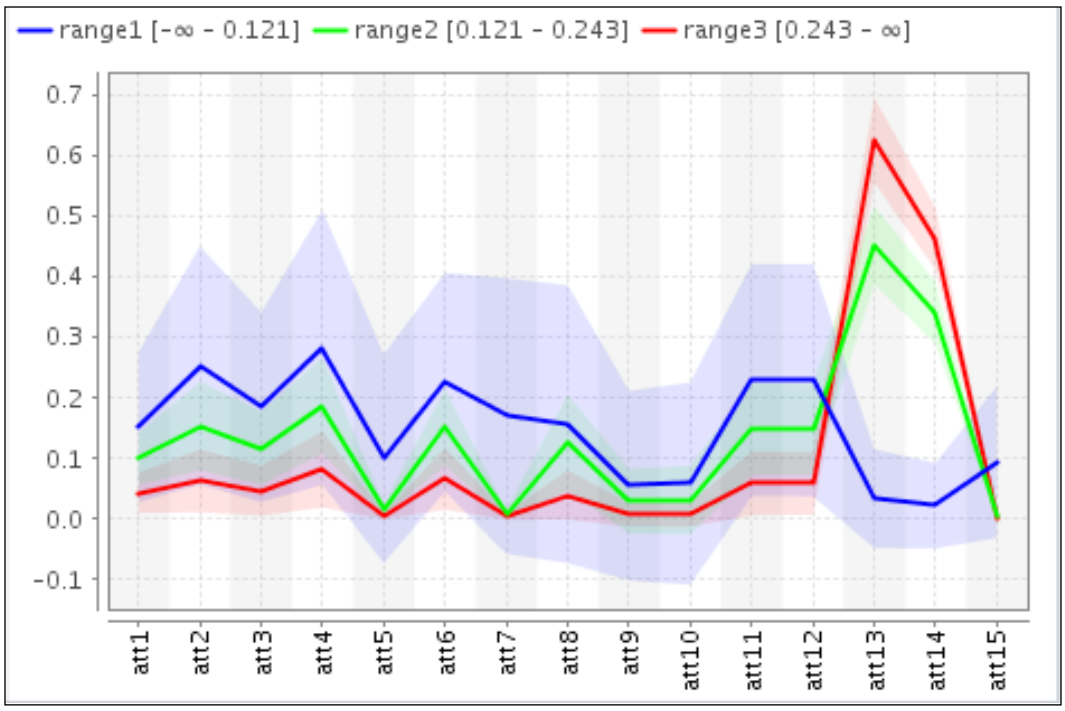
An illustration of a parallel plot using this real but obfuscated data is shown in the following screenshot:




There are 16 attributes and 3,848 examples in the example set. This means that there are 3,848 different lines on the graphic, one for each example. **att16** is chosen as the color so that when it has a high value, the line is colored red and when it has a low value it is colored blue. The graph has been locally normalized by checking the local normalization check box on the plotter, so that the range of the attributes is between zero and one. In monochrome, this may not show up very well. So, the area of focus is the the right-hand side of the graph, where **att13** and **att14** are shown as having a relatively higher value at the same time as **att16** has a higher value (as indicated by the color of the lines).

This indicates correlation between these attributes. The data can be systematically investigated to determine relationships and as before, questions will be raised about the data that will give a greater understanding when answered.

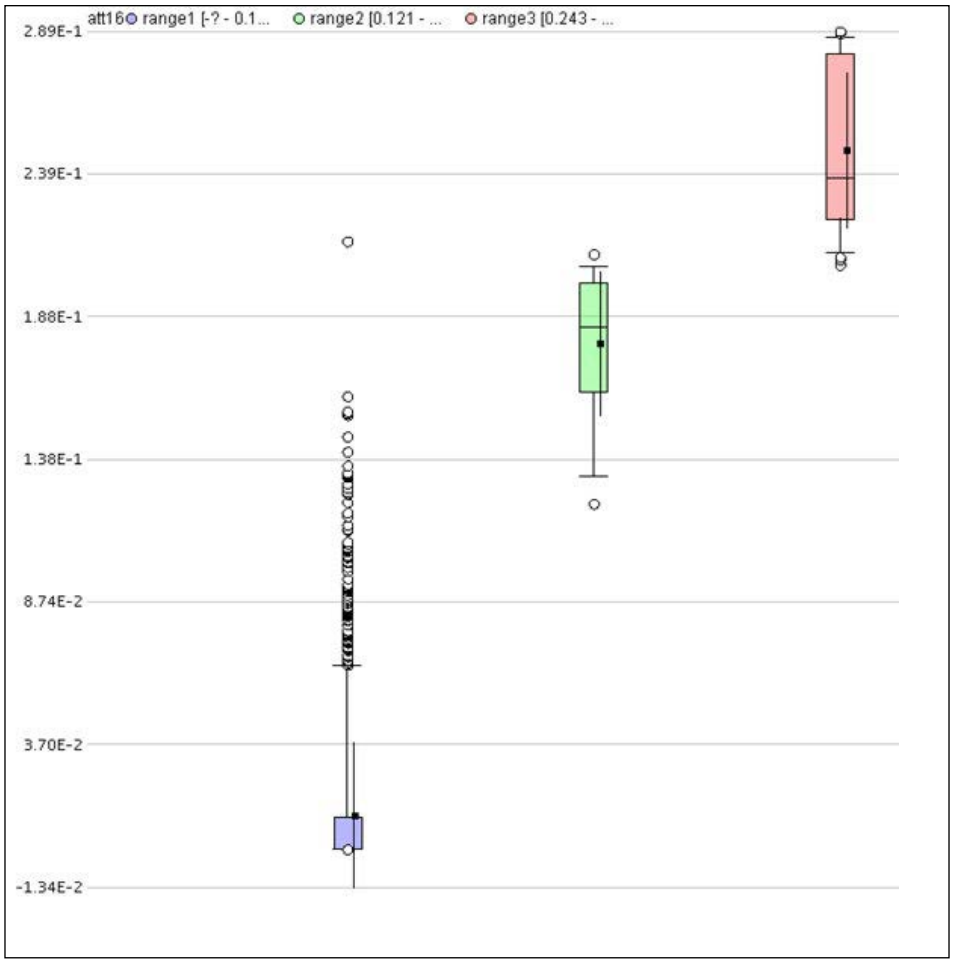
This plotter works well with larger numbers of attributes (that is, more points on the x axis). For large numbers of examples, however, the number of lines can make the display look cluttered. One way to reduce this is to use the Deviation plotter. This plots an average value for each attribute with different lines for different values of another chosen attribute. It also includes an upper and a lower bound of one standard deviation. One attribute is chosen as the color, but this must be a nominal value to get multiple colors and different averages—and hence, different lines. An example plot is shown in the following screenshot:




This is the same data that we presented earlier, and it clearly shows the relationship between **att13**, **att14**, and **att16**. The removal of the clutter makes it easier now to see for the first time that there is perhaps a negative correlation for most of the other attributes against **att16**.

# Quartile color

To get a sense of the range of data points for attributes as a function of another attribute, the quartile color plot can be used. This is similar to the quartile plotter described earlier except that the color is set by another attribute that must be a nominal. Focusing on **att13** and **att16**, a quartile color plot is shown in the following screenshot (a small number of outlying points have been removed to allow the image to display more clearly in this book):




**att16** dictates the number and colors of the bars that are drawn. The left y axis shows the range for **att13**. The graph shows that higher values of **att16** correspond to higher values of **att13**. The outliers are significant and it is possible to see many that overlap with the middle range value of **att16**. This provides evidence of outliers in the data and is worth investigating in order to determine the root cause.

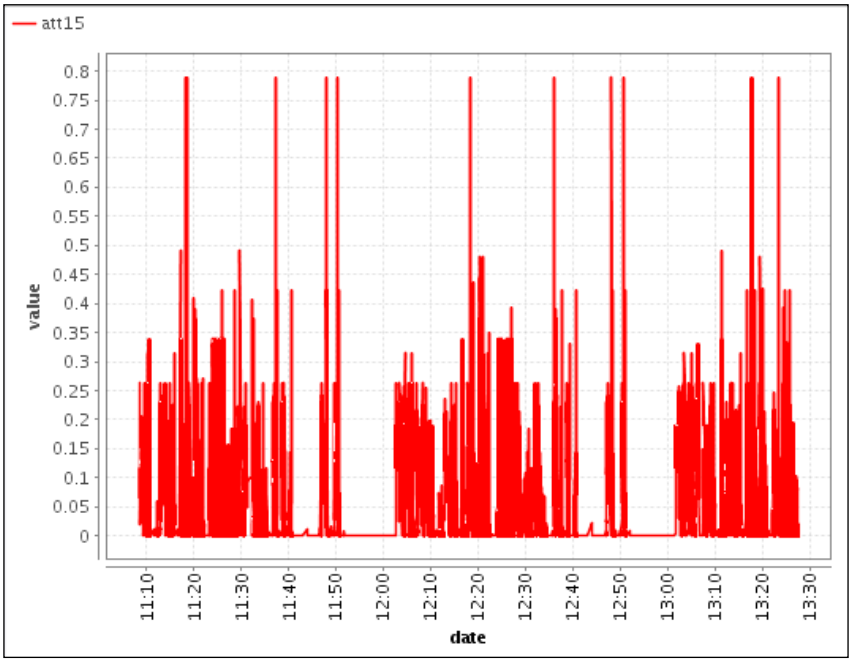
When exploring real data, a systematic investigation would be done with all the attributes to get a sense of how the attributes depend on one another.

# Time series data

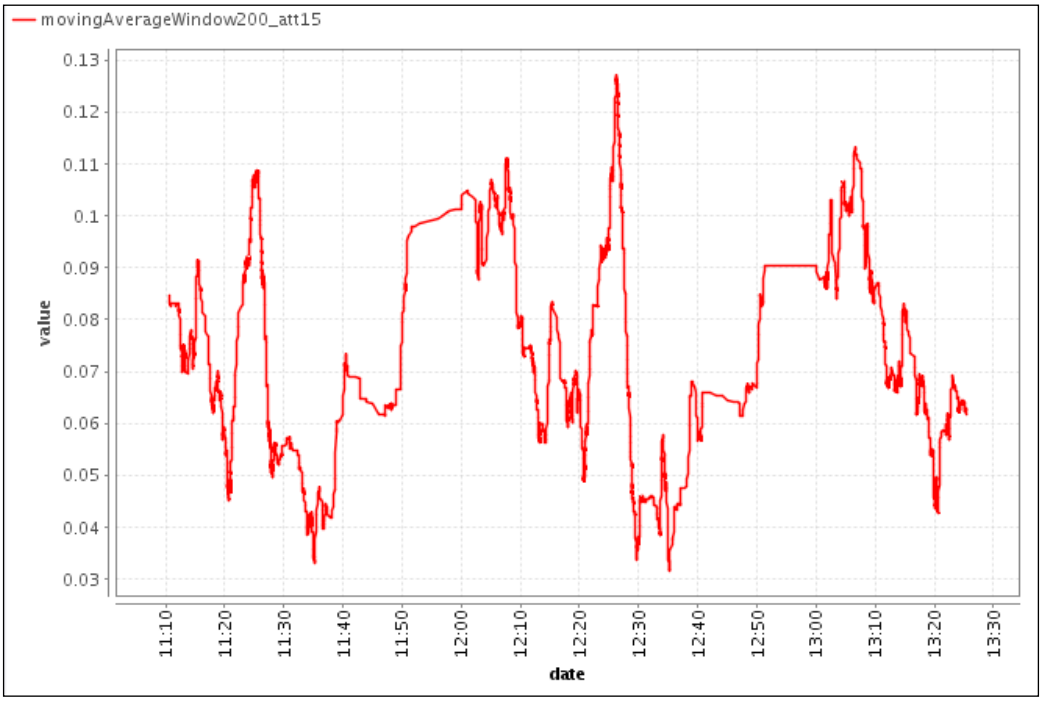
Real data is often in the form of a time series and this makes visualization difficult when there are many attributes over an extended time period. There are a number of plotters as well as some summarizing techniques that can help. The following section gives some examples. For one of the sections, it is necessary to download and install the Series Processing extension from the RapidMiner marketplace. This is done from the RapidMiner Studio GUI by navigating to **Help | Updates and Extensions**. From here, type `Series Extension` in the search box and once the results are returned, select the entry and follow the onscreen instructions.

# Plotting series

The series plotter and series multiple plotter simply plot the series data. Again, using the process `readDataToVisualize.xml`, the following graph shows **att15** plotted as a function of time. The graph shows that there is some structure as a function of time but it can be difficult to interpret because there are so many data points:

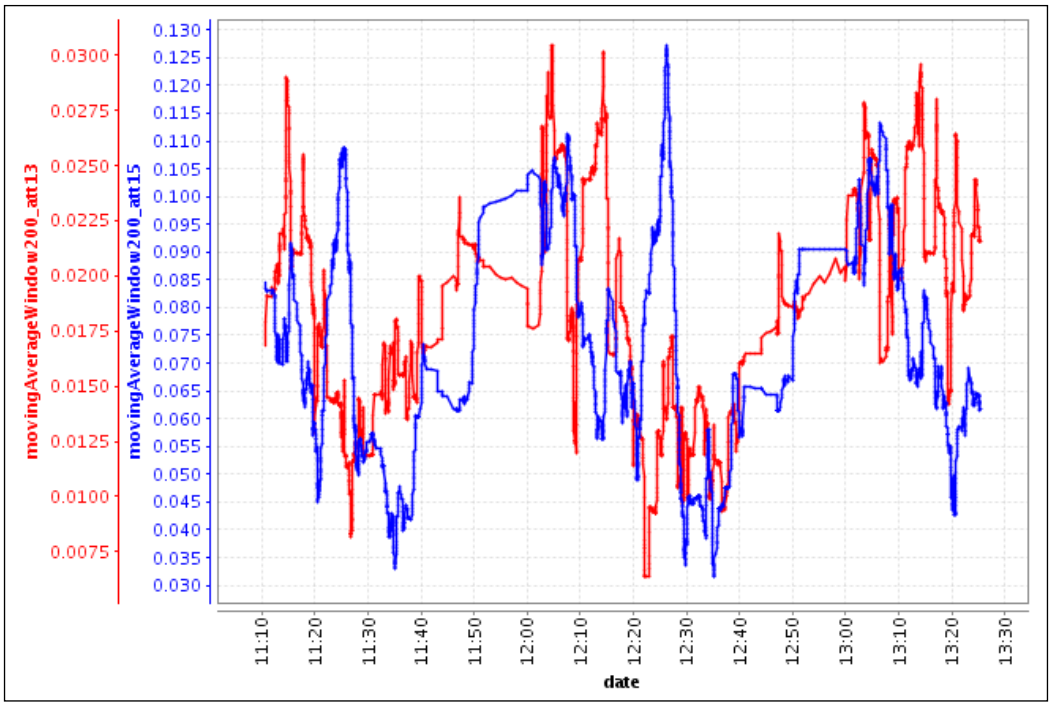


One approach to simplify this is to use the `Moving Average` operator to smooth the data out. This operator simply calculates a moving average for an attribute, given a window size, and creates a new example in the example set. An example of using the `Moving Average` operator with a window size of 200 is shown in the following screenshot:




A process called `MovingAveragePlotter.xml` is provided with this book to generate the result shown in the previous screenshot. This process generates moving averages for all the attributes in this data and does various tidying activities to make the names of the generated attributes easy to understand.

It is possible to display both series on a single graph using the series multiple plotter, and recalling that attributes 13 and 15 showed evidence of correlation from previous results, the two moving average plots for these attributes is shown in the following graph:

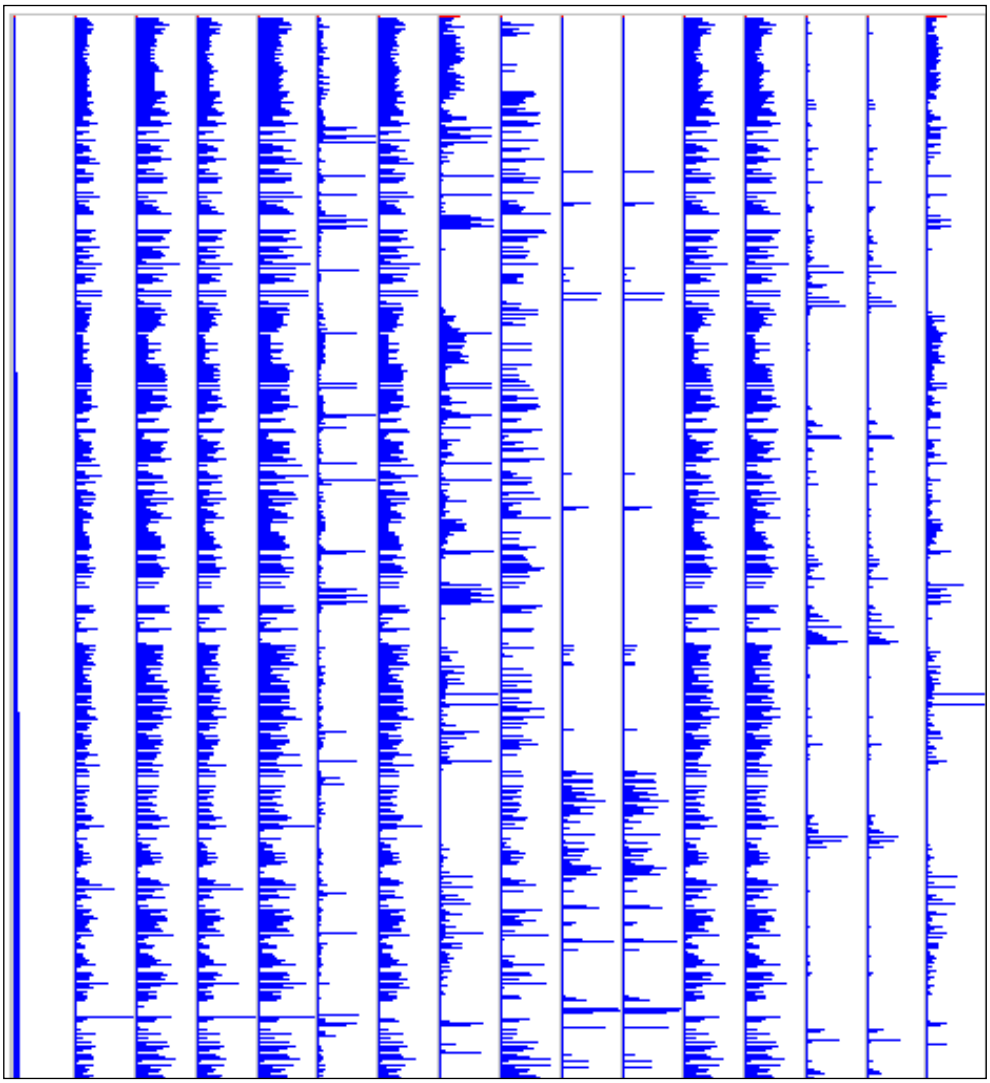



This graph gives the evidence that there is a time dependent variation that correlates between the attributes, although it is not exact. This illustrates the importance of a good visualization when trying to understand the data.

# Using the survey plotter

When the number of series to plot becomes very large, it can become unwieldy to use the series plotter. In this case, the survey plotter can be useful.

The best way to understand how this plotter works is to look at an example, like the one shown in the following screenshot:



This plot can be recreated using the `MovingAveragePlotter.xml` process. Note that for reasons of space and readability, the previous plot shows a small portion of the survey plot in this case. The plot has the first column for the plotter set to `date` and the color column is set to the `color` attribute that is generated by the process.

Each vertical plot shows how one attribute varies as a function of another. If the data contains `date` as an attribute, then sorting by date produces a time series view of each of the other attributes. This is what is shown in the previous screenshot. Each vertical represents a time series for a different attribute, with time increasing downwards. The attributes in this example start with the date at the left, followed by **att1** to **att15** (both inclusive). Think of this display as a 90 degree clockwise rotation of a time series.

This view brings out the relations between attributes. It is extremely clear which attributes correlate with one another and within the context of exploratory data analysis, this raises questions that, once answered, will help the data to be understood better. In the previous screenshot the following attributes appear to be correlated: **att1**, **att2**, **att3**, **att4**, **att6**, **att11**, and **att12**. Furthermore, the same can be said for **att9** and **att10**.

By setting the color of the survey plot to be an attribute, the series are colored based on the value of this attribute. This allows correlations between it and other attributes to be seen.

The end result of using this plotter is a better understanding of time series as well as more detail about how a multivariate time series behaves and the possibility of getting an insight into how attributes relate to one another.

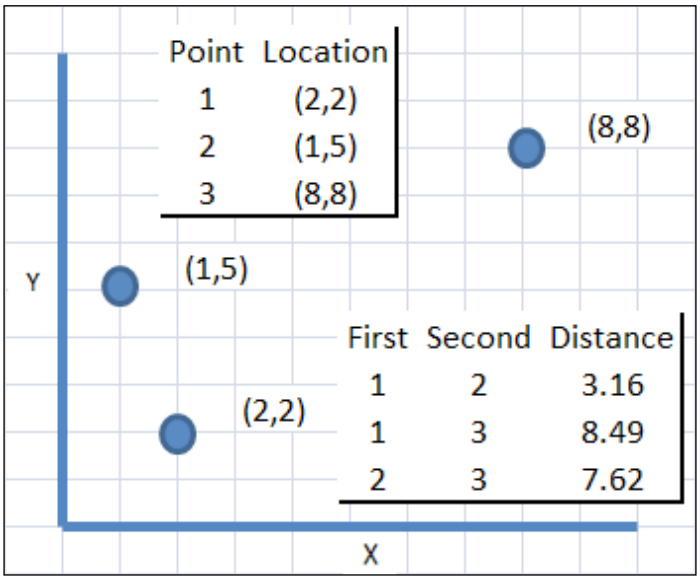
The relation between attributes is one aspect of understanding through visualization. Another aspect is how examples relate to one another and this is covered in the next section.

# Relations between examples

Understanding how examples relate to each other is important. This is because examples that are close to one another may be duplicates, so it is worth considering and understanding how they arise and what needs to be done, if anything, about them.

Closeness in this context is some sort of distance measure such as Euclidean distance or cosine similarity. Many possible distances can be calculated using RapidMiner and a brief explanation of Euclidean distance is given in the next section.

The following screenshot shows three data points in two dimensions:




The points are labeled **1**, **2**, and **3** and the Euclidean distances between them are shown in the inset table. The Euclidean distance between the **First** and **Second** point is given by the following equation:

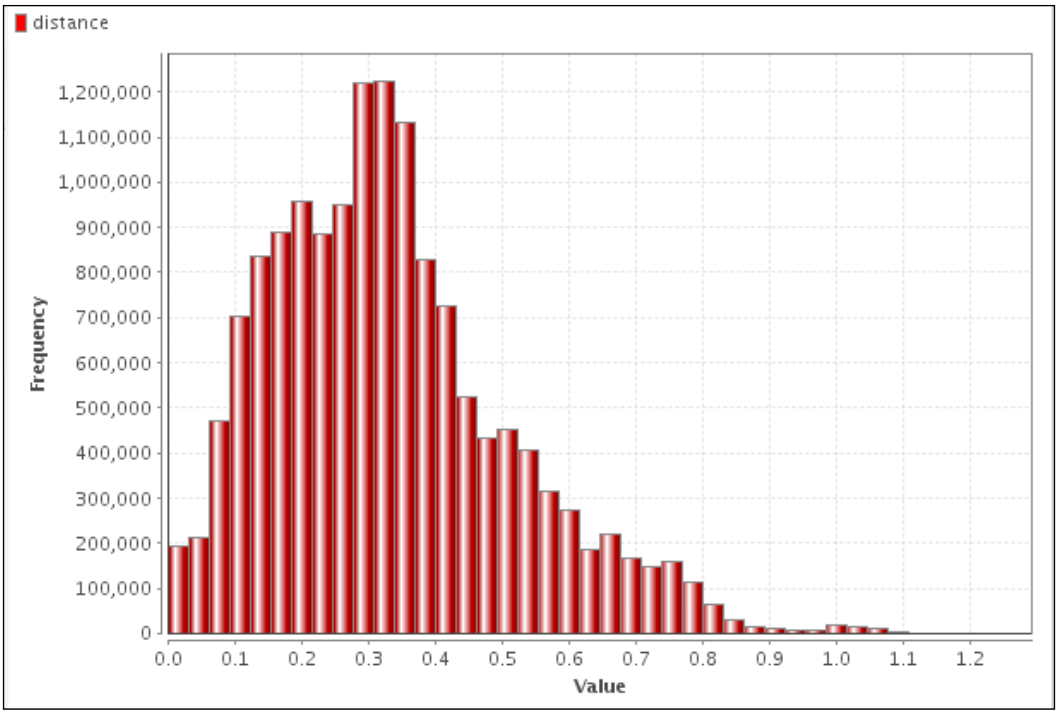
$$\sqrt{(2-1)^2+(2-5)^2}=\sqrt{10}\approx 3.16$$

Intuitively, we can see that the distance between points **1** and **2** is smaller than their distance from **3**. This gives the idea that these two points could be more closely related than the third, and this information is valuable to help us understand the data.

This approach extends to higher dimensions, but it quickly becomes impossible to visualize when there is a lot of data. There are two approaches described here that can help us with this. The first of these involves plotting a histogram of the distances.

# Using histograms

As an example, the following graph shows all the pair-wise distances for the `DataToVisualize.csv` data provided with this book. Simply run the `DistancesPlotter.xml` process provided. This process uses the `Data to Similarity` operator to create data for this histogram view. Using this example set in the results view, select the histogram plotter and plot the **distance** to create the following screenshot:



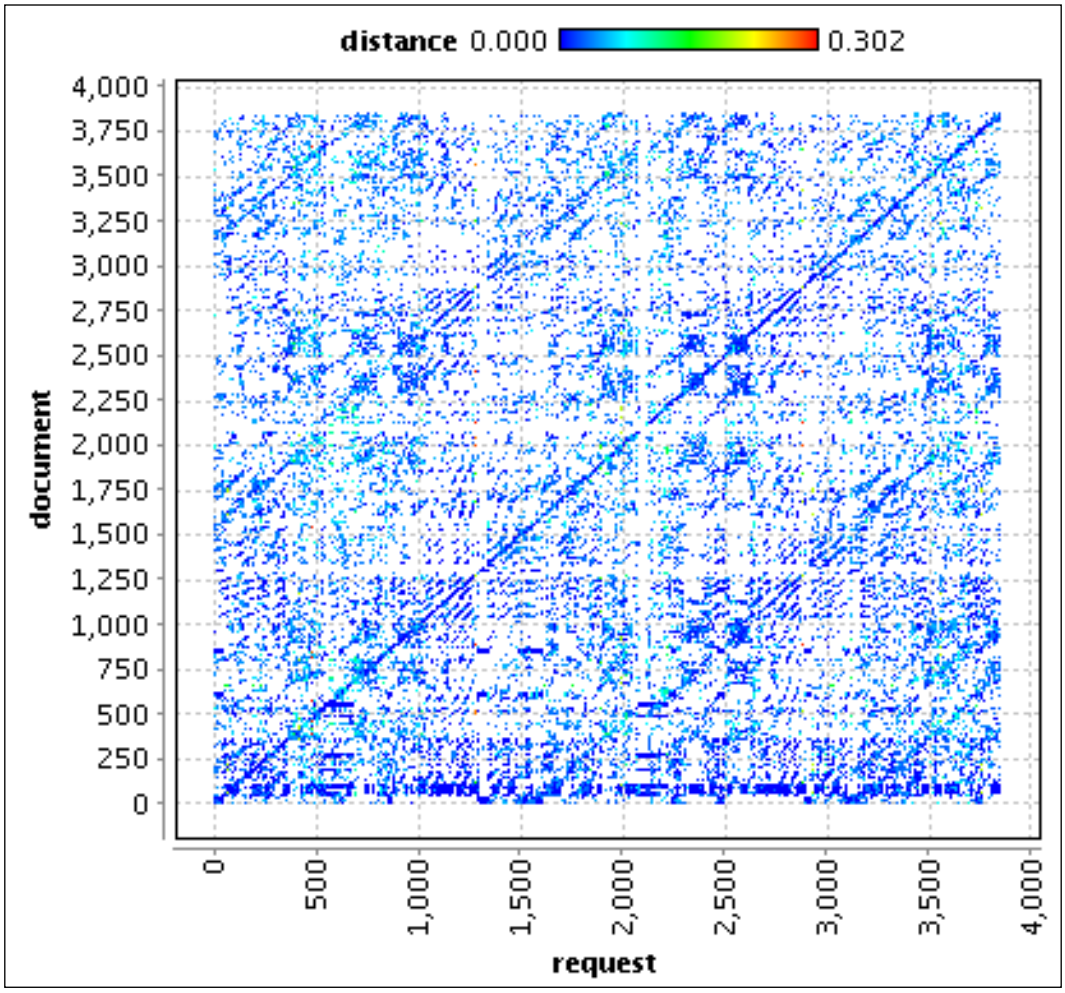

This is a large dataset containing nearly 15 million pairs and it may be the limit of what can be realistically displayed on the RapidMiner GUI. Nonetheless, examination of this shows that there are no outrageous outliers and the peaks at distances of **0.2**, **0.35**, and the small peak at **1.0**, indicate interesting things in the data.

# Using block plots

An alternative way to display relations between examples is to display them in a grid, with one set of examples represented along the x axis and the other set along the y axis. The intersection is then colored to represent the distance between the examples.

Calculating distances can be done with the `Data to Similarity Data` operator (as done with the histogram in the preceding diagram) but a better alternative is the `Cross Distances` operator. This operator provides a method for selecting the nearest or furthest distances, which can be vital if the number of pairs of attributes is very large because too many pairs will not be displayable in the RapidMiner GUI.

The following screenshot shows such a plot. This is the block plotter from the result of the `Cross Distances` operator within the process `DistancesPlotter.xml`. The x axis is set to **request**, the y axis is set to **document**, and the color is the **distance**.




There is considerable structure in the data. Given that this data has a time series element, the graphic shows how examples are changing as a function of time. The most interesting things that stand out are the diagonal lines that give evidence of a repeating pattern (every 26 minutes, interestingly). The horizontal lines at 550, 1,300, 2,100, and 2,900 are also interesting and need to be understood.

# Summary

This chapter has given an overview of some of the techniques used to visualize data. In addition, it has given some context to allow a visualization technique to be chosen. RapidMiner Studio allows quick manipulation of data to allow it to be enhanced, so that it can be visualized better. This chapter has given us some ideas about this.

It is particularly true that in the case of visualization, there is tremendous scope for creative presentation and exploration, and this chapter is only a start. You will find yourself visualizing data all the time.

The next chapter discusses parsing and converting attributes into different forms or into new attributes. This is an important part of visualizing data since it is sometimes necessary to do this to make visualizations more appealing. Generating new attributes is also an important technique in general as part of extracting features from data to help get the most value from it.

# 4
# Parsing and Converting Attributes

Having read the data, in order to understand and explore it more effectively, there is usually a need to convert attributes into different formats, parse them to extract additional information or features, as well as create additional attributes to help represent the data in new ways for new insights.

For example, an extremely common task is converting date and time values into a common format so they can be manipulated. Another example is extracting file names from file paths or domain names from URLs. Furthermore, combining two or more attributes with summary information from the rest of the data or from external sources to make a new attribute may help make a predictive model more powerful.

For real data, it is important to think about unseen data that could be encountered, since it is important to handle this correctly in order for models to function accurately.

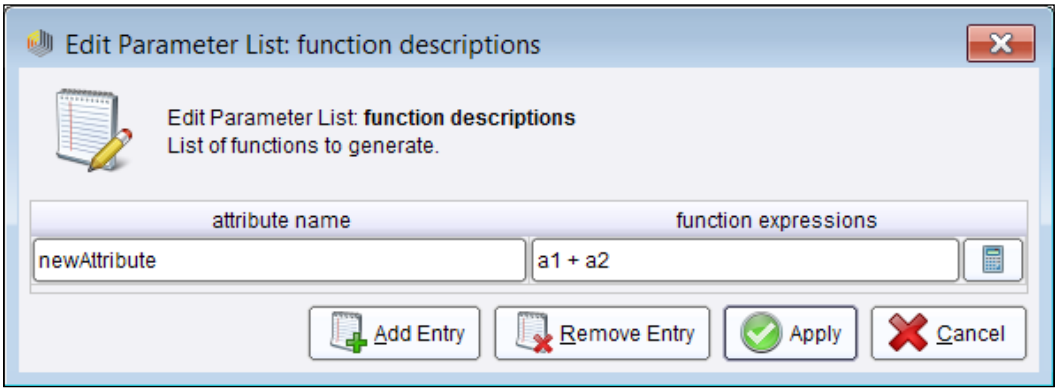
Some operators automatically generate attributes with names reflecting the operations performed. These attributes often need to be renamed in order to be used later on because the attribute names contain characters that are interpreted as mathematical operations. The values themselves often need global search and replace operations on them as part of the imposition of a data dictionary across larger projects.

RapidMiner Studio has operators that can be used individually or together to achieve the previous objectives; they allow attributes to be created and renamed with values derived from other attributes, and they allow values to be modified in systematic ways.

# Generating attributes

The `Generate Attributes` operator is used very frequently. It allows new attributes to be generated from other attributes, constant values, macros, and built-in functions. The way to think of this operator is to regard it as an automatic loop over all the examples in an example set. The newly generated attribute is added to all the examples. If the value of the new attribute is derived from the values of other attributes, the single value for the new attribute is taken from the values of the other attributes of the current example. This means that if macros are to be used, they must be defined before the generation of new attributes.

The simplest expression to create a new attribute is shown in the following screenshot:




In the previous screenshot, **a1** and **a2** are the existing attributes and the new attribute created, called **newAttribute**, is the sum of these for the example being processed. A subtle point is that the type of the new attribute is worked out dynamically. If new test data is encountered and one of the attributes is a polynominal while the other attribute is a number, the result will be an error. If both attributes are polynominals, the result will also be a polynominal. Data import should take care of getting the type of data correct, but it is worth bearing in mind for unseen data.

Macros can be used. For example, `%{m1} + a2` will add the value of the macro m1 to the attribute a2. If the macro is not a number, the expression `%{m1} + a2` would treat the value of the macro as an attribute name. For example, if the macro m1 has the value a1, the previous expression would become `a1 + a2`, so the end result would be the sum or concatenation of attributes a1 and a2 depending on their types. To force the macro to be treated as a string, place it in quotes. With the macro m1 equal to the string a1, the expression `"%{m1}" + a2` would evaluate to `"a1" + a2`.

The following table summarizes these different situations:

| m1 (a macro) | a1 (an attribute) | a2 (an attribute) | expression | result |
|---|---|---|---|---|
| 1 | 2 | 3 | %{m1}+a2 | 4 |
| a1 | 2 | 3 | %{m1}+a2 | 5 |
| a1 | one | two | %{m1}+a2 | onetwo |
| one | one | two | "%{m1}"+a2 | onetwo |
| a1 | one | two | "%{m1}"+a2 | a1two |

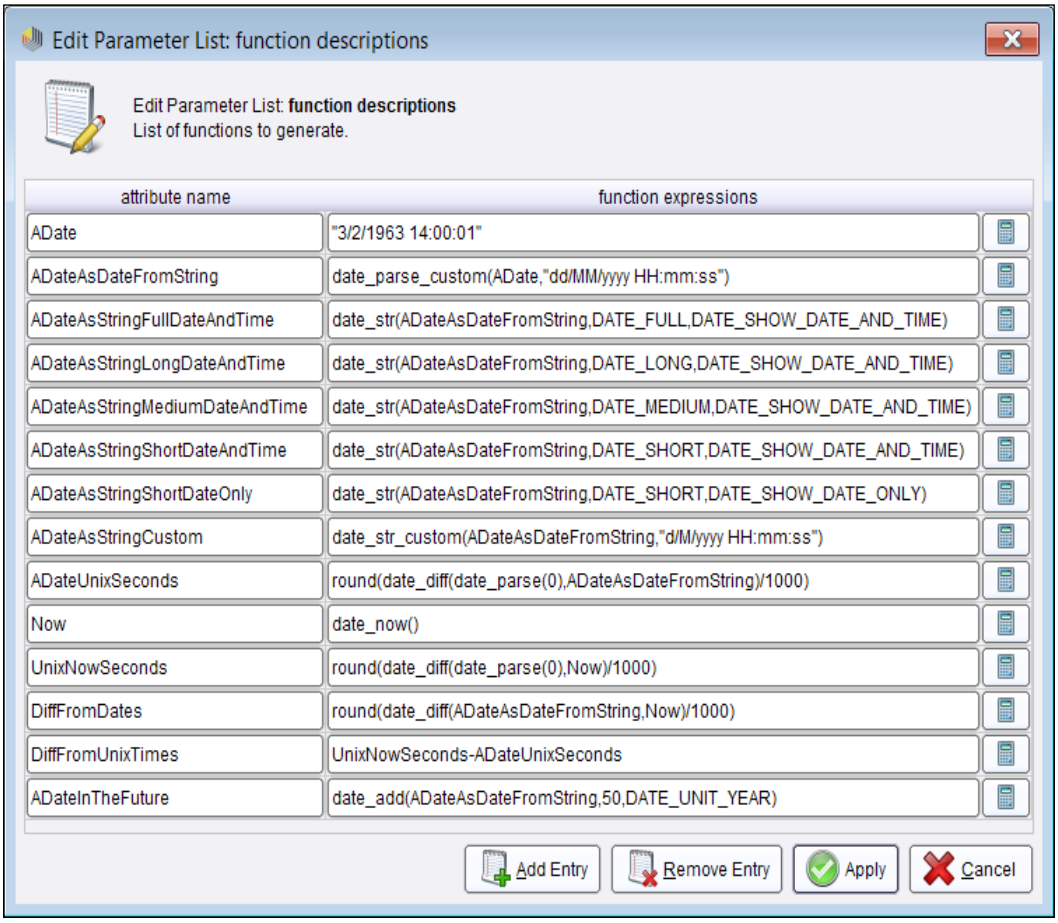
A process called `generateAttributeExamples.xml` is available with the files that accompany this book, which illustrates the previous example.

A large number of functions are available and there is help available from the operator description as well as the Edit Expression dialog, which is accessed by pressing the button to the right of the **function expressions** in the previous screenshot. How these functions work is usually obvious from the name. There are subtle points relating to some of the functions and these are described in the following sections.

# Date functions

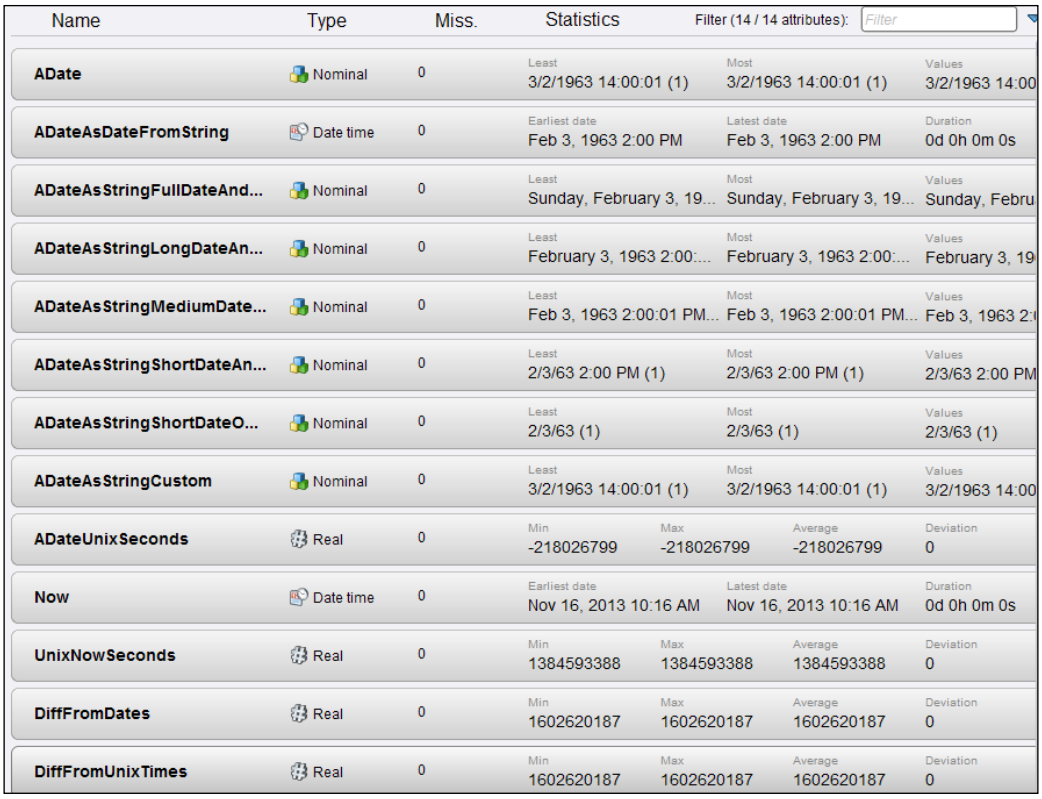
The functions in the `Date` group manipulate dates and can convert strings to dates and back. Some detailed examples are given in the next screenshot because dates can be a source of difficulty when encountered in real data.

The following screenshot shows some examples of date calculations within the `Generate Attributes` operator. For reference, the `dates.xml` process is available with the files that accompany this book.




Various calculations are performed on the date provided as the first attribute. Note how the result of a previous step can be used in subsequent steps.

The result of running this is shown in the following screenshot, which shows the `meta-data` of the created attributes:



| Name | Type | Miss. | Statistics | | | |
|---|---|---|---|---|---|---|
| ADate | Nominal | 0 | Least: 3/2/1963 14:00:01 (1) | Most: 3/2/1963 14:00:01 (1) | Values: 3/2/1963 14:00 | |
| ADateAsDateFromString | Date time | 0 | Earliest date: Feb 3, 1963 2:00 PM | Latest date: Feb 3, 1963 2:00 PM | Duration: 0d 0h 0m 0s | |
| ADateAsStringFullDateAnd... | Nominal | 0 | Least: Sunday, February 3, 19... | Most: Sunday, February 3, 19... | Values: Sunday, Febru | |
| ADateAsStringLongDateAn... | Nominal | 0 | Least: February 3, 1963 2:00:... | Most: February 3, 1963 2:00:... | Values: February 3, 19 | |
| ADateAsStringMediumDate... | Nominal | 0 | Least: Feb 3, 1963 2:00:01 PM... | Most: Feb 3, 1963 2:00:01 PM... | Values: Feb 3, 1963 2: | |
| ADateAsStringShortDateAn... | Nominal | 0 | Least: 2/3/63 2:00 PM (1) | Most: 2/3/63 2:00 PM (1) | Values: 2/3/63 2:00 PM | |
| ADateAsStringShortDateO... | Nominal | 0 | Least: 2/3/63 (1) | Most: 2/3/63 (1) | Values: 2/3/63 (1) | |
| ADateAsStringCustom | Nominal | 0 | Least: 3/2/1963 14:00:01 (1) | Most: 3/2/1963 14:00:01 (1) | Values: 3/2/1963 14:00 | |
| ADateUnixSeconds | Real | 0 | Min: -218026799 | Max: -218026799 | Average: -218026799 | Deviation: 0 |
| Now | Date time | 0 | Earliest date: Nov 16, 2013 10:16 AM | Latest date: Nov 16, 2013 10:16 AM | Duration: 0d 0h 0m 0s | |
| UnixNowSeconds | Real | 0 | Min: 1384593388 | Max: 1384593388 | Average: 1384593388 | Deviation: 0 |
| DiffFromDates | Real | 0 | Min: 1602620187 | Max: 1602620187 | Average: 1602620187 | Deviation: 0 |
| DiffFromUnixTimes | Real | 0 | Min: 1602620187 | Max: 1602620187 | Average: 1602620187 | Deviation: 0 |

Note that the strings created using `DATE_SHORT` built-in formats are ambiguous because the month and day have been swapped. This is because of locales and country codes and life is usually too short to work out how to get round this. So, the safest thing to do is to use the `date_str_custom` function which takes a format string. The strings `ADate` and `ADateAsStringCustom` are the same as a result. The other point to note is the use of UNIX timestamps to represent dates and times. The UNIX time is in fact in milliseconds so it is important to remember this when performing calculations.

# Regular expression functions

Another useful function is `finds()`, which returns true or false if a subsequence of an attribute is matched by the provided regular expression. For example, if an attribute called `sentence` contains the string `The quick brown fox jumped over the lazy dog`, and the `Generate Attributes` operator creates a new attribute using the following function expression, the result will be true:

```
finds(sentence,"\\bThe\\W+(?:\\w+\\W+){1,6}?the\\b")
```

This regular expression determines if `The` is between 1 and 6 words from `the`. Changing the `{1,6}` to `{1,4}` changes the result to false because there are five words between `The` and `the`. Note that two backslashes are required in the string to escape the backslash required for the `\w`, `\W`, and `\b` special characters.

The use of this technique allows powerful validation processes to be created to confirm that imported data and generated attributes are valid. Sometimes, unseen test data contains values outside of what was seen during the test phase, and this can be a source of subtle, hard-to-find problems. Creating a new attribute that is true or false depending on some regular expression will allow all invalid examples to be flagged. The `Filter Examples` operator allows the invalid examples to be filtered and counted.

The other important use of this technique is to filter out data that is not required. This is especially important in the real world, where data volumes can be large.

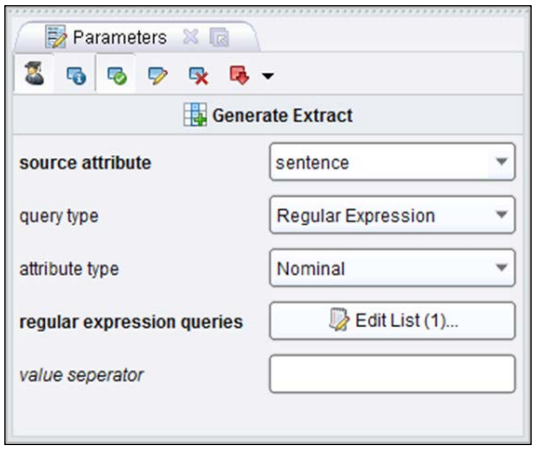
# Generating extracts

Other operators are available to extract data from within the value of attributes. For example, the `Generate Extract` operator can parse an attribute's value to extract selected data. This operator has a number of options for querying. Most typically, regular expressions are used for unstructured data, and XPath for structured data.

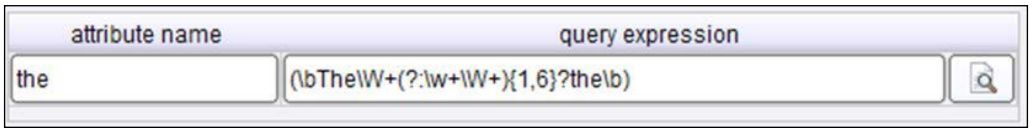
# Regular expressions

Revisiting the `finds()` function example, as part of `Generate Attribute`, an attribute called `sentence` contains the value `The quick brown fox jumped over the lazy dog` and the requirement is to extract text where the word `The` is within six words of the word `the`.

The parameters for **Generate Extract** are shown in the following screenshot:




The regular expression needed is shown in the following screenshot:




The regular expression is slightly different from that used previously. The double backslash form is not required to escape backslashes. The whole expression is enclosed in parentheses and quotes are not required around the expression.

The resulting attribute contains `"The quick brown fox jumped over the"`.

The parentheses at the beginning and end implement a regular expression capturing group, without which the operator will fail. This is because the operator uses the first capturing group as the result to populate the result attribute.

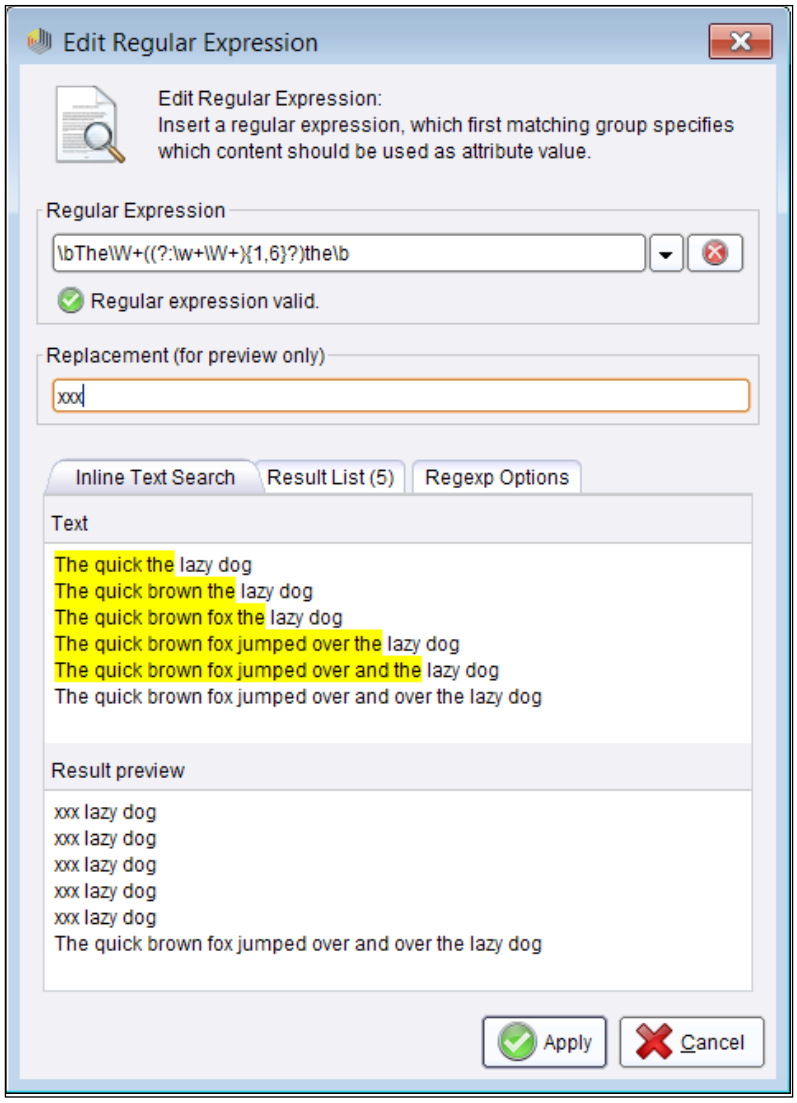
To illustrate how this can be modified, the regular expression is changed as follows:

```
\bThe\W+((?:\w+\W+){1,6}?)the\b
```

The result is `quick brown fox jumped over`.

The first capturing group now captures all the words and white space between `The` and `the`. An extremely useful feature is the regular expression editor. This can be started by clicking on the button to the right of **query expression**, as shown in the previous screenshot.

The editor lets regular expressions be tried interactively. An example is shown in the next screenshot. The yellow highlights show the successful matches, the result preview shows what a replace operation would look like, and the result list outputs the text of the matches.



Regular expressions can be daunting to learn, and this book has deliberately gone straight to a relatively complex example to show what is possible. This and other examples in this book have been inspired by various sources on the Internet, too numerous to mention (refer to *Chapter 10*, *Debugging*, for some suggestions).
To get the best out of RapidMiner Studio, it is well worth getting comfortable with regular expressions.

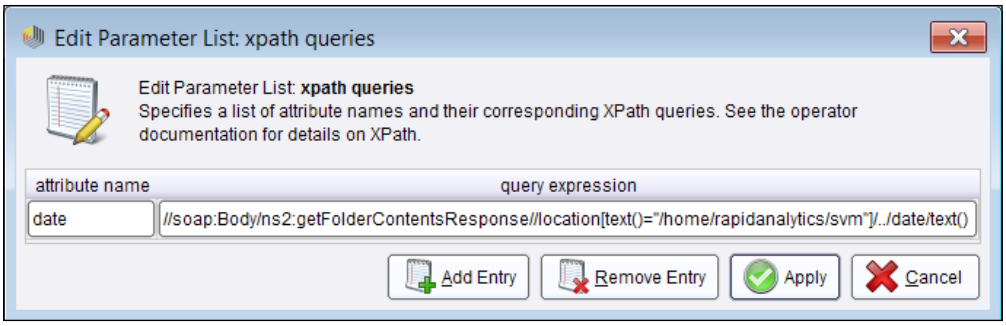
# XPath

No book on RapidMiner Studio would be complete without some discussion of XPath. This too has a learning curve but, as with regular expressions, there are plenty of examples available on the Internet. Again, this book has deliberately gone to a relatively complex first example in order to show the possibilities. Refer to *Chapter 10*, *Debugging*, for some suggestions for learning resources.

XPath works on structured XML data and an example is shown in the following code:

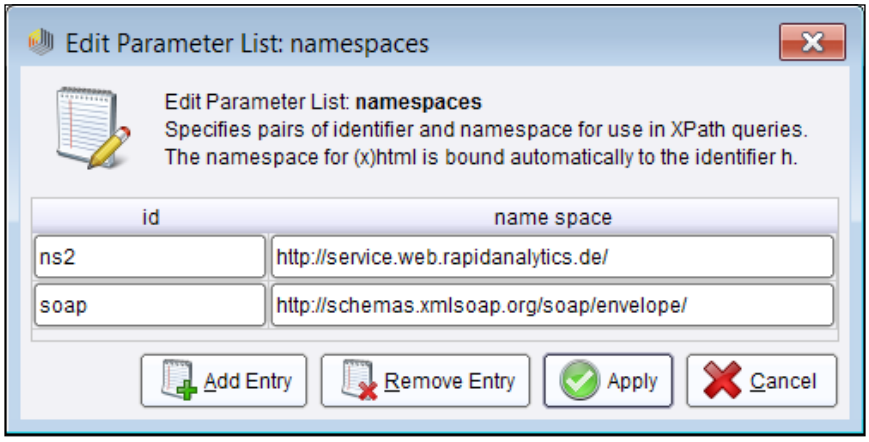
```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ns2:getFolderContentsResponse xmlns:ns2="http://service.web.
rapidanalytics.de/">
      <return>
        <status>0</status>
        <entries>
          <status>0</status>
          <date>1352240369000</date>
          <latestRevision>1</latestRevision>
          <location>/home/rapidanalytics/svm</location>
          <size>15561</size>
          <type>process</type>
          <user>rapidanalytics</user>
        </entries>
        <entries>
          <status>0</status>
          <date>1352332393000</date>
          <latestRevision>1</latestRevision>
          <location>/home/rapidanalytics/t1</location>
          <size>1639</size>
          <type>process</type>
          <user>rapidanalytics</user>
        </entries>
        <location>/home/rapidanalytics</location>
      </return>
    </ns2:getFolderContentsResponse>
  </soap:Body>
</soap:Envelope>
```

This is a **Simple Object Access Protocol** (**SOAP**) response from a RapidMiner Server server showing the contents of a folder in its repository. To extract the timestamp of the `/home/rapidanalytics/svm` process, the XPath shown in the following screenshot could be used in the **Generate Extract** operator.




This XPath searches for the named process within all location nodes. Having found it, the search moves up one level in the document and then extracts the text within the date node. This example is more complex than it needs to be and could be reduced to :
`//location[text()="/home/rapidanalytics/svm"]/../date/text()`.

However, the original illustrates the use of namespaces. These are defined in the `namespace` parameters and the next screenshot shows this. The values are taken from the original document.



It is also important to uncheck the assume HTML checkbox for this to function correctly. The end result is the value 1352240369000, which is a UNIX time in milliseconds for the specific process. Refer to the `generateExtract.xml` process in the files that accompany this book for more information.

# Renaming attributes

It is often the case that many operators generate new attributes and the names are usually self-evident to help explain what the attribute contains. Some created names, however, can contain characters such as `()=-`. These names cannot be handled by the `Generate Attributes` operator because they are interpreted as mathematical operators or operations. In this case, it is necessary to rename the attributes to more benign names and the `Rename by Generic Names` operator can be used in this case. This operator simply renames attributes to the form of att1, att2, att3, and so on. Once this is done, the `Generate Attributes` operator can be used in the normal way using the newly generated attribute names.

Renaming can also be done using the simpler `Rename` and the more powerful `Rename by Replacing` operators. These operators allow more control to be exerted over renaming and this makes it easier to rename the attributes back to the original names, which are often needed to help explain the attributes to others.

Also, of course, renaming can be used simply to make attributes' names more meaningful and the data easier to understand.

# Searching and replacing attribute values

Perhaps it is bad planning, but I often find that I have to make global searches and replacements for attribute values. For example, if a spelling mistake is discovered that is systematically present in all data that prevents it from being matched to data from other sources, it is necessary to globally correct the error. Note this is different from renaming attribute names. Search and replace is about changing values of the attributes throughout the example set.

There are a number of operators that can help including `Map`, `Replace`, and `Replace (Dictionary)`. Which operator to use depends on how complex the replacement is and how many replacements have to be made.

# Using the Map operator

The `Map` operator is the simplest and is best used to replace `whole` nominal values with alternatives. For example, if nominal attributes contain `color` and must be replaced completely with `colour`, the `Map` operator is ideal.

# Using the Replace operator

The `Replace` operator is used to change parts of an attribute's value. For example, if an attribute holds `the color green` and needs to be replaced with `the colour green`, use the following regular expression to match attribute values: `(.*)color(.*)`

Use `$1colour$2` to dictate the replacement.

In this, `$1` and `$2` are numbered capturing groups that correspond to the parts of the nominal either side of the matched item to be replaced.

More complex regular expressions are possible. There is a vast number of resources available to assist with the art of regular expressions and some are given in *Chapter 10*, *Debugging*.

# Using Replace (Dictionary)

The `Replace (Dictionary)` operator uses word value pairs in one example set (the data dictionary) to replace words in another. This operator allows regular expressions in the data dictionary, which allows a great deal of flexibility. One good feature of this operator is that by default, all occurrences of the word to replace are found and replaced. So for example, the nominal **the color green is one of the colors of the rainbow** can be changed to **the colour green is one of the colours of the rainbow**, simply by using a data dictionary mapping **color** to **colour**. All occurrences of the target are changed.

The next screenshot shows the example set before the replace operation:

ExampleSet (2 examples, 0 special attributes, 1 regular attribute)

| Row No. | nominal |
|---|---|
| 1 | the color green is one of the colors of the rainbow |
| 2 | is black a color? |

The result of the replace operation is shown in the next screenshot. Each occurrence of the word **color** is replaced with the British English spelling **colour**.

ExampleSet (2 examples, 0 special attributes, 1 regular attribute)

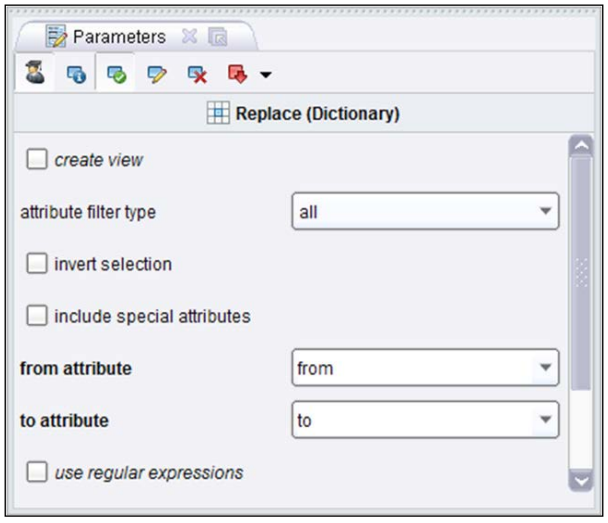
| Row No. | nominal |
|---|---|
| 1 | the colour green is one of the colours of the rainbow |
| 2 | is black a colour? |

To do this, the data dictionary example set is needed, containing attributes to dictate the **from** and **to** replacement to be applied. This is shown in the following screenshot:

ExampleSet (1 example, 0 special attributes, 2 regular attributes)

| Row No. | from | to |
|---|---|---|
| 1 | color | colour |

Finally, the `Replace (Dictionary)` operator requires parameters to be set up correctly. These are illustrated in the following screenshot:




A `mappingAndReplacing.xml` process is available with the files that accompany this book. This allows the previous examples to be recreated.

# Summary

This chapter has given an overview of how to convert attributes into different formats, extract additional information from them, as well as create new attributes. You will do this a lot.

Of particular importance is the use of regular expressions, and this chapter has given the first detail relating to their effective use. If there is one thing that is worth becoming proficient at to get the best out of RapidMiner Studio, it is regular expressions and this chapter shows that. In reality, it is not too difficult to get to an effective level to be very productive.

Having got to this point, we have done the initial import of data and have got the attributes we need. Now, we will start to look at the data in more detail; the next chapter considers outliers, which are points that do not seem to fit with the rest of the data.

# 5
# Outliers

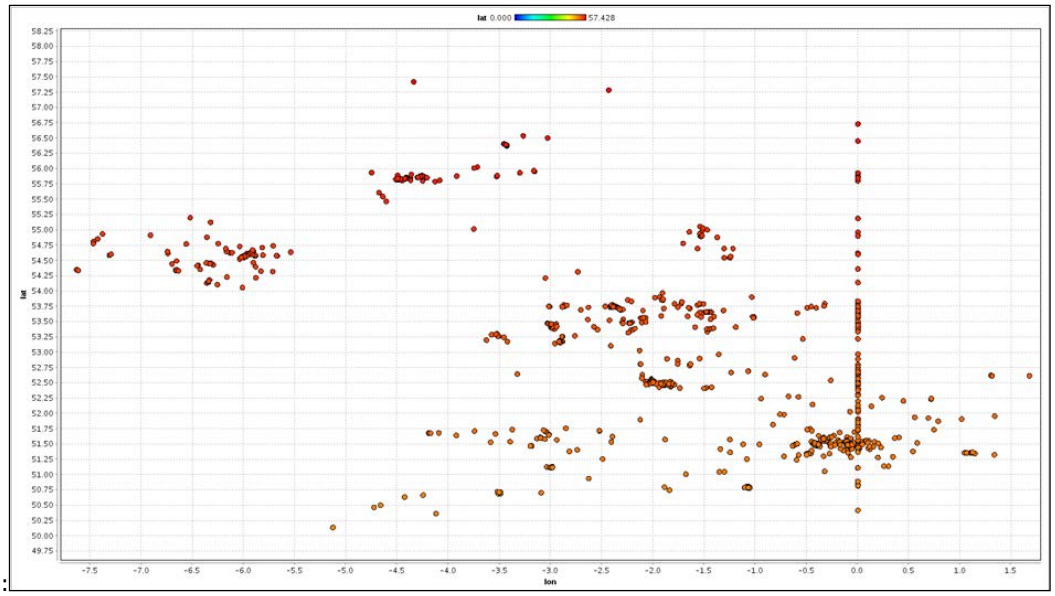
Outliers are always present in real data and this chapter gives an introduction to help detect and deal with them. An outlier is an observation that does not fit with others. Mathematically, an outlier can be considered numerically distant from other points. They can arise in different ways through measurement error, or they can be present simply because of the distribution of data. It is common that real data contains outliers and their presence can affect the results of a data mining exercise adversely. Having said that, some data exploration activities look for outliers; fraud detection is one example. It is, therefore, very important to identify them, work out why they happen, and what to do about them.

The basic, obvious approach is manual inspection. This is fine but has limits, so it is usually necessary to employ automated and systematic approaches. Having identified outliers, the question of where they arise must be answered and from there a strategy is needed to deal with them. This must include unseen data and take account of an outlier in an attribute that has not been seen before.

## Manual inspection

Manual inspection is an important method. People are generally good at seeing patterns and can detect anomalies with ease. The challenge is presenting the data in such a way so as to allow patterns to be seen. Creativity is important and some of the visualization techniques described in *Chapter 3*, *Visualizing Data*, will help in this case.

As an example, the following screenshot shows some illustrative data plotted using a simple scatter plot:

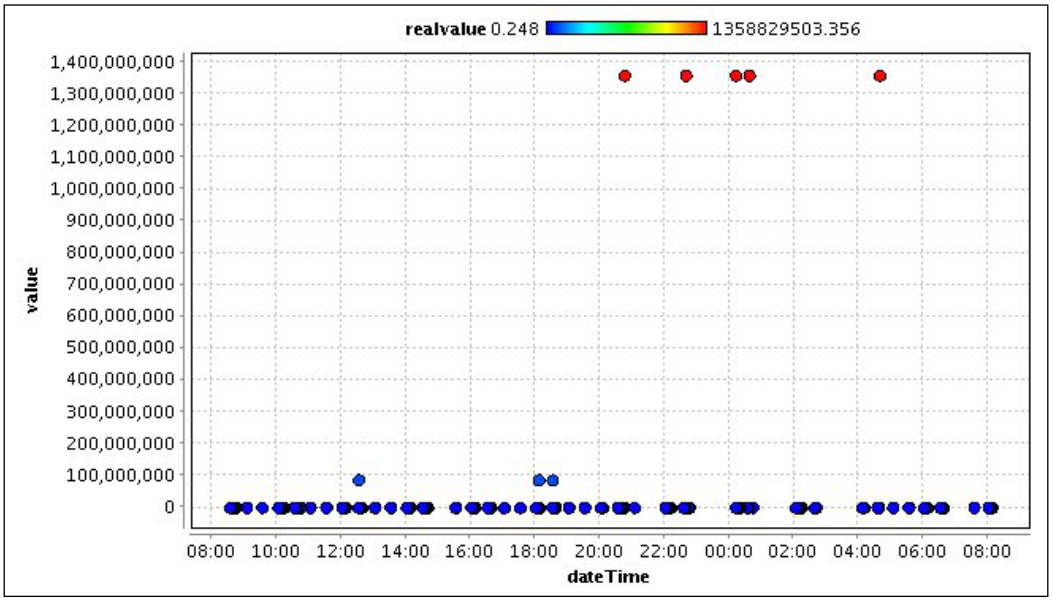



This data represents the retail sales data and comes from real locations in the UK; each point has latitude and longitude information. This means that the plot should represent a map, and the plot indeed shows an outline view of the UK. Northern Ireland is to the northwest, Scotland is to the north, and England to the south.

It is immediately obvious that there is something wrong with London, or, to be precise, there are a lot of points at the zero longitude but seemingly with valid latitudes. In this particular case, there was a bug in the import process that converted postcodes (known as zip codes in the US) to latitude and longitude values. Once this was corrected and the data was reprocessed, the problem data points disappeared.

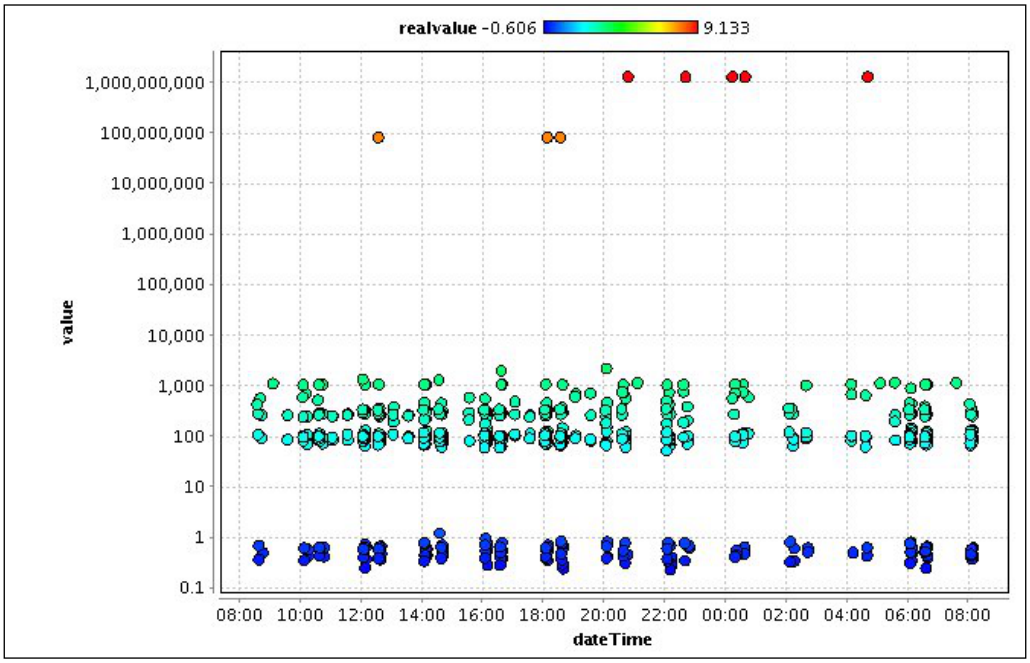
There is another more subtle error that is perhaps only obvious to someone with the knowledge of the UK coastline. There are some points to the extreme east that do not correspond to the UK mainland. A closer inspection of the data revealed that there was another bug in the data import process that incorrectly put Birmingham in the North Sea. Once this was corrected, all the data points were correctly placed in the UK mainland.

Another example is shown in the following screenshot (refer to the `simpleDisplay.xml` process and the data contained in `simpleData.csv`):




This is a simple scatter plot with time along the x axis and the value of measurement along the y axis. As seen here, there are several data points that are nine orders of magnitude away from the rest, and it is clear there is something here that warrants more investigation. There are also three questionable data points that are about eight orders of magnitude away from the rest.

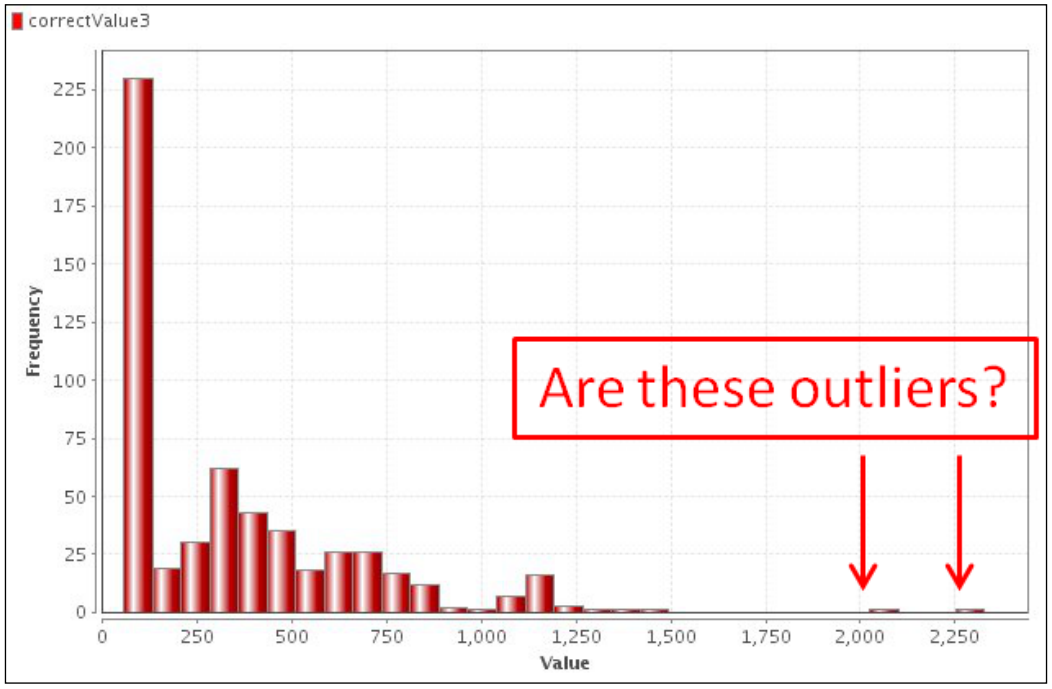
The scatter plot has a log scale checkbox, and if this is selected for the y axis, the plot is redrawn as shown in the next screenshot. Now some lower valued data points are seen below the main bulk of the data.




In this case, a closer inspection of the data reveals that the large value outliers were caused by a misunderstanding relating to how to interpret and calculate derived values in a specific edge case. The lower outliers were caused by using the wrong units while calculating values. The values should have been multiplied by one thousand to give values in the correct range.

Taking stock of this, it is clear that the challenge with visual inspection is that, by definition, it requires a person to be involved. If new data containing previously unseen outliers is processed and there is no one available, outliers would be missed. Clearly, a good practice, therefore, is to perform simple automated range checks on data to pick up any obvious outliers.

Another challenge is to know when an outlier is real because this is very subjective. To illustrate this point, we can refer to the following screenshot. Once all the outliers described in the previous paragraphs have been fixed and a histogram of the results is drawn, the output will be as seen in the screenshot. This screenshot can be reproduced using the `simpleDisplay.xml` process that accompanies this book; plot the attribute `correctValue3`.




Now the presence of outliers is not clear. Perhaps there are two of them with a value above 2,000, which have been highlighted with **Are these outliers?**. The question is now whether these are genuine outliers or if they represent a fundamental structure within the data. If the data legitimately follows this distribution, resolving outliers could adversely affect the ultimate outcome. Understanding the distribution may dictate the modeling approach that will be taken later.

Up to now, all the data has been small and easy to display. In real life, data is larger; so, this means different approaches are needed, which are described in the following sections.

# Increasing the data volume

Real data gets bigger with more examples and with more attributes.

Considering the case of more examples, the previous screenshot is essentially plotting the distribution of a single variable and an outlier is a point that is far from the mean. Clearly, when a lot of data is being considered, it can be difficult to plot it all, and in this case, the assessment of the statistical range of the data needs to be done to look for outliers. However, it is still important to understand the basic shape of the data, and random sampling using the `Sample` operator, described in *Chapter 8*, *Reducing Data Size*, is one way to achieve this. Even a 1 percent sample will still give enough information for reasonable inferences to be drawn about the overall shape of the data.

When the data contains multiple attributes, each can have outliers, but as the number increases, it becomes increasingly difficult to really see outlying points. In addition, as the number of attributes increases, the number of possible examples increases. For example, 100 attributes each with 10 possible values will theoretically give rise to $10^{100}$ (one Googol) possible examples. If the available data had 1,000 examples, it is arguable that there is insufficient data to draw any conclusions on whether a point is an outlier or not. It becomes more important to consider the relationship between attributes for a given example. This moves us to the more automated techniques described in the following sections.

# Rules for handling outliers

RapidMiner provides a number of outlier detection operators that can be used to look for outliers. It is in the nature of these operators that to a greater or lesser extent, they need to be tuned based on the characteristics of the data. In addition, once set up, automation detects general outliers but does not answer the question as to where these arise. Before going into detail about these in the next section, it is worth considering that this inevitably means outliers may tend to be discarded. This is unfortunate since the data is precious and discarding it cannot be taken lightly. It is also the case that the presence of an outlier may be predictive of the class, so the deletion of the outlier may make the predictive power of any model worse. One of the outlier operators discussed in the following section uses the class label to determine outliers.

Class and attribute correlation is discussed in more detail in *Chapter 6*, *Missing Values*, in the context of missing values where some techniques are shown to allow missing values to be correlated with other attributes and labels for the data to help inform a decision on how to handle them.

# Automated detection of example outliers

Four different outlier detection techniques are described in the following sections. They are as follows:

- `Detect Outlier (Distances)`
- `Detect Outlier (Densities)`
- `Detect Outlier (LOF)`
- `Detect Outlier (COF)`

None of these algorithms will automatically find the correct outliers for the data being explored. Given their parameters, they will flag up candidate outlier points to allow a person to get involved and make the final determination. This is an important point that needs to be built into any data exploration process.

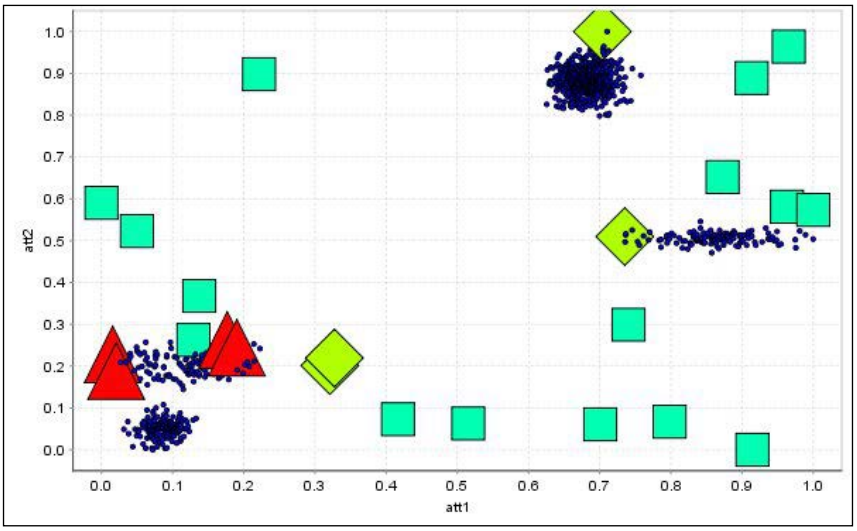
# Detect Outlier (Distances)

The simplest operator is `Detect Outlier (Distances)`. Each example is considered in turn and the distance to the *k*th nearest neighbor is determined (k is provided as a parameter). Note that distance in this context means the Euclidean distance or one of a number of possibilities. The top *n* (*n* is provided as a parameter) examples that are at the largest distance from their *k*th nearest neighbor are marked as outliers.

The algorithm is conceptually simple and requires no underlying knowledge of the distribution of the data. It requires a value for *k* to be chosen and will always choose an outlier even if the data does not actually have any.

An empirical approach must be adopted for choosing *k*. Choosing 1 as a value for *k* will use the nearest point and not take into account the presence of any other points nearby that could indicate membership of a cluster. This tends to mark points at the edge of less dense clusters as outliers. Choosing a value of *k* that is too large will start to include legitimate points as outliers within denser clusters and miss the genuine outliers. This is illustrated in the following three screenshots where the images show some artificial data arranged in clusters with a random noise overlaid. This artificial data has 1,000 examples with two attributes, and is generated with the standard `Generate Data` operator using the target function Gaussian mixture clusters. In addition, 20 random points were added with values in the 0 to 1 range. The graphics also show the result of outlier detection with the shape of the point indicating if the outlier correctly matched the known outlier. Refer to `outlierDistancePlotter.xml` for the process that can recreate these.

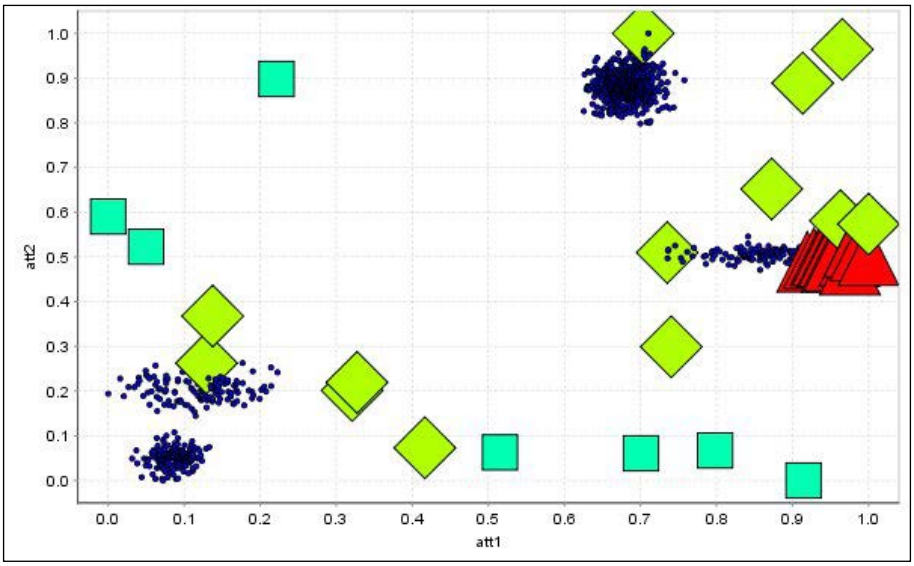
The first of these shown in the next screenshot highlights 20 outliers detected with *k* set to 1.




In the previous screenshot, triangles represent valid points being set as outliers, diamonds represent outliers being classed as valid points, and squares represent correctly identified outliers. The small points are the valid points correctly identified as valid.

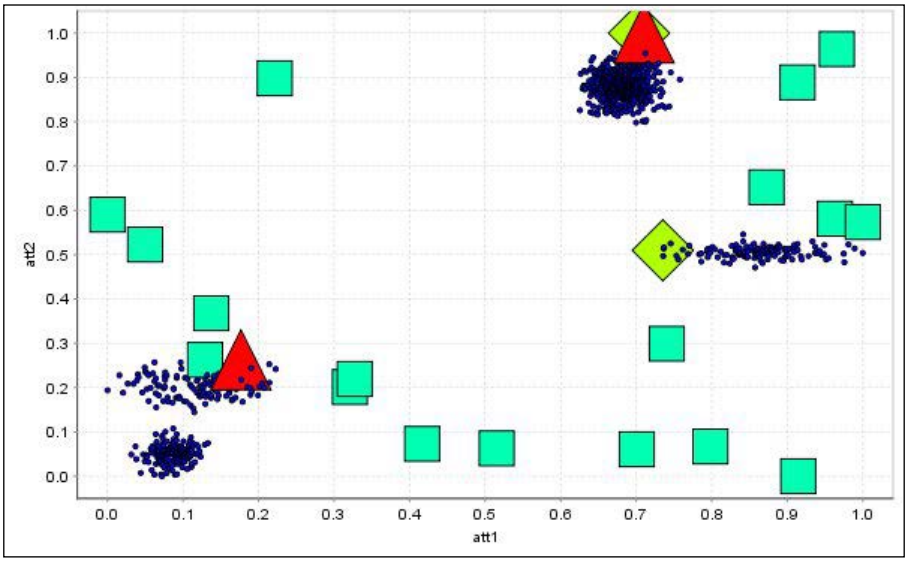
The screenshot shows four points (the triangles) incorrectly set as outliers when in fact they are legitimately a part of a cluster. In addition, four points (the diamonds) are counted as valid points.

Changing *k* to a large value changes how points are classified. For example, the following screenshot shows what happens when *k* is set to 137.




As explained before, the shapes of the points indicate what the outlier detection has produced. This time a large number of points in a cluster are being marked as outliers when they should not be. In addition, more genuine outliers are being missed.

Intuitively, there is a point somewhere between these two extremes that has the ideal value of *k*, and only by empirical observation in the context of your data will it be found. The following screenshot shows the case with *k* set to `2`.




Now the number of incorrectly identified outliers and misclassified real points has reduced to two each. In real life, of course, you do not have the luxury of knowing which points are in fact outliers. Unfortunately, it is a case of trial and error to understand why some points are being shown as outliers.

In passing, these graphics were produced using the advanced plotting capabilities of RapidMiner Studio. The process uses the `Generate Attributes` operator on the result of the outlier detection to create new attributes based on whether the outlier was accurate or not. These new attributes were used to set the shape and colors of individual points. The comments in the process file give instructions on how to recreate these diagrams.

A final point is that given an outlier is always found, it may be advisable to use outlier detection during initial exploration. However, once in production, a different approach should be used, and it is certainly not advised to use automated detection and implied deletion.

# Detect Outlier (Densities)

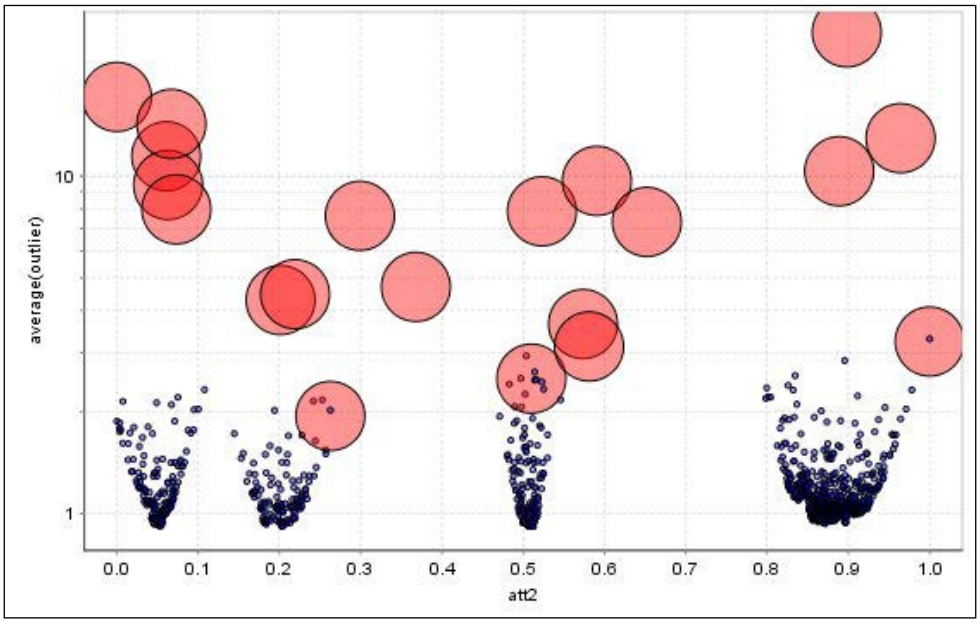
The `Detect Outlier (Densities)` operator is also relatively simple to use. The operator requires two parameters, a distance and a proportion, which makes it slightly harder to set up. Unlike the distances operator, it does not necessarily find outliers and so may be more suited for use in a production environment. For a pair of `distance` and `proportion` parameters, the operator marks a point as an outlier if there are more than the `proportion` points further than the `distance` away from it. Looked at another way, a multidimensional sphere is drawn around each point corresponding to the `distance` parameter, and the number of other points that are within the sphere is compared to the total outside points. If there is more than the `proportion` outside the sphere, the point is an outlier. This corresponds to a density measurement; points in less dense neighborhoods are outliers.

Generally the `proportion` parameter should be set to a number approaching 1 (note that the parameter is in the 0 to 1 range, representing 0 percent to 100 percent); the closer the parameter is to 1, the fewer the number of outliers. The `distance` parameter must be set somewhere between the maximum and minimum distance in the data; the larger this value, the fewer the number of outliers. It always requires an empirical investigation to determine the optimum values, and the determination will usually include an iterative investigation to determine why there are outliers and what rule is needed to deal with them.

The main weakness of the density approach is that the algorithm cannot cope with regions of different densities. This leads us to the **Local Outliers Factor** (**LOF**) operator.

# Detect Outlier (LOF)

The `Detect Outlier (LOF)` operator proposed by *Breunig et al* is able to cater for different densities and therefore is able to find outliers from regions of different densities within a dataset. The algorithm takes two simple parameters corresponding to a minimum and maximum number of *k* nearest neighbors. A local density is estimated from these neighbors and this allows regions of lower density to be determined. Points in low density regions are outliers. The obvious question is what values should you choose? As usual it depends, and empirical investigation is the only way. Having said that, the values of *k* do not seem to have too much impact on whether an example is indicated as an outlier. However, there is a drawback; the operator does not give a Boolean flag to indicate whether an example is an outlier and instead a numerical value is returned. The higher this is, the more likely it is that the example could be an outlier. This means that a relative determination must be made to decide whether an example is really an outlier. For example, by using the data from the previous section that discussed the `Detect Outliers (Distances)` operator and performing an `LOF` outlier determination for multiple different values of the upper and lower bound for *k* and then aggregating the results, the following graph is produced. The process to do this, called `bestLOFFinal.xml`, is available with the files that accompany this book. The instructions for configuring the advanced plotter to display this graph are included with this process.



The graph in the previous screenshot differs from the previous screenshots because **att 1** is not plotted. Instead, the graph shows the **average(outlier)** factor produced by the outlier detection (**average(outlier)** on the y axis) as the upper and lower *k* values are changed. Recall that the original data contains 1,020 data points and what is shown in the previous screenshot is the average outlier factor, for multiple combinations of the lower and upper bounds. The upper and lower parameters are not shown but the result on the **average(outlier)** factor can be seen. Using **att2** gives the opportunity to relate the graph to the real data as shown previously. As can be seen, outliers, shown as larger points, are consistently above the main clusters of points. It is also interesting to observe the minimum values where the clusters are seem to form concave shapes. Local minima presumably correspond to the centers of the clusters. By empirical observation, it is possible to select a threshold above which a point is an outlier. In this case, a point with an outlier factor above 4 is a candidate to be considered as an outlier, although that needs to be understood in the context of the data.

The complexity of the operator can adversely affect performance for large datasets and this is an additional point to bear in mind.

# Detect Outliers (COF)

If the data has class labels, it makes intuitive sense that an outlier for one class would have a different characteristic when compared with another class. The `Detect Outliers (COF)` operator can be used in this case. This operator is complex in its operation and the interested reader can research further by referring to *A Comparative Study of Outlier Mining and Class Outlier Mining* by *Motaz K.Saad* and *Nabil M.Hewahi*.

The algorithm has straightforward parameters. The two important ones are the number of neighbors and the number of class outliers. The number of neighbors parameter must be chosen empirically, although experiments indicate that it is not that critical. A very low value will presumably lead to oversensitivity to single points and a very large value may include points that are in different classes.

The number of outliers simply dictates how many outliers will be found. Note that this is independent of the number of classes. For example, if this is set to 10 and there are five classes, the process may determine two outliers from each class or another combination. This means the operator always finds outliers, but, unlike the distance operator, an outlier factor is also returned for all examples that are marked as outliers. This allows a threshold to be used as an additional check for outliers. All non-outliers have this outlier attribute set to infinity. Generally, the lower this value, the more likely the example is an outlier. By performing an empirical investigation similar to the local outlier factor method, it is possible to determine a range for thresholds.

The algorithm is relatively expensive in operation and large datasets may require more processing time, so this must be kept in mind when using it. In addition, if there is no label in the input data, clearly the operator cannot be used. This means that unlabeled test data cannot be checked for outliers and one of the alternatives must be used.

# Summary

Outliers must be considered while exploring real data, and this chapter has given some techniques for spotting them as a part of a recommended systematic process that allows the root cause behind the creation of the outlier to be determined. In addition, automated handling could be implemented while bearing in mind that it may be dangerous to give complete autonomy to a system because it may delete perfectly good data. It is better perhaps to implement automated checking to highlight outliers in unseen data so as to allow a human to get involved.

Bear in mind that real data never behaves as well as fake data. What matters is being able to quickly determine what data could be an outlier, then work out whether it is or not. This chapter has given some tools to help you with this.

Another big issue with real data is missing values. As we shall see in the next chapter, it is important to determine some rules to handle these.

# 6
# Missing Values

Very often, the values of attributes within examples do not have a value. This is missing data. It normally arises in many ways and is very important to deal with since some algorithms suffer profoundly even with a small percentage of missing data. There are different types of missing data, and these can affect the approach used to deal with it.

Deleting the examples with missing data is not a good strategy. Not only is all the data potentially valuable, but it is also entirely possible that the missing data is correlated to the predictions, which might be the whole point of the data mining process. It is also a bad idea to manually fill in the missing values. Not only is this not scalable, but this also risks introducing a bias that can ruin subsequent modeling activities. Instead, a systematic approach based on an understanding of how the missing data arises is better.

RapidMiner allows investigations to be performed quickly, and this chapter gives some very detailed explanations of the exploratory processes available using various looping operators.

## Missing or empty?

Before starting, it is important to be clear on the distinction between missing and empty data. They are very different. Missing data may have a value, whereas empty data may not. It is unfortunate that they both look the same, and sometimes, only a domain expert can tell them apart. A good analogy is to compare the journey of a commuter train and an express train, where attributes are the times when they stop at stations along the route. The commuter train stops at many intermediate stations, whereas the express train stops at far fewer stations for the same length of track. The absence of attribute values for stopping times at the intermediate stations for the express train is not missing data. Domain knowledge of trains is needed to know this, but it is quite clear to all train users that inventing a value for stopping times at the intermediate stations for the express train is wrong.

Having got this distinction clear, the next step is to understand what the different types of missing data are. This is important because it dictates how the missing values should be handled.

# Types of missing data

*Little* and *Rubin*, the authors of the book, *Statistical Analysis with Missing Data* (Second edition, 1987), categorized missing data in three ways. They represent three different mechanisms, which cause missing data to arise and are described in more detail in the following sections. Understanding the type can help us take an informed decision about how to deal with missing values.

# Missing completely at random

This is the situation where the missing data neither depends on the value of available data nor on missing data itself. Another way to think of this is to imagine how **missing completely at random** (**MCAR**) data could be synthesized. Imagine a dataset consisting of 100 attributes and 10,000 examples starting with no missing values. Randomly select an attribute from the 100 available and then randomly select an example from the 10,000 available examples. Set the value to missing and repeat this process to obtain the desired amount of missing data. The missing values in this case are MCAR.

# Missing at random

**Missing at random** (**MAR**) is the situation where the missing values are dependent in some way on the values of the other attributes (including potentially the label if it is present) but not on the values of the missing data itself. To synthesize data like this, return to the consideration of the 100 attributes and 10,000 examples dataset mentioned in the previous section. Choose one of the attributes to potentially be missing, then for each example use the values of the other attributes to decide if it really should be missing. For example, if we consider attribute1 to be missing, a simple rule could be to look at the value of attribute2, and if this is greater than a threshold, set attribute1 as missing.

An example of this might be a situation where some test equipment fails to record a value. A closer examination of the data shows that the failure coincides with the equipment being powered off for routine maintenance and this is indicated by some other attribute in the data. The measurement of values, which are available to be gathered, is not affected at all by whether the equipment is available or not.

# Not missing at random

This is the situation where the missing values depend on the value of the missing data itself. An example of this might be a computer that measures, once a minute, the average CPU load that it experiences. When the computer is busy and the CPU is very loaded, values are more likely to be missing. This happens because the computer cannot keep up and record a measurement, as it is too busy. It is observed that the data will contain fewer values for the CPU load at the high end. This illustrates the importance of understanding the mechanism that leads to missing values. Ignoring the missing values in this case will bias any investigation, so that it will appear that the computer is not loaded, and this may lead to a failure to spot a major problem.

To synthesize **not missing at random** (**NMAR**) data using the previous 100 attributes and 10,000 examples dataset, it is necessary to choose an attribute to be missing, observe its value and then apply a rule to decide if it should be marked as missing.

Note that we are using the value itself to decide if it should be missing and this, of course, raises an interesting issue. In real life, the data is missing and you do not have the value before it was missing. You can speculate about the mechanism that leads to the missing values but fundamentally there is no way to be sure. Discussing this further is out of the scope of this book, so we can settle on the mechanism for creating the data and from there, see if there are ways to detect it.

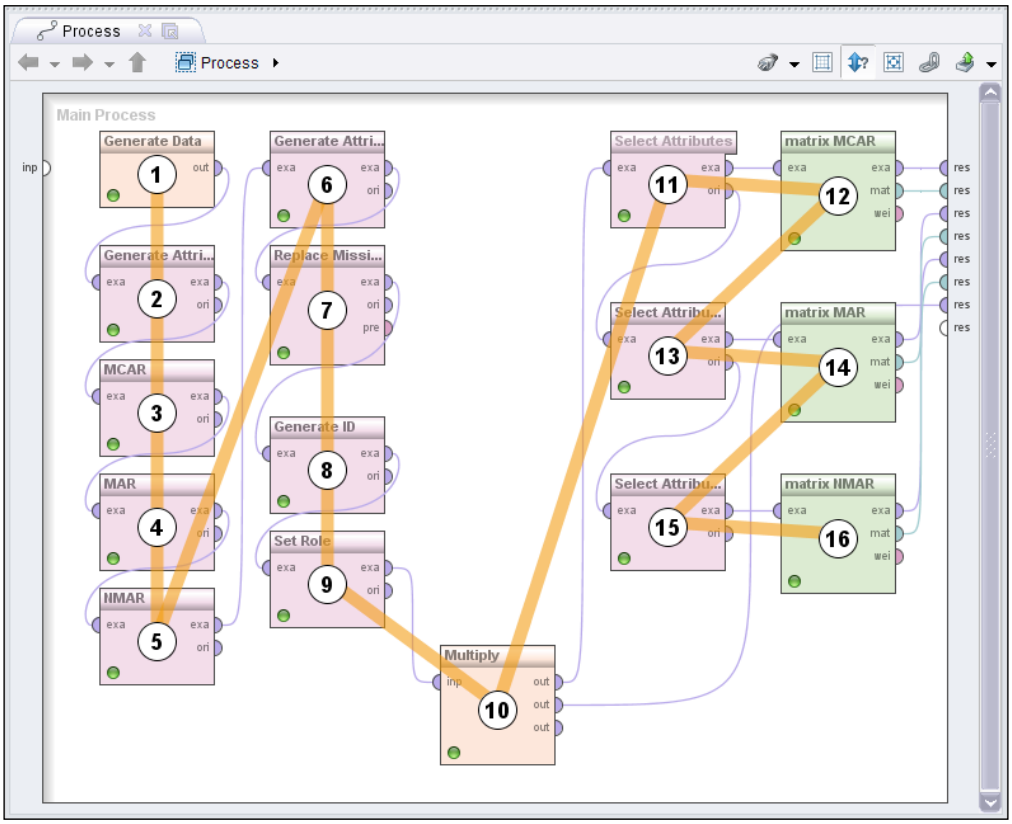
# Categorizing missing data

Having settled on the types of missing data, the question that arises is what are the approaches for categorizing it?

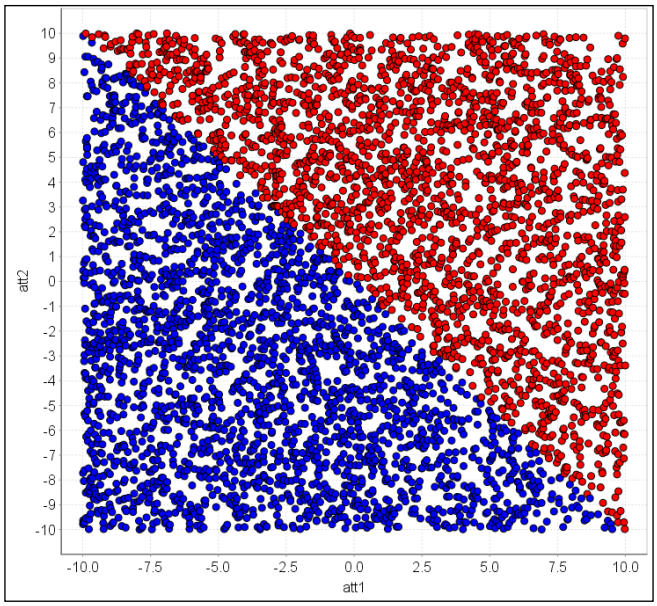
This section gives a detailed set of worked examples using synthetic data and a RapidMiner Studio process that is available with the files that accompany this book. These are intended to be followed with the text. The process is called `MCARDetection.xml`.

The first step is to make some synthetic data containing missing data of each type. In order to illustrate the key points, it is necessary to reduce the size of the synthetic data, so it can be easily displayed and understood. Of course, real data will not be like this, but the techniques are usable with high-dimension data.

The RapidMiner Studio process to be used is shown in the following screenshot:




This generates a simple example set with 10,000 examples and two attributes named `att1` and `att2` (the operators labeled **1** and **2** in the screenshot). The label is generated from the values of `att1` and `att2` using the sign of the result to dictate the label. The following screenshot shows a plot using RapidMiner Studio, where the two attributes are placed on the x and y axes and the color is dictated by the label, which corresponds to the sign of the sum of the attributes.




The RapidMiner process then generates a new attribute for `att1` using MCAR, MAR, and NMAR rules. In other words, some of the values of `att1` are set to missing based on these rules, but rather than change `att1`, a new attribute is created to hold the value. In addition, another attribute is generated that contains a simple true or false flag to indicate whether the new `att1` attribute is missing or not in each of the three cases (operators **3** to **10** included in the process screenshot).

Simple rules are given here for the three cases:

- Firstly, for MCAR, the attribute **att1MCAR** is missing based on the following expression in the `RapidMiner Generate Attributes` operator:

  ```
  if (rand() > 0.1,false,true)
  ```

- For the MAR case, the attribute **att1MAR** is missing as seen from the following expression:

  ```
  if (att2 > 2 && att2 < 6,true,false)
  ```

- Finally, for the NMAR case, the following expression is used to generate **att1NMAR**:

  ```
  if (att1 >  3 && att1 < 7,true,false)
  ```

A small fragment of the data in tabular form is shown in the following screenshot:

ExampleSet (10000 examples, 2 special attributes, 8 regular attributes) / 10,000 examples): all

| id | label | att1 | att2 | att1MCARStatus | att1MCAR | att1MARStatus | att1MAR | att1NMARStatus | att1NMAR |
|---|---|---|---|---|---|---|---|---|---|
| 1 | positive | 2.468 | 7.267 | false | 2.468 | false | 2.468 | false | 2.468 |
| 2 | negative | 1.292 | -8.625 | false | 1.292 | false | 1.292 | false | 1.292 |
| 3 | negative | -5.590 | -0.307 | false | -5.590 | false | -5.590 | false | -5.590 |
| 4 | positive | -6.351 | 9.370 | false | -6.351 | false | -6.351 | false | -6.351 |
| 5 | positive | 1.241 | 8.733 | false | 1.241 | false | 1.241 | false | 1.241 |
| 6 | negative | -8.768 | 3.852 | true | ? | true | ? | false | -8.768 |
| 7 | negative | -5.190 | -1.978 | true | ? | false | -5.190 | false | -5.190 |
| 8 | negative | -6.225 | -9.413 | true | ? | false | -6.225 | false | -6.225 |
| 9 | negative | -7.154 | -8.453 | false | -7.154 | false | -7.154 | false | -7.154 |
| 10 | negative | -4.696 | -4.395 | true | ? | false | -4.696 | false | -4.696 |
| 11 | positive | 5.304 | -2.017 | false | 5.304 | false | 5.304 | true | ? |
| 12 | positive | 0.911 | 0.057 | false | 0.911 | false | 0.911 | false | 0.911 |
| 13 | positive | 5.377 | 8.017 | false | 5.377 | false | 5.377 | true | ? |
| 14 | positive | 0.328 | 6.166 | false | 0.328 | false | 0.328 | false | 0.328 |
| 15 | negative | -2.608 | -1.474 | false | -2.608 | false | -2.608 | false | -2.608 |
| 16 | negative | -2.585 | -9.906 | true | ? | false | -2.585 | false | -2.585 |
| 17 | positive | 5.895 | 0.971 | false | 5.895 | false | 5.895 | true | ? |
| 18 | negative | -4.110 | 2.683 | false | -4.110 | true | ? | false | -4.110 |
| 19 | positive | 5.533 | 3.061 | false | 5.533 | true | ? | true | ? |
| 20 | negative | -4.080 | -7.754 | false | -4.080 | false | -4.080 | false | -4.080 |

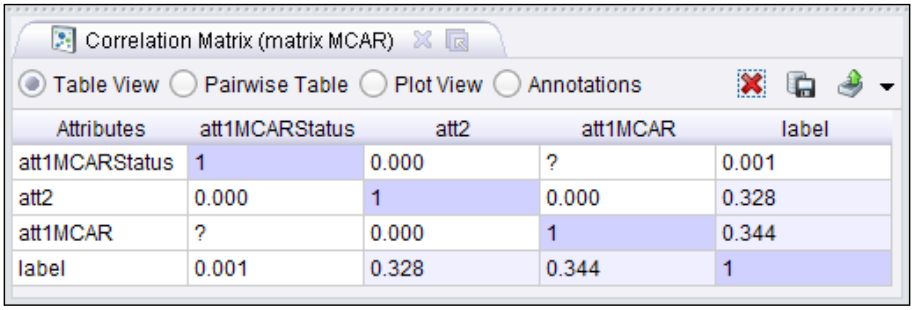
In real life, all the examples in the **att1** column will not be available. All that is available is only one of either the **att1MCAR**, **att1MAR**, or **att1NMAR** columns. Furthermore, the underlying mechanism that generates the missing data is also not available. The point of this exercise is to see if it is possible to determine the mechanism which leads to the missing data using various techniques that, in turn, will drive the best method to handle them.

In the table view shown earlier, the status columns have been generated from the missing values and `true` means the value is missing. The status columns are useful when plotting the data.

Given this data, the next step is to see whether the mechanism that generated the missing data can be determined. Two approaches will be taken. First, the use of correlation to determine if the attributes depend on one another, and second, manual inspection. Operators **11** to **16** perform the correlation calculations for the three missing attribute generation regimes. Operators **12**, **14,** and **16** are the `Correlation Matrix` operators, which use squared correlation.

# Finding MCAR data

The MCAR correlation matrix, which is the output of operator **12**, within the process screenshot is shown in the following table:

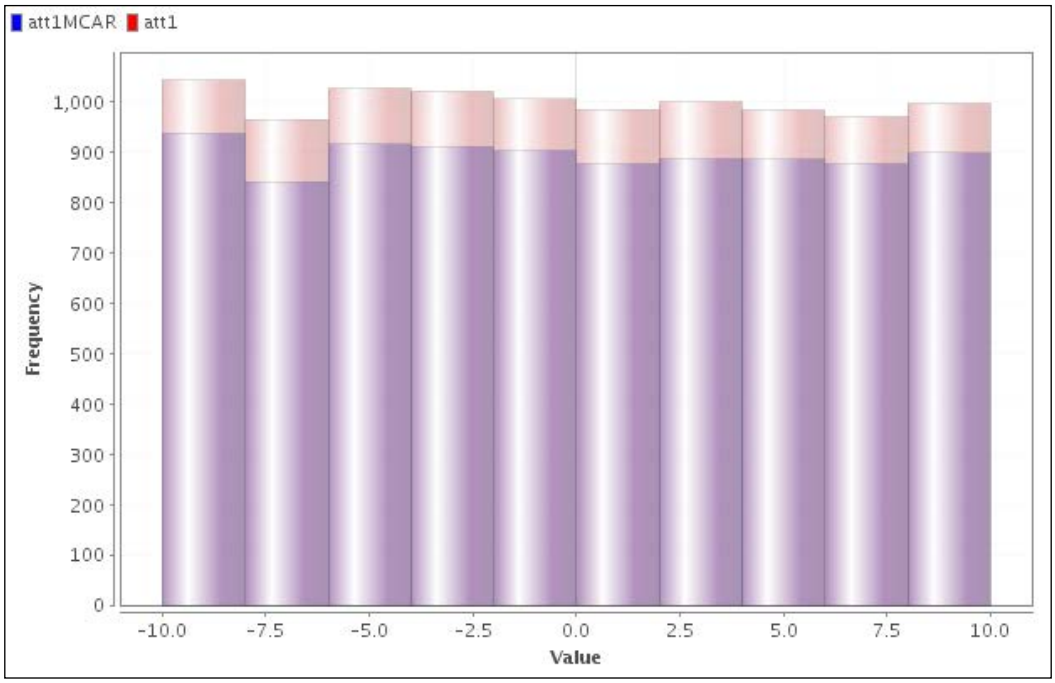



| Attributes | att1MCARStatus | att2 | att1MCAR | label |
|---|---|---|---|---|
| att1MCARStatus | 1 | 0.000 | ? | 0.001 |
| att2 | 0.000 | 1 | 0.000 | 0.328 |
| att1MCAR | ? | 0.000 | 1 | 0.344 |
| label | 0.001 | 0.328 | 0.344 | 1 |

Bear in mind that the attributes **att2**, **att1MCAR**, and **label** are all that will be available, and the **att1MCARStatus** attribute is derived from **att1MCAR**. The correlation operator determines how two attributes depend on one another. A squared correlation is used in this case, and a value of 0 indicates no correlation, while a value of 1 indicates correlation or anticorrelation.

As shown in the table, **att2** is partially correlated with the label (the value is **0.328**), and for the non-missing values, **att1MCAR** is also partially correlated with the label (the value is **0.344**). The missing state of **att1MCAR** is shown by the **attMCARStatus** attribute, and this shows no correlation between both **att2** and the **label** attribute (both the values are very close to 0). This is evidence that the missing values of **att1MCAR** are MCAR but as we shall see, the attribute could still be NMAR.

Manual inspection of the data will give the opportunity to spot patterns. A histogram generated using the plot capabilities within RapidMiner Studio on the example set output from the `Correlation Matrix` operator, which displays the distribution of **attMCAR** and **att1** values, is shown in the following screenshot:




There are 10,000 examples and the values for the attributes range between -10 and +10. The histogram has been set to have 10 bins and shows the count of the number of values in that bin, and as can be seen, there are approximately an equal number of values across all the bins for both attributes. If, as domain experts, we know that the distribution of **att1** follows a certain distribution, such as the one shown, and we see that the distribution of **att1MCAR** is the same, then this is evidence that the missing values for **att1** have been generated completely at random. This gives us evidence that the `att1MCAR` attribute has not been generated with an NMAR mechanism.
In real life, `att1` has missing values, so we will not get the **att1** histogram seen previously. In this situation, we have to rely on domain knowledge.

# Finding MAR data

To find the MAR data, repeat the MCAR investigation. But in the case of MAR, it results in a correlation matrix as shown in the following screenshot:

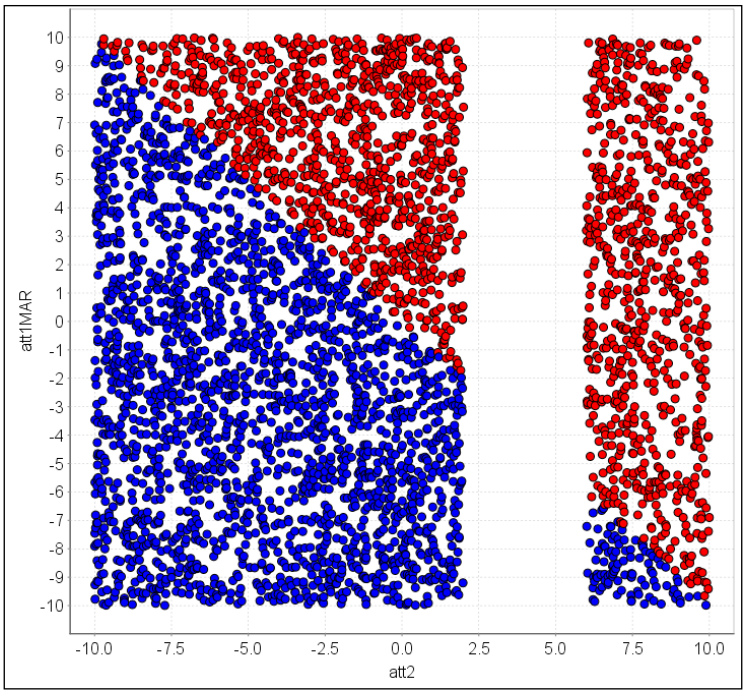
Correlation Matrix (matrix MAR)

Table View | Pairwise Table | Plot View | Annotations

| Attributes | att1MARStatus | att2 | att1MAR | label |
|---|---|---|---|---|
| att1MARStatus | 1 | 0.119 | ? | 0.040 |
| att2 | 0.119 | 1 | 0.000 | 0.328 |
| att1MAR | ? | 0.000 | 1 | 0.317 |
| label | 0.040 | 0.328 | 0.317 | 1 |

This differs from the MCAR case because there is now a correlation between **att2** and the missing or present status of `att1`, as indicated by **att1MARStatus**. Note that there is no correlation between **att1MAR** and **att2**.

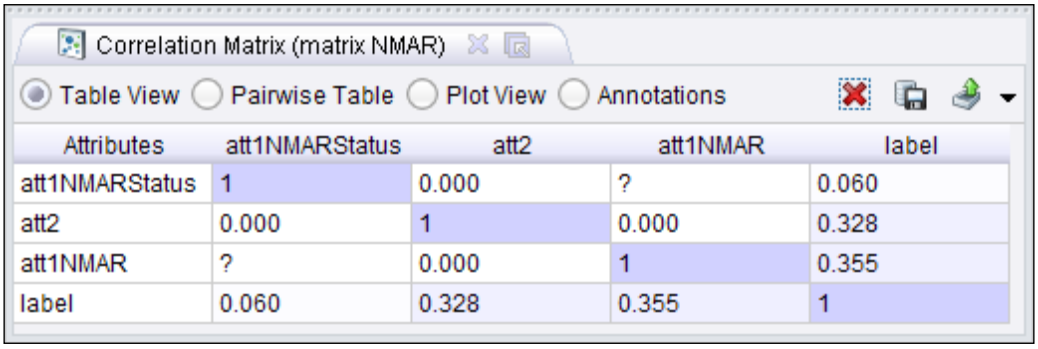
There is some relationship between **att2** and **att1MAR**, and this can clearly be seen by plotting them together on a scatter plot as shown in the following screenshot:



The graph clearly shows that **att1MAR** is missing if **att2** is between 2 and 6 (in accordance with the formula in the `Generate Attributes` operator). This means that the missing values of **att1MAR** follow the MAR mechanism.

# Finding NMAR data

For the NMAR case, the correlation between the available attributes is shown in the following figure:

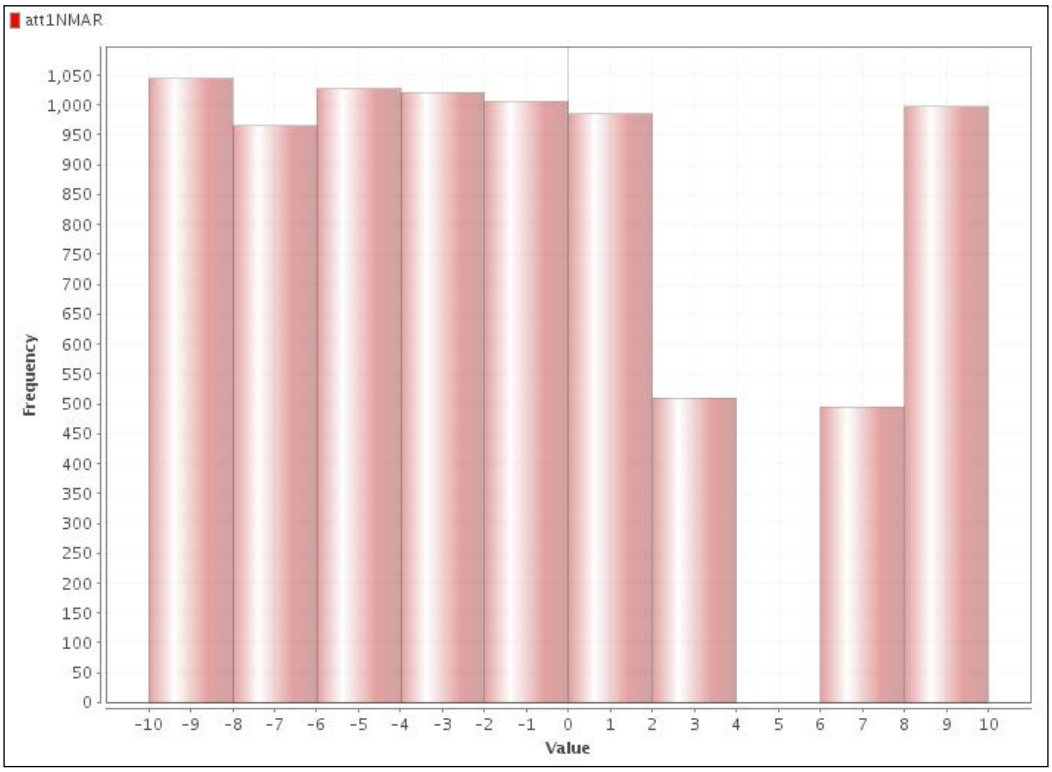



| Attributes | att1NMARStatus | att2 | att1NMAR | label |
|---|---|---|---|---|
| att1NMARStatus | 1 | 0.000 | ? | 0.060 |
| att2 | 0.000 | 1 | 0.000 | 0.328 |
| att1NMAR | ? | 0.000 | 1 | 0.355 |
| label | 0.060 | 0.328 | 0.355 | 1 |

Recall that NMAR means the missing status of **att1NMAR** depends on the value of the attribute before it was missing. By definition, therefore, there is no value to see because it is missing, so there is no way to prove that the underlying mechanism is NMAR. The correlation matrix is also very similar to the MCAR case, and this can lead to a mistaken conclusion that the data is MCAR, leading to an incorrect method for handling the missing data.

There is one subtle difference, however, and it is the slight correlation between the **att1NMARStatus** and **label** attributes. If we believe that `att1` itself has predictive powers for the `label` attribute and we can see that the corresponding missing status attribute, which is either true or false, also has predictive power, we may hypothesize that the missing status depends somehow on the value before it was missing.

If a histogram of `att1NMAR` is plotted, the result is shown as follows:




Compare this to the histogram for the MCAR case. If we expected an even distribution for `att1NMAR`, we can see that it's missing if its value is between **3** and **7**. Of course, if we don't know, as domain experts, what the distribution of `att1NMAR` should be, it is completely possible that the distribution represents the MCAR one.

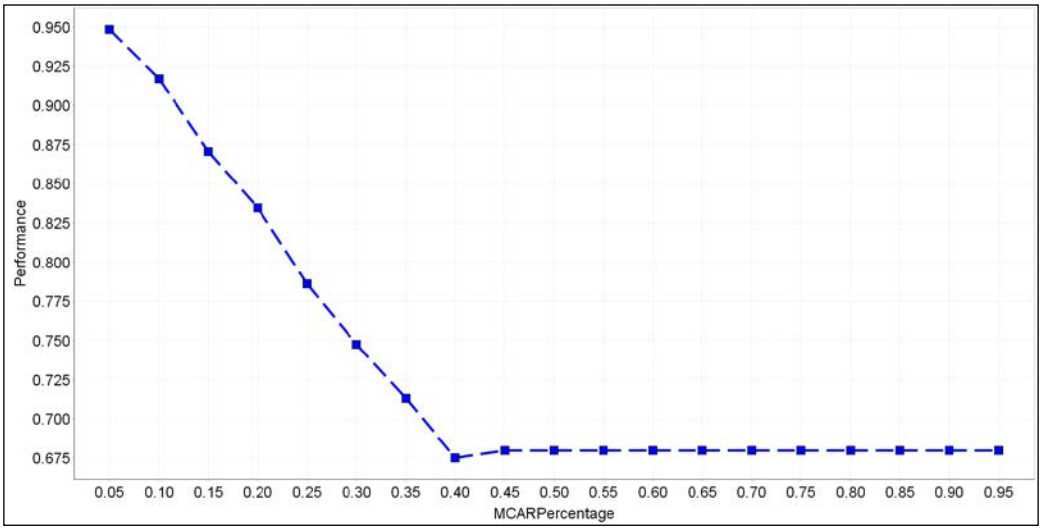
# A cautionary note

Real data will never be as easy to interpret, and you are very likely to find that the missing attributes exhibit MAR, NMAR, and MCAR behavior at once. As it very often happens, each has to be dealt with on a case-by-case basis.

# Effect of missing data

Missing data can reduce the effectiveness of classification models in terms of accuracy and bias.

To illustrate this point, the following graph shows the performance of a simple decision tree classifier based on the data from the previous sections for different amounts of MCAR data for `att1`:




As seen in the previous graph, the performance drops steadily as more missing data is present. In this case, when there is no `att1` value, the decision tree classifier uses `att2` values alone to make a prediction, which leads to the performance shown. Different classifiers have different sensitivities to missing data, and furthermore, the data itself and the influence of the missing values on the result will also affect the performance. However, the conclusion remains that while exploring data, identifying the missing data and finding ways to deal with it is important for a successful outcome. This leads to the next section where various options are given to handle the missing values.

# Options for handling missing data

The exploration of data identifies missing data, and the overall process outside the scope of the exploration needs to consider the options for handling it and how these are affected by the type of missing data. Some guidelines are given in the following sections to help you make your decisions.

# Returning to the root cause

It is obvious that missing data is a bad thing. So if it happens, it's always worth stepping back and determining why it's missing in the first place. The time spent on fixing the root cause of missing data will save time later and improve the quality of the data exploration and mining process in general.

# Ignore it

Some learning algorithms cope with missing values but some do not. An example of one that is extremely sensitive to missing data is the support vector machine, which will produce very poor results even with one missing attribute. For example, using the `LibSVM` operator with the 10,000 examples from an earlier section in this chapter, it is possible to achieve a 99.6 percent classification performance. Adding a few missing values reduces this immediately to 50 percent and there is no warning message given. It could easily happen that when processing unseen data, model accuracy will be completely compromised if a few rogue missing values creep in.

Ignoring missing data in the sense of allowing it to happen without understanding that it is there and additionally understanding what effect it could have is unwise. The key point is to ensure that if missing data is to be allowed, an active decision must be taken to do so.

# Manual editing

Manual editing has some drawbacks. Not only is this not scalable as the amount of missing data increases, but also it is error prone, leading to bias and furthermore, it does not address the deployment problem when unseen test data is presented to a model. In this case, the person doing the manual editing has to be available, has to remember the rules used to edit the data, and may have to cope with missing values that do not fit the manual rule.

Generally, it is not wise to perform manual editing. If you feel you are doing it, ensure that whatever is done is turned into an automated rule that can be applied later.

# Deletion of examples

This is a common approach, also known as case deletion or list-wise deletion. Most people do this, and it is acceptable if the number of examples to be deleted is small, and more importantly, if the missing data has been determined to be MCAR.

If the missing data is NMAR, deleting examples may risk bias being introduced. Deletion of examples also leads to a loss of all the other attributes in the example, which are not missing. This loss of data is generally to be avoided since the data is precious.

# Deletion of attributes

This too is a common approach and is acceptable if the number of missing values represents a large proportion of the whole. It is also acceptable if the attribute does not have much influence on the final result. Clearly, such an attribute would be a candidate for removal anyway. If it additionally turns out to have missing values with an MCAR profile, this would be enough reason to remove it.

If the missing values are MAR or NMAR, deleting the attribute is likely to affect model accuracy and more careful consideration would need to be given to deletion.

# Imputation with single values

A simple approach to replacing missing values is to replace them with a value, for example the value 0, or the mean of the non-missing attributes. If the missing data is MCAR, this is acceptable and usually the mean is the best choice since it will have the smallest impact on the characteristics of the data as a whole.

If the missing data is MAR, it depends on other attributes, and it is better in this case to use a modeling technique to work out a value for the missing attributes. This is discussed in the next sections.

If the data is NMAR, there is no easy way to choose a single value for a replacement and case-by-case consideration is required. As an example, the NMAR data from an earlier section in this chapter looked as though `att1` was missing if its own value was between 3 and 7. In this case, the average for the non-missing values of `att1` is approximately -1.3. Using this to replace the missing values of `att1` may have an adverse effect since we know that the value should be between 3 and 7. In this case, it would be more sensible to use the value 5 for a single replacement. Of course, in this case, we have the luxury of knowing how the data was generated. This will not normally be the case.

# Modeling

If the data is MAR, it is sometimes possible to use modeling techniques to determine a value for the missing data based on other attributes. In effect, the missing values become labels to be predicted based on the values of the other attributes.

Using the 10,000 example data used earlier in this chapter, the missing values of `att1` can be predicted from the `att2` and `label` attributes. Clearly, `att2` by itself can tell you nothing about `att1`, only the chance that it is missing but the addition of the label allows some prediction to be made about what `att1` would have been. Using the label to predict a missing attribute value and then using this later will lead to a problem, and so should be avoided. In practice, real data usually contains many attributes and the values of the missing attributes will depend in some way on them. This means that it should always be possible to construct a model to predict missing attributes from others without needing the label.

# Summary

This chapter introduced us to missing values and underlined the importance of identifying and handling them in order to improve model accuracy.

A number of different mechanisms that underlie the creation of missing attributes were discussed as well as some ways to detect them and then deal with them. The underlying mechanism drives decisions about how to handle the missing values. As with most data exploration techniques, all the situations must be handled on a case-by-case basis, but the importance of missing data means that it is worth having a basic framework to handle it.

Having reached this point, we imported data, cleaned it, and removed outliers and missing values. The next step is to restructure it by transforming it into a different format or by summarizing in new ways to suit the problem to be solved or gain new insights into an overall understanding. This is the subject of the next chapter.

# 7
# Transforming Data

Transforming data can often make it easier to mine. This chapter discusses some common data transforms whose objective is to present data in the form of example sets containing examples that have their own attributes. Typically one of these attributes is predicted based on the values of the other attributes.

Once you get the idea of the example and example set clear, you will find that data mining becomes a lot easier. When looking at new data for the first time, you will find yourself mentally transforming it into an example set-like format. It is also good discipline to produce all new data in an example set-like format so that future data exploration and mining activities are made easier.

However, some learning and practice is required to get confident with the data transformation operators. This chapter will cover some detailed examples that will show you how to make example sets with some typical real data.

The first step is to revisit attribute creation to show how to create new attributes based on other attributes in the same example as well as other values from the example set as a whole.

Next, we will look at aggregation. This is a way of summarizing data that is also useful for combining fragments of a transaction into a single example.

Pivoting and de-pivoting are then considered. Pivoting is useful when data is available as rows of name-value pairs that should be considered together in single examples. De-pivoting is useful when multiple attributes contain an implied additional dimension that is better represented as a separate dimension.

Finally, windowing is covered. This is useful when consecutive examples representing a time series need to be converted into single examples representing a time period within the series.

# Creating new attributes

We have already covered attribute generation in a previous chapter, but there are some additional techniques to allow new attributes to be generated from attributes in the same example as well as from values in other examples.

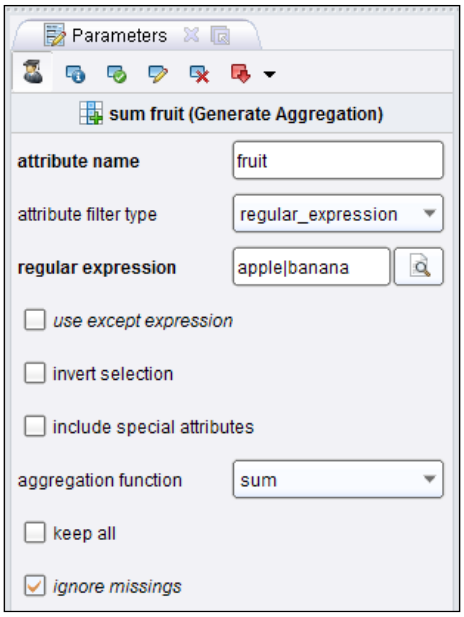
Consider the simple example set shown in the following screenshot (see the process `readFruitAndVeg.xml` in the files that accompany this book to recreate this):

ExampleSet (11 examples, 1 special attribute, 4 regular attributes)

| Row No. | id | apple | banana | carrot | daikon |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 1 | 2 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 | 0 |
| 4 | 1 | 0 | 1 | 0 | 1 |
| 5 | 1 | 0 | 0 | 0 | 1 |
| 6 | 2 | 1 | 3 | 0 | 1 |
| 7 | 2 | 1 | 0 | 2 | 1 |
| 8 | 2 | 0 | 1 | 0 | 0 |
| 9 | 3 | 1 | 0 | 0 | 1 |
| 10 | 3 | 0 | 0 | 2 | 0 |
| 11 | 4 | 0 | 3 | 0 | 1 |

The previous screenshot describes the count of each item for a number of transactions. If we want to calculate the total number of fruits and the total number of vegetables, we can use the `Generate Aggregation` operator (**apple** and **banana** are fruits and **carrot** and **daikon** are vegetables).

The following screenshot shows typical parameters that could be used to generate such a set of totals (refer to the process `manipulateFruitAndVeg.xml` to see this):

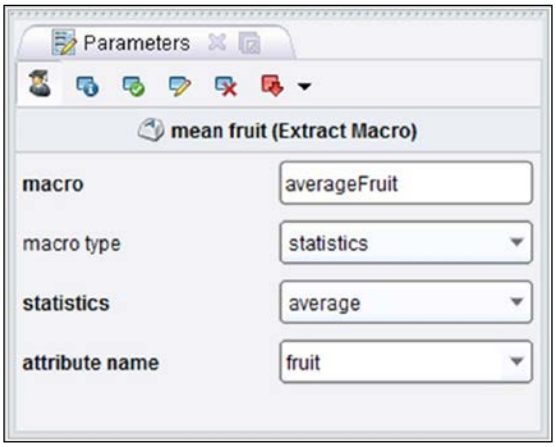



The **attribute name** field is set to the name of the field to be created. The **attribute filter type** parameter is set to **regular_expression**, and in this case it is set to `apple|banana` to select the two items of fruit in the example set. The **aggregation function** parameter is set to **sum** in order to add the attribute values for **apple** and **banana** together and the **keep all** check box is cleared, which has the effect of deleting the **apple** and **banana** attributes from the example set. A second operator would be needed to do the equivalent for **vegetables**, and in this case the regular expression would be `carrot|diakon`. After applying these operators, the example set appears as shown in the following screenshot:

ExampleSet (11 examples, 0 special attributes, 3 regul

| Row No. | id | fruit | vegetable |
|---|---|---|---|
| 1 | 1 | 1 | 0 |
| 2 | 1 | 3 | 1 |
| 3 | 1 | 0 | 1 |
| 4 | 1 | 1 | 1 |
| 5 | 1 | 0 | 1 |
| 6 | 2 | 4 | 1 |
| 7 | 2 | 1 | 3 |
| 8 | 2 | 1 | 0 |
| 9 | 3 | 1 | 1 |
| 10 | 3 | 0 | 2 |
| 11 | 4 | 3 | 1 |

To obtain the mean and standard deviations for the **fruit** and **vegetable** attributes, the `Extract Macro` operator can be used. This operator allows various summary statistics about the example set to be placed into a macro. It should be noted that the sample standard deviation is calculated and not the population. For example, the following screenshot shows the parameters needed for the `Extract Macro` operator to determine the average for the **fruit** attribute and place the value into a macro called **averageFruit**.




The macro name (**macro** shown in the screenshot we just saw) is set to `averageFruit` and **macro type** is set to **statistics** from the drop-down list (a number of other options are available and the interested reader is encouraged to experiment with them), the **statistics** option is set to **average** and the **fruit** attribute must be chosen so that the average for this is calculated.

To calculate the standard deviation of the **fruit** attribute, a second `Extract Macro` operator is needed with **macro** set to `standardDeviationFruit` and the **statistics** parameter set to `deviation`. Two more operators are needed for the same calculations on the **vegetable** attribute (refer to the `manipulateFruitAndVeg.xml` process to see these).

Once this is done, the `Generate Attributes` operator can be used to calculate the z-score—the number of standard deviations an attribute is away from the mean. This is shown in the following screenshot:

| attribute name | function expressions |
|---|---|
| zFruit | (fruit-%{averageFruit})/%{standardDeviationFruit} |
| zVegetable | (vegetable-%{averageVegetable})/%{standardDeviationVegetable} |

Various macros are used with the attribute's name to calculate the value for the new attribute for each example.

The final example set is shown in the following screenshot:

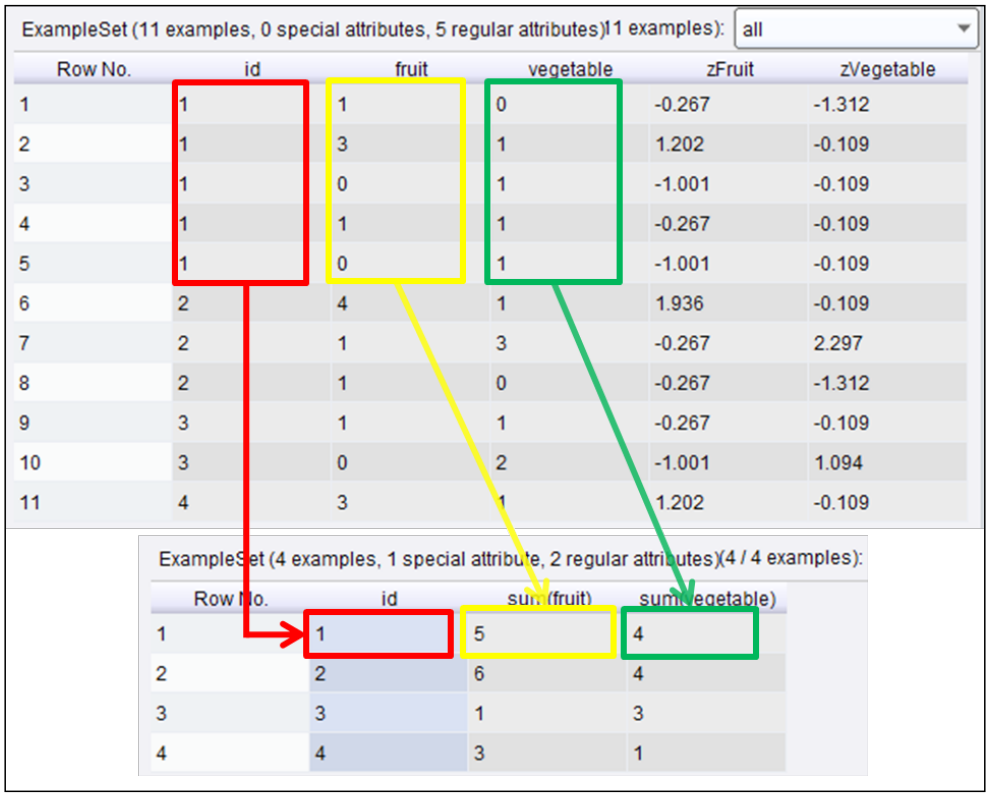
ExampleSet (11 examples, 0 special attributes, 5 regular attributes)

| Row No. | id | fruit | vegetable | zFruit | zVegetable |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | -0.267 | -1.312 |
| 2 | 1 | 3 | 1 | 1.202 | -0.109 |
| 3 | 1 | 0 | 1 | -1.001 | -0.109 |
| 4 | 1 | 1 | 1 | -0.267 | -0.109 |
| 5 | 1 | 0 | 1 | -1.001 | -0.109 |
| 6 | 2 | 4 | 1 | 1.936 | -0.109 |
| 7 | 2 | 1 | 3 | -0.267 | 2.297 |
| 8 | 2 | 1 | 0 | -0.267 | -1.312 |
| 9 | 3 | 1 | 1 | -0.267 | -0.109 |
| 10 | 3 | 0 | 2 | -1.001 | 1.094 |
| 11 | 4 | 3 | 1 | 1.202 | -0.109 |

By checking this manually, we find that the average for the **fruit** column is 1.364 and for the sample standard deviation is 1.362. For a value of 1, the z-score would be -0.267, which agrees with the numbers in the screenshot.

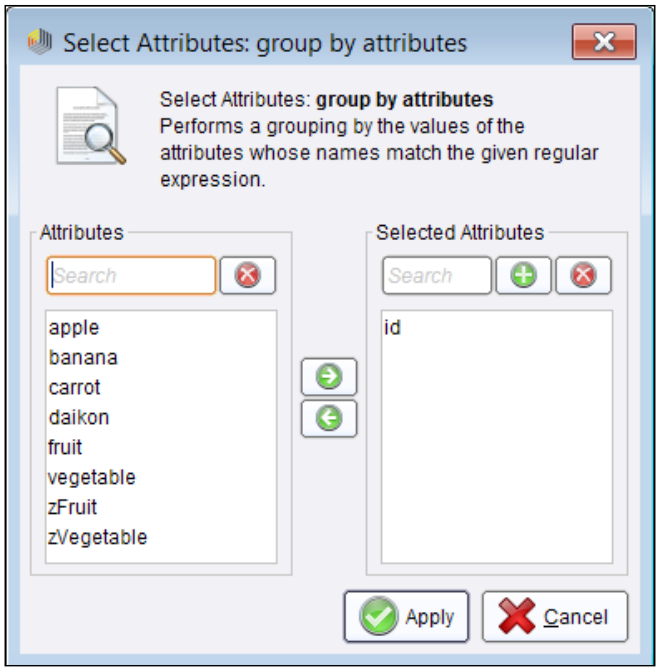
# Aggregation

Aggregation combines examples with the intention of reducing their number and uses aggregation rules to combine these attributes and make single new attributes. For example, the data shown in the previous screenshot has multiple entries for the same ID. If these were the transactions in a day, aggregation would be used to combine them to get a daily view. The following screenshot represents a picture of what is required:

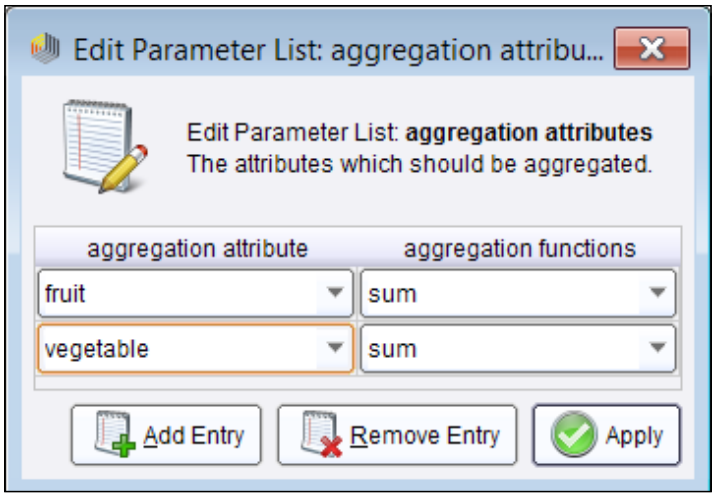



The **id** attribute is used as the column to group examples by (similar to the idea of grouping in SQL). For each example with the same ID, the sum of the **fruit** and **vegetable** attributes is calculated to create a new attribute in one example of the final example set.

The operator that can carry out this aggregation is `Aggregate`. The parameters for this operator to implement the required aggregation are shown in the following two screenshots. The first is shown in the following screenshot of the **group by** dialog box and handles the grouping of examples. Here, the selection of the **id** attribute will group all examples with the same ID together.




The aggregation within a group is then controlled by the **aggregation attributes** dialog box for this operator. This is shown in the following screenshot:

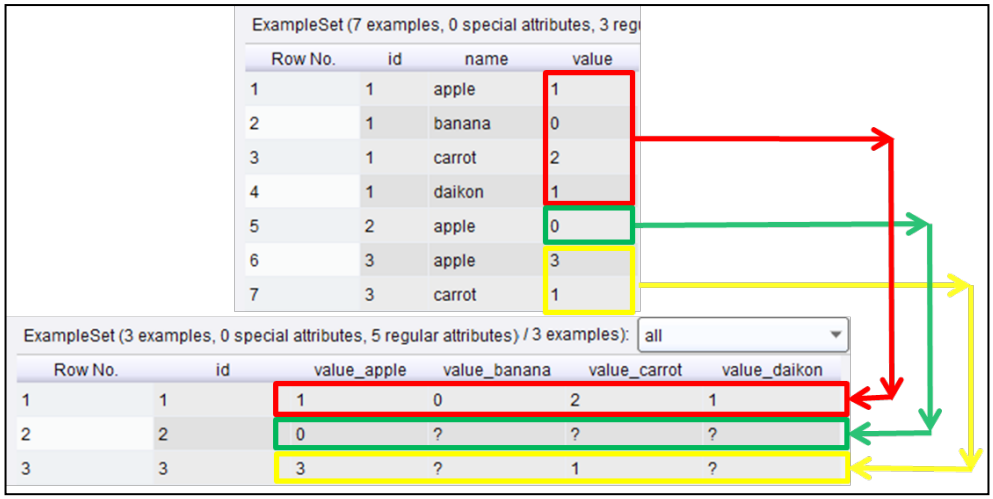


The values for **aggregation attribute** and **aggregation functions** are chosen. Because of this, all examples in the group have the function applied to the attribute; the final result will be stored in a new attribute. The name of the new attribute is derived from the aggregation attribute and the aggregation function. In this case, **sum** is used so the attribute will be called **sum(fruit)** and **sum(vegetable)**.

An important general point about aggregation is that missing values are often present in the data to be aggregated. The parameters of the operator allow these missing values to be handled.

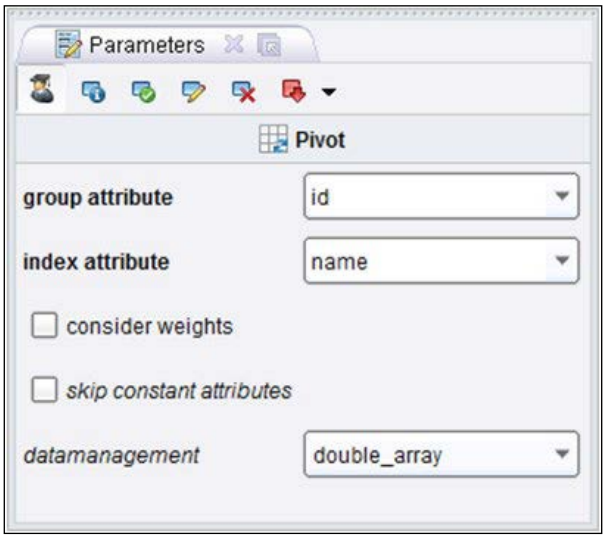
# Using pivoting

The best way to understand how pivoting works is by looking at an example. Refer to the process `pivoting.xml` provided with this book. This is used to produce the result shown in the following screenshot. It displays an example set containing name-value pairs for a specific `id`, which in this case could be regarded as an identifier:



The pivot result shows how new attributes have been created for each ID based on the content of the **name** attribute in the **example set** input. The number of examples in the result is equal to the number of unique identifiers in the **id** column (in this case, three), and the number of new attributes created is equal to the number of unique values in the name column (in this case, it is four). The names of the attributes in the result are derived from joining the name of the **value** column and the values in the **name** column. Clearly, there can be empty values shown by question marks and these must be handled. In the case earlier, they would probably be set to 0 as a step following the pivot operation.

The parameters required for the `Pivot` operator to create this result are shown in the following screenshot:




As can be seen in the screenshot, these parameters are very simple. The **id** attribute is used as the value for **group attribute** to group examples together, and the index attribute is used to create new attributes whose names are based on the value of the attribute within the group. For example, in the pivoted data, the attribute **value_carrot** for **id 3** has the value 1. This means that in the original data, there is a row where **id** is **3**, **name** is **carrot**, and **value** is **1**.

If other attributes are present in the example set before pivoting, they are also included. This can make the result complex, so it is worth considering the elimination of extra attributes by using `Select Attributes` or by setting their roles to be special using the `Set Role` operator.

If there are duplicate entries in the input example set, the pivot operation will take the last one it encounters and ignore the rest up to that point. To handle this, some aggregation must be done first to create a unique ID for the `Pivot` operator to handle.

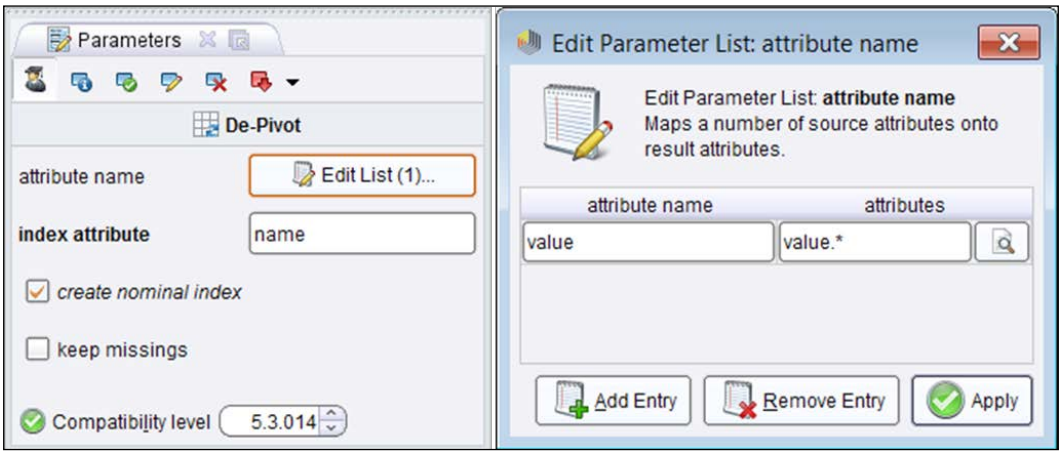
The names produced by the `Pivot` operator can get quite complicated and you may find yourself having to rename them using the various `Rename` operators.

# Using de-pivoting

As with pivoting, de-pivoting is best understood by looking at a picture. The previous screenshot showed how pivoting works, and de-pivoting, as its name implies, can be seen as a reverse operation. Refer to the `pivoting.xml` process provided with this book.

Refer to the screenshot in the *Using pivoting* section and reverse the operation. The result contains data that has multiple values in a single example set, perhaps recorded over a period of time. What de-pivoting will do is to turn the example set back into name-value pairs with one example for each in the original example set. So if the original example set contains four attributes in addition to an id that ties them together, the de-pivoted result will contain four examples for each id. Each of these examples will have a common id, an attribute with a value based on the name of the original attribute, and a value derived from the intersection of the original attribute and the common id.

The parameters needed to make the `De-Pivot` operator work are shown in the following screenshot:



The **index attribute** is the name of an attribute that will be created as a result of the de-pivot operation. The **attribute name** parameter is set to **value**, and this will be the name of an attribute that will be created in the de-pivot result. A regular expression is used to select the attributes to include in the result; in this case the expression is `value.*` (in simple English, this means *match an attribute that begins with value and has zero or more arbitrary characters after it*). This selects the four attributes beginning with value.

This results in the example set are as shown in the following screenshot:

ExampleSet (7 examples, 0 special attributes, 3 regular at

| Row No. | id | name | value |
|---|---|---|---|
| 1 | 1 | value_apple | 1 |
| 2 | 1 | value_banana | 0 |
| 3 | 1 | value_carrot | 2 |
| 4 | 1 | value_daikon | 1 |
| 5 | 2 | value_apple | 0 |
| 6 | 3 | value_apple | 3 |
| 7 | 3 | value_carrot | 1 |

The only difference between this and the original example set are the values for the **name** attributes. These are all preceded with **value**_ because this is a faithful reproduction of the source attributes. This can be resolved by using a `Replace` operator, which can be seen in the `pivoting.xml` process.

The following points should be noted for the parameters to the `De-Pivot` operator:

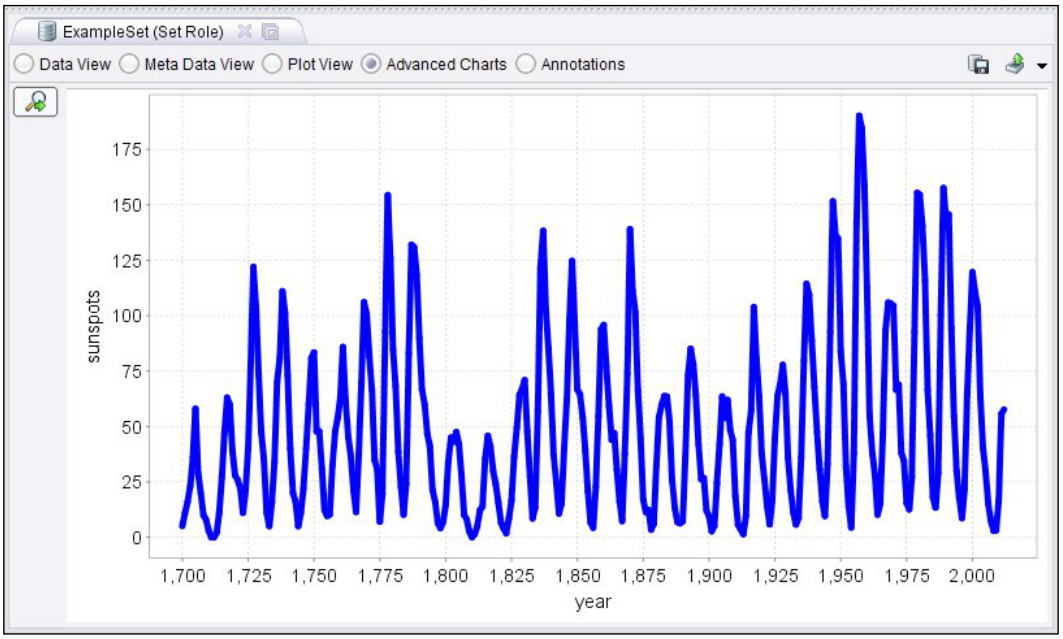
- The **create nominal index** checkbox must be checked if the values of the index attribute are going to be nominal (which they are in this case).
- The **index attribute** field itself (in this case, **name**) has as many unique values as there are attributes that match the expression contained within the list for the attribute name. For the example given, there are four attributes that match the expression, hence there are four unique values for the attribute.
- Each matched attribute from the attribute name list creates a name in the index column, and the corresponding value is created from the intersection of the matched attribute and the ID.

Practice makes perfect and the best way to get to grips with this operator is to try it.

# Windowing data

Windowing is typically used to turn time series data into example sets containing examples with multiple attributes corresponding to sequential points. These example sets can then be used for model building, classification, or predictive analysis. Windows can also be used to visualize data.

The best way to explain this operator is to use real data to illustrate its main features. Some real sunspot data is shown in the following screenshot (the process to recreate this is available in `readSunspot.xml`, which accompanies this book):

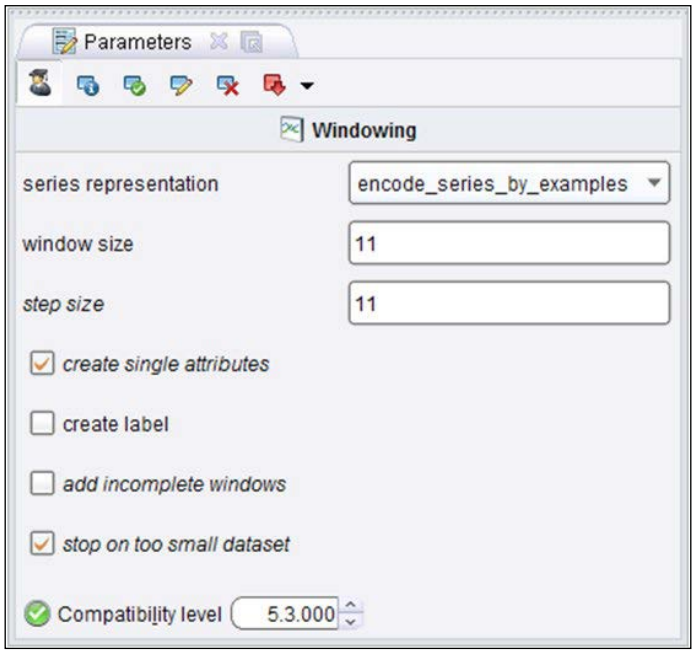



This data is a yearly count of the number of dark spots visible on the sun and follows a number of cycles of which the 11-year one is the most well known. Applying the `Windowing` operator with a window size of 11 and a step size of 11 to an example set containing a single attribute called **sunspots** yields the data shown in the following screenshot (the **year** attribute has been made into an ID using the `Generate Id` operator):

ExampleSet (28 examples, 1 special attribute, 11 regular attributes) Filter (28 / 28 examples): all

| ... | year | sunspots-10 | sunspots-9 | sunspots-8 | sunspots-7 | sunspots-6 | sunspots-5 | sunspots-4 | sunspots-3 | sunspots-2 | sunspots-1 | sunspots-0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1710 | 5 | 11 | 16 | 23 | 36 | 58 | 29 | 20 | 10 | 8 | 3 |
| 2 | 1721 | 0 | 0 | 2 | 11 | 27 | 47 | 63 | 60 | 39 | 28 | 26 |
| 3 | 1732 | 22 | 11 | 21 | 40 | 78 | 122 | 103 | 73 | 47 | 35 | 11 |
| 4 | 1743 | 5 | 16 | 34 | 70 | 81 | 111 | 101 | 73 | 40 | 20 | 16 |
| 5 | 1754 | 5 | 11 | 22 | 40 | 60 | 80.900 | 83.400 | 47.700 | 47.800 | 30.700 | 12.200 |
| 6 | 1765 | 9.600 | 10.200 | 32.400 | 47.600 | 54 | 62.900 | 85.900 | 61.200 | 45.100 | 36.400 | 20.900 |
| 7 | 1776 | 11.400 | 37.800 | 69.800 | 106.100 | 100.800 | 81.600 | 66.500 | 34.800 | 30.600 | 7 | 19.800 |
| 8 | 1787 | 92.500 | 154.400 | 125.900 | 84.800 | 68.100 | 38.500 | 22.800 | 10.200 | 24.100 | 82.900 | 132 |
| 9 | 1798 | 130.900 | 118.100 | 89.900 | 66.600 | 60 | 46.900 | 41 | 21.300 | 16 | 6.400 | 4.100 |
| 1 | 1809 | 6.800 | 14.500 | 34 | 45 | 43.100 | 47.500 | 42.200 | 28.100 | 10.100 | 8.100 | 2.500 |
| 1 | 1820 | 0 | 1.400 | 5 | 12.200 | 13.900 | 35.400 | 45.800 | 41 | 30.100 | 23.900 | 15.600 |
| 1 | 1831 | 6.600 | 4 | 1.800 | 8.500 | 16.600 | 36.300 | 49.600 | 64.200 | 67 | 70.900 | 47.800 |
| 1 | 1842 | 27.500 | 8.500 | 13.200 | 56.900 | 121.500 | 138.300 | 103.200 | 85.700 | 64.600 | 36.700 | 24.200 |
| 1 | 1853 | 10.700 | 15 | 40.100 | 61.500 | 98.500 | 124.700 | 96.300 | 66.600 | 64.500 | 54.100 | 39 |
| 1 | 1864 | 20.600 | 6.700 | 4.300 | 22.700 | 54.800 | 93.800 | 95.800 | 77.200 | 59.100 | 44 | 47 |
| 1 | 1875 | 30.500 | 16.300 | 7.300 | 37.600 | 74 | 139 | 111.200 | 101.600 | 66.200 | 44.700 | 17 |

The parameters set for the `Windowing` operator to achieve this is shown in the following screenshot:



The parameters **window size** and **step size** are both set to `11` in this case. The window size value has the effect of creating 11 new attributes with names ranging from **sunspots-10** to **sunspots-0**. The **step size** value dictates how many values to step forward to start each new window.

The first row corresponds to the first 11 entries in the input example set, which in this case are the years 1700 to 1710 (both inclusive). Close examination of the data reveals that the year 1700 had five sunspots and this corresponds to the value for **sunspots-10** in row one. The second row corresponds to the years 1711 to 1721 (both inclusive). The year 1711 had zero sunspots, and this corresponds to the value for **sunspots-10** in row **2**.

A label attribute can be added to the windows by checking the **create label** check box. This allows the value of another attribute to be used as a label, and the horizon parameter that appears in this case allows the value of the label to be chosen from an arbitrary point in the future.

The end result of this activity is an example set with examples corresponding (in this case) to the 11-year solar cycle. The process that does this is `manipulateSunspots.xml` and is available with the files that accompany this book.

# Summary

This chapter has introduced some more advanced techniques for transforming data into a format more suited to exploration and data mining. These included generating attributes based on other attributes in the same example as well as attributes in other examples through the use of macros. In addition to this, aggregation, pivoting, de-pivoting, and windowing were all discussed.

In my experience, a great deal of time is spent transforming data using these techniques, and it is worthwhile investing time learning how to use them.

The next chapter considers how to reduce data size by simple sampling methods or more complex model-based approaches. While on the face of it this may seem like a bad idea, the reality of real data is that there can be too much of it and it would take too long to process unless some reductions are done.

# 8
# Reducing Data Size

One definition of **Big Data** states that it is big if it is just at or beyond the edge of the capabilities of an organization to process it. There is always too much data, and human nature, being what it is, makes it inevitable that the boundaries of what is possible will be reached. The main problem is that more data takes more time to process. To a certain extent, more money, more computing power, and a more sophisticated parallel approach can help, but some data mining processes scale as the second, third, or worse power of the number of examples and attributes. Doubling the data size quadruples the runtime and there comes a point where there is not enough money or time to finish the job. Using techniques that scale linearly is possible for certain types of problems. However, it still costs money and effort to get there.

An important activity is to recognize this and find ways to reduce both the number of examples and attributes, while balancing this against the accuracy of predictions or modeling requirements. This chapter, therefore, discusses methods for reducing data size.

The chapter starts with methods for removing examples using the `Sample` operator and its variants. From there, the chapter progresses to methods to remove attributes, including removal of useless attributes and attribute weighting, and also illustrates model-based approaches.

## Removing examples using sampling

The `Sample` operators allow example subsets to be chosen, and there are a number of different techniques to use depending on what is required and the characteristics of the data.

A very common use of the operator is to simply reduce the size of data to test the flow of a complex process. If the full data consists of millions of rows, the execution time may be immense and it is annoying to find a bug right at the end of a long run. Reducing the data size to a few percent of the total allows bugs to be found early.

The `Sample` operator allows a proportion of an example set to be selected. There are three possible options. First, an absolute size of the result; second, a proportion of the example set; and third, a probability that an example will appear. The absolute size is useful when a fixed number of examples is required, while the proportion is useful if a fixed percentage of the whole is required. Probability is similar to the proportion option but considers each example, and based on the probability, causes it to be filtered, which can lead to a different number of examples when compared to the proportion case. If the data contains a label, it is possible to balance the proportion of labels in the generated data and specifically choose the proportion of examples within the filtered example for different values of the label. This is useful if you have data where one label dominates, leading to class imbalances; the sampling can even up the class distribution. The `Sample (Stratified)` operator can also be used to create a sample where the proportion of label values in the sampled data matches the original data.

The `Sample (Bootstrapping)` operator is used to build datasets that are larger than the original dataset. It does this by sampling with replacement. At first sight, this may seem pointless but when faced with a dataset with a large class imbalance, it is often important to build training sets that have an equal class balance. This is done by bootstrapping the original data to increase it in size until the desired number of examples of one class are present. From there, the example set is sampled, so different label proportions appear in the result.

A process called `sampleExamples.xml` is provided with the files that accompany this book. This contains examples of all the sample operators described in the previous paragraphs.

Sampling inevitably introduces errors. The size of the error will be driven completely by the data exploration and mining processes that are being performed, and investigation and analysis will be needed to estimate or measure errors.

# Removing attributes

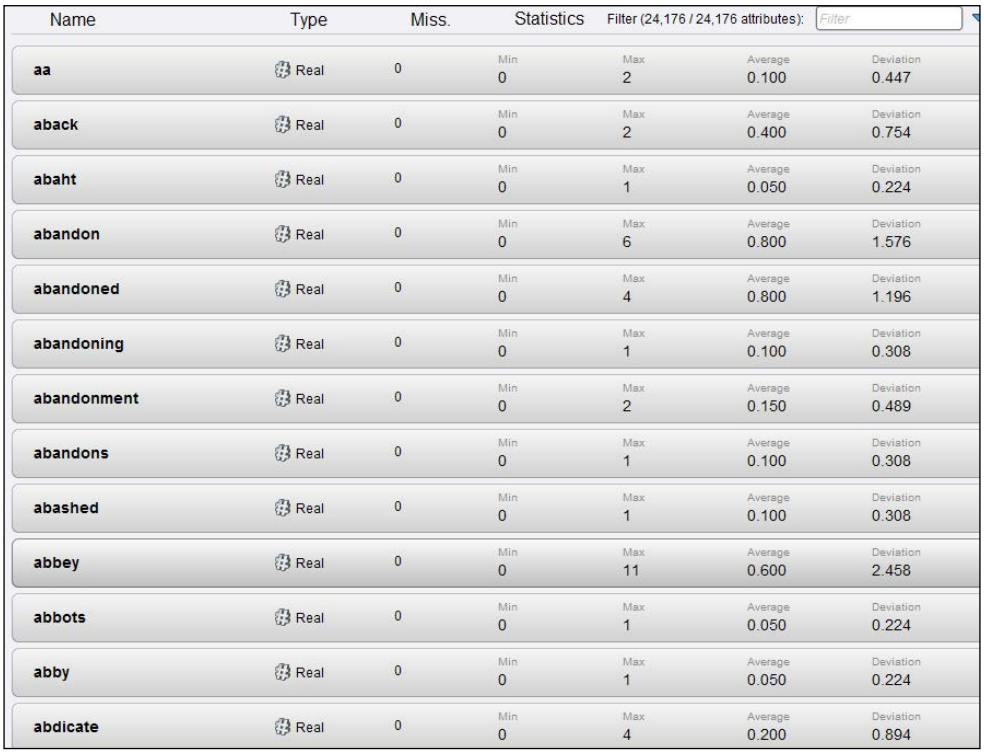
Three different techniques for removing attributes are illustrated in the following sections. These are as follows:

- Remove useless attributes by employing simple statistical techniques.
- Weighting, which determines how much influence or weight an individual attribute has on the label. The assumption in this case is that the data is being used for a classification problem and the removal of attributes will speed up the modeling process but reduce the accuracy.
- Model-based, which uses a classification model to determine the most predictive attributes of the label. As with weighting, the assumption is that the data is being used for classification.

# Removing useless attributes

The `Remove Useless Attributes` operator is well named but it is worth understanding how it works to ensure that useful attributes are not accidently removed.

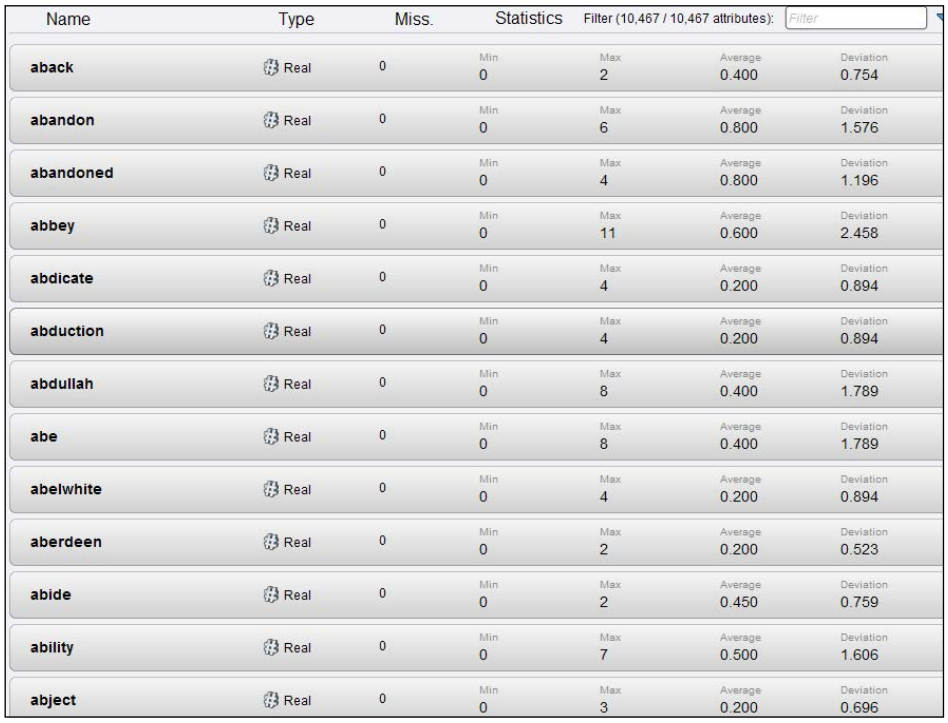
The following screenshot shows **Statistics View** for the first few attributes of a document vector containing **24176** attributes (refer to the process, `reduceLargeDocumentVector.xml`).



| Name | Type | Miss. | Statistics | Min | Max | Average | Deviation |
|---|---|---|---|---|---|---|---|
| aa | Real | 0 | | 0 | 2 | 0.100 | 0.447 |
| aback | Real | 0 | | 0 | 2 | 0.400 | 0.754 |
| abaht | Real | 0 | | 0 | 1 | 0.050 | 0.224 |
| abandon | Real | 0 | | 0 | 6 | 0.800 | 1.576 |
| abandoned | Real | 0 | | 0 | 4 | 0.800 | 1.196 |
| abandoning | Real | 0 | | 0 | 1 | 0.100 | 0.308 |
| abandonment | Real | 0 | | 0 | 2 | 0.150 | 0.489 |
| abandons | Real | 0 | | 0 | 1 | 0.100 | 0.308 |
| abashed | Real | 0 | | 0 | 1 | 0.100 | 0.308 |
| abbey | Real | 0 | | 0 | 11 | 0.600 | 2.458 |
| abbots | Real | 0 | | 0 | 1 | 0.050 | 0.224 |
| abby | Real | 0 | | 0 | 1 | 0.050 | 0.224 |
| abdicate | Real | 0 | | 0 | 4 | 0.200 | 0.894 |

Each attribute is a real number and the average and standard deviation of 20 examples in the example set are shown in the previous screenshot. The `numerical min deviation` parameter of the `Remove Useless Attributes` operator causes an attribute to be removed if its standard deviation is less than or equal to the parameter. For example, in the previous screenshot, if the parameter is set to `0.5`, of the first six attributes, the `aa`, `abaht`, and `abandoning` attributes would be removed while `aback`, `abandon`, and `abandoned` would not.

The result of applying this to the example set is shown in the following screenshot:



| Name | Type | Miss. | Statistics | Filter (10,467 / 10,467 attributes): Filter | | | |
|---|---|---|---|---|---|---|---|
| aback | Real | 0 | Min 0 | Max 2 | Average 0.400 | Deviation 0.754 | |
| abandon | Real | 0 | Min 0 | Max 6 | Average 0.800 | Deviation 1.576 | |
| abandoned | Real | 0 | Min 0 | Max 4 | Average 0.800 | Deviation 1.196 | |
| abbey | Real | 0 | Min 0 | Max 11 | Average 0.600 | Deviation 2.458 | |
| abdicate | Real | 0 | Min 0 | Max 4 | Average 0.200 | Deviation 0.894 | |
| abduction | Real | 0 | Min 0 | Max 4 | Average 0.200 | Deviation 0.894 | |
| abdullah | Real | 0 | Min 0 | Max 8 | Average 0.400 | Deviation 1.789 | |
| abe | Real | 0 | Min 0 | Max 8 | Average 0.400 | Deviation 1.789 | |
| abelwhite | Real | 0 | Min 0 | Max 4 | Average 0.200 | Deviation 0.894 | |
| aberdeen | Real | 0 | Min 0 | Max 2 | Average 0.200 | Deviation 0.523 | |
| abide | Real | 0 | Min 0 | Max 2 | Average 0.450 | Deviation 0.759 | |
| ability | Real | 0 | Min 0 | Max 7 | Average 0.500 | Deviation 1.606 | |
| abject | Real | 0 | Min 0 | Max 3 | Average 0.200 | Deviation 0.696 | |

The total number of attributes has reduced to **10467**. The rationale behind this approach is that an attribute with a smaller relative standard deviation has less variation when compared to the other attributes. Therefore, it is more likely to be less influential if the data is being used for classification. This approach is likely to be invalid if the attributes are not normalized. This can be understood by considering two attributes. The first attribute has a range between 0 and 1 and the second, a copy of the first, is scaled by a factor of 1,000. The standard deviation of the scaled attribute will be 1,000 times larger than the original, and the `Remove Useless Attributes` operator will choose to keep the scaled version in preference to the original. Extending this to different attributes, we can see that attributes with larger ranges will have larger absolute standard deviations and will consequently not be marked as being useless.

One important point is that if the standard deviation of an attribute is 0, there is no variation and the attribute has the same value for all the examples. In this case, the attribute will not have an impact on the result of a classification, adds nothing to understanding differences between examples, and can safely be deleted. This is the default setting for the operator.

When the data contains nominal attributes, the `nominal useless above` and `nominal useless below` parameters can be used. For each attribute, the operator determines the proportion of the most frequent nominal value. If this proportion is greater than or equal to the `nominal useless above` parameter, the attribute is deleted. This gives the possibility that attributes with one dominant nominal value can be deleted since they are less likely to be predictive. When this parameter is set to `1`, the default attributes that have only a single nominal value are deleted.

The `nominal useless below` parameter allows attributes with too many nominal values to be deleted. If the most common attribute is present in fewer examples than the proportion given by the parameter, it is likely that the different nominal values are too numerous. For example, if there are 100 examples and one particular attribute has 100 different nominal values, the proportion of the most common nominal value will be 0.01. Setting the `nominal useless below` parameter to this will cause the attribute to be deleted. The `nominal remove id like` parameter is a shortcut to this.

Generally speaking, this operator is difficult to use because of the difficulty of knowing the impact of setting the various thresholds incorrectly. A feedback loop would be required to check the impact. Nonetheless, the default behavior is very useful and, additionally, the ability to quickly remove attributes that do not vary greatly can be a useful way of understanding the data.

# Weighting attributes

When building classifiers or performing unsupervised clustering, the number of attributes can profoundly affect the processing time required. Weighting is a technique to rank attributes that have the most influence on the label or are most correlated with the principal components that explain the variation within the data. The attributes with the most weight can then be retained and the effect on model accuracy can be measured, and this can be balanced against processing time. In some cases, elimination of low-weight attributes can even improve the performance of classification.

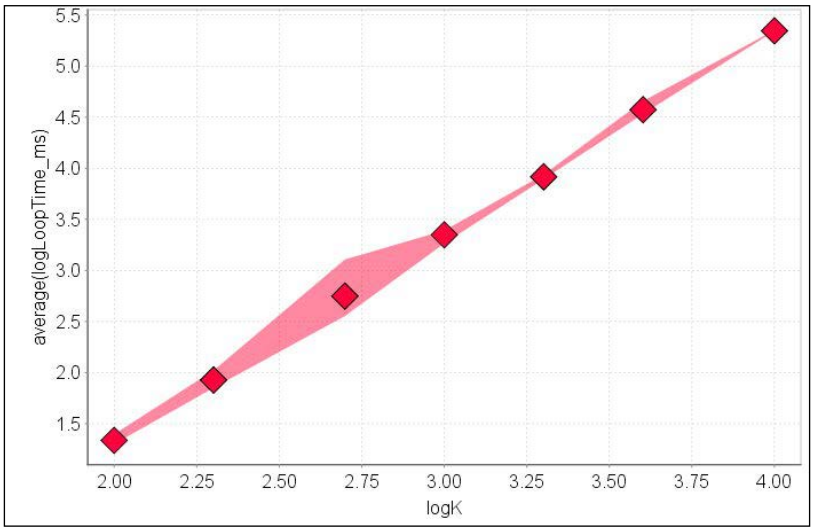
In addition, it can be very difficult to see which attributes are predictive of the class label when building classifiers. This is especially difficult if there are thousands of attributes in the example set. Some classifiers such as the various types of decision trees or rule induction produce a model that can be read by a person. Many others don't and weighting gives the possibility to eliminate attributes that appear not to be important while determining their effect on model accuracy. More predictive attributes can then be seen and this gives a domain expert the opportunity to focus on these as part of the route to a greater understanding of the data.

Refer to `weightLargeLabelledDocumentVector.xml` for an initial example process that performs a simple weighting using correlation as well as a `select by weight` operator (explained later) to reduce the number of attributes.

There are a number of different weighting algorithms, and the precise details of how they work are beyond the scope of this book. It is important, however, to understand how much time some of the operators need because some are quicker than others.

Of the most commonly used weighting methods, weighting by correlation and chi-squared statistics is usually the quickest. Weighting by information gain and information gain ratio are slightly slower, and weighting by **Principal Component Analysis** (**PCA**) is usually the slowest.

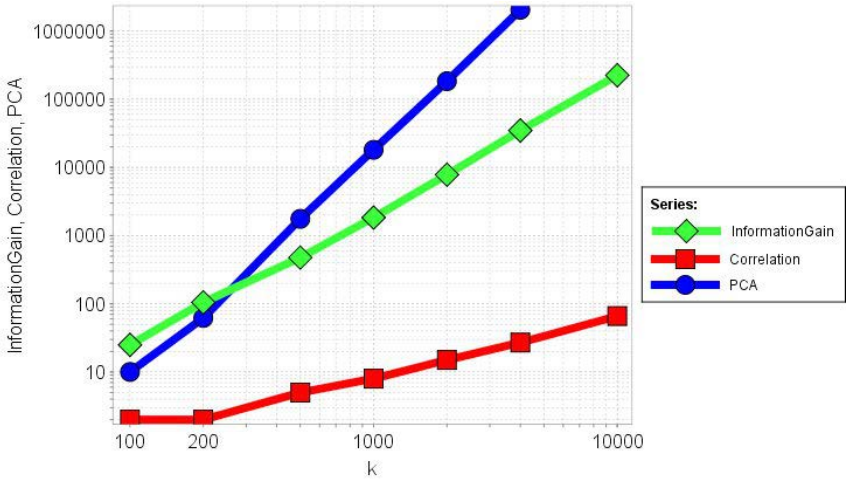
This is illustrated in the following screenshot that shows the time required for the `Weight by Information Gain` operator to run, as the number of attributes is varied (the number of examples is fixed at 20; the process and method is described in *Chapter 9*, *Resource Constraints*). The bands on the graph represent the minimum and maximum performance for multiple runs. This is because in general, performance measurements vary between runs as a result of differences in the computing environment. It is important to provide enough results to ensure a degree of statistical significance.




The results are plotted on a log-log plot and show that $k = 10{,}000$ (10 raised to the fourth power) attributes require about 200,000 ms (10 raised to the power 5.3) to process, which is about 3 minutes 20 seconds. This would obviously be different if a different computer was used and the number of examples differed.

Given the straight line, we can estimate the time needed for a larger number of attributes. When the number of attributes is 100,000, the estimated time would be about 7 hours. At 1 million attributes (assuming we could even process this number with the resources available), the time would be of the order of 1 month.

A comparison between three different weighting methods is shown in the following graph, which shows the number of attributes along the x axis and the time in milliseconds along the y axis.




As seen in the previous screenshot, the `correlation` method is the most rapid and `PCA` is the slowest. Projecting the graphs forward, it is possible to infer that the `PCA` approach would require 11 days for 10,000 attributes, 65 years for 100,000, and 141,000 years for 1 million. Clearly, `PCA` must be handled with care because it will quickly become difficult to use for fairly normal-sized example sets.

Having produced a set of weights, they can be used to select attributes through the use of the `Select by Weight` operator. This operator requires a weight relation to be chosen by the user, and it uses this to select attributes within the example set. The possible values for the weight relation are given as follows (this text has been copied from the online help, the RapidMiner GUI):

- `less_equals`: Attributes with weights equal to or less than the weight parameter are selected
- `less`: Attributes with weights less than the weight parameter are selected
- `top_k`: The *k* attributes with the highest weights are selected

- `bottom_k`: The *k* attributes with the lowest weights are selected
- `all_but_top_k`: All attributes other than the *k* attributes with the highest weights are selected
- `all_but_bottom_k`: All attributes other than the *k* attributes with the lowest weights are selected
- `top_p%`: The top *p* percent attributes with the highest weights are selected
- `bottom_p%`: The bottom *p* percent attributes with the lowest weights are selected

The typical method of using of this operator when building classifiers is to choose the top *k*, where *k* is a small number, and then to investigate how this affects accuracy as *k* is varied.

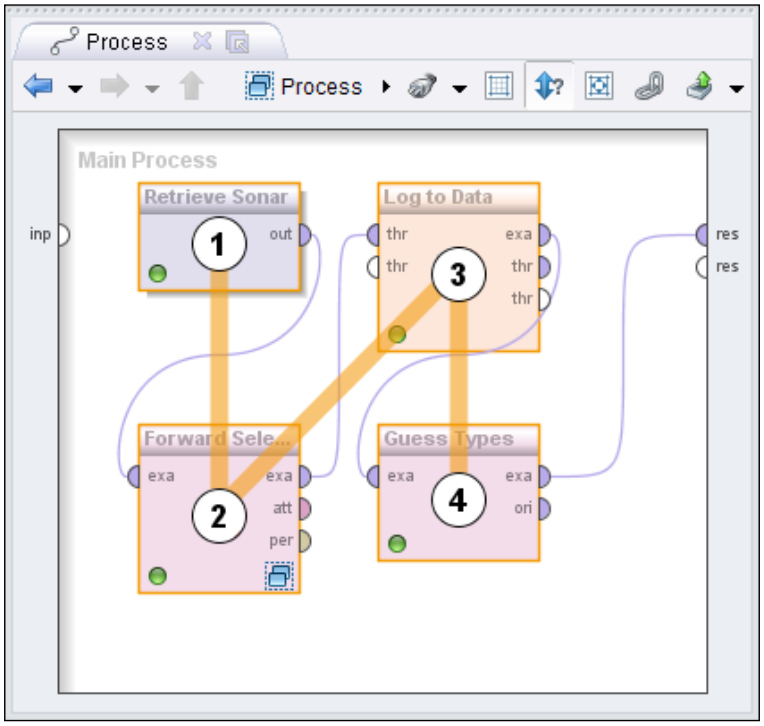
Selecting by weight is also useful when eliminating attributes from the test data, which were not used to build a model based on the training data. The basic approach is to create a set of weights using the `Data to Weights` operator based on the training data. This creates a set of weights set to 1 for all the attributes. These weights can then be used with the `Select by Weight` operator to eliminate any new attributes that may happen to find their way into the data mining operation.

# Selecting attributes using models

Weighting by the `PCA` approach, mentioned previously, is an example where the combination of attributes within an example drives the generation of the principal components, and the correlation of an attribute with these generates the attribute's weight.

When building classifiers, it is logical to take this a stage further and use the potential model itself as the determinant of whether the addition or removal of an attribute makes for better predictions. RapidMiner provides a number of operators to facilitate this, and the following sections go into detail for one of these operators with the intention of showing how applicable the techniques are to other similar operations. The operator that will be explained in detail is `Forward Selection`. This is similar to a number of others in the Optimization group within the Attribute selection and Data transformation section of the RapidMiner GUI operator tree. These operators include `Backward Elimination` and a number of `Optimize Selection` operators. The techniques illustrated are transferrable to these other operators.

A process that uses `Forward Selection` is shown in the next screenshot. This process is `optimize.xml` and is available with the files that accompany this book.



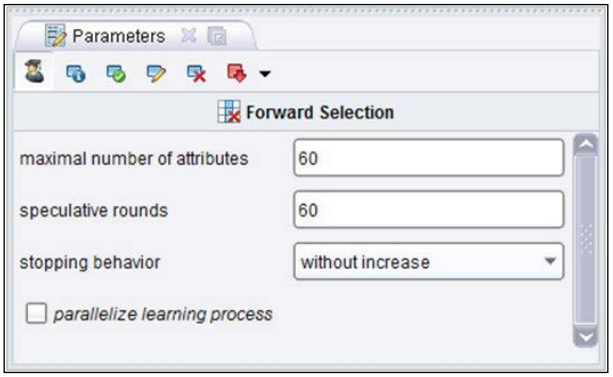

The `Retrieve` operator (labeled **1**) simply retrieves the sonar data from the local sample repository. This data has 208 examples and 60 regular attributes named `attribute_1` to `attribute_60`. The label is named `class` and has two values, `Rock` and `Mine`.

The `Forward Selection` operator (labeled **2**) tests the performance of a model on examples containing more and more attributes. The inner operators within this operator perform this testing.

The `Log to Data` operator (labeled **3**) creates an example set from the log entries that were written inside the `Forward selection` operator. Example sets are easier to process and store in the repository.

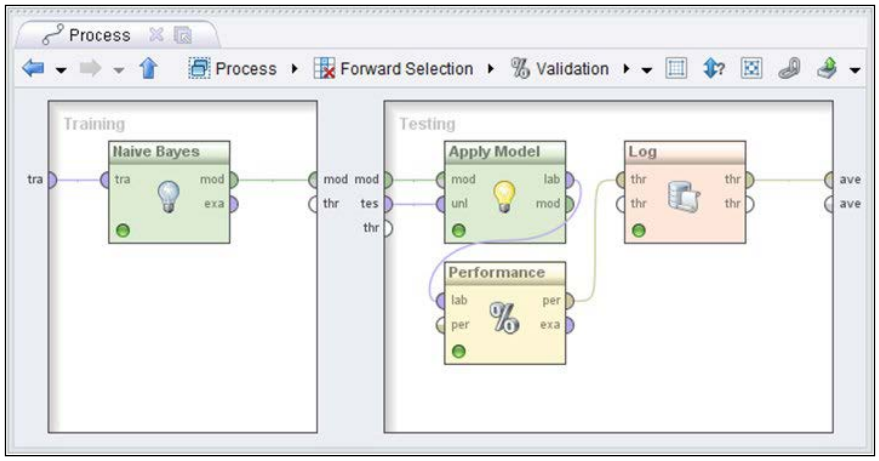
The `Guess Types` operator (labeled **4**) changes the types of attributes based on their contents. This is simply a cosmetic step to change real numbers into integers to make plotting them look better.

Now, let's return to the `Forward Selection` operator, which starts by invoking its inner operators to check the model performance using each of the 60 regular attributes individually. This means it runs 60 times. The attribute that gives the best performance is then retained, and the process is repeated with two attributes using the remaining 59 attributes along with the best from the first run. The best pair of attributes is then retained, and the process is repeated with three attributes using each of the remaining 58. This is repeated until the stopping conditions are met. For illustrative purposes, the parameters shown in the following screenshot are chosen to allow it to continue for 60 iterations and use all the 60 attributes.

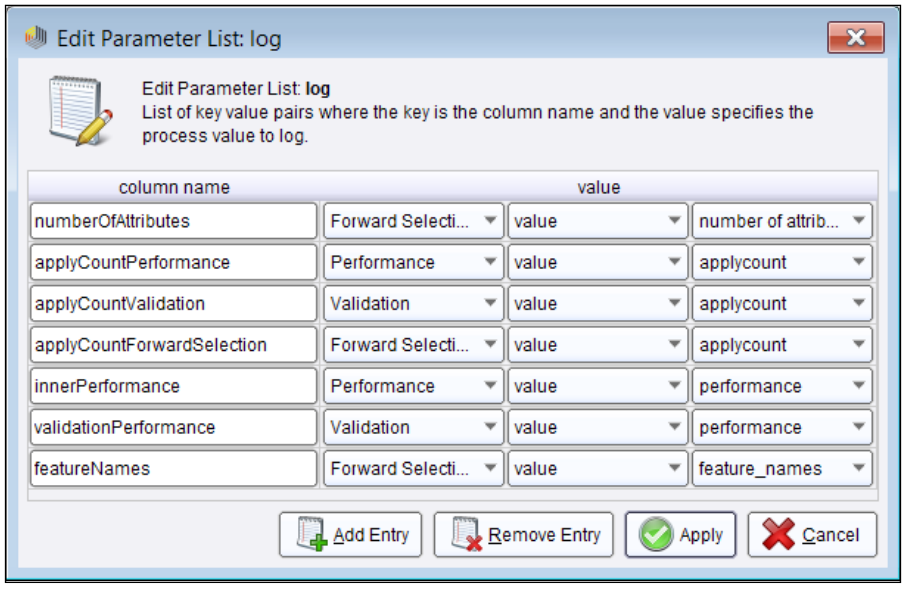



The inner operator to the `Forward Selection` operator is a simple cross validation with the number of folds set to three. Using cross validation ensures that the performance is an estimate of what the performance would be on unseen data. Some overfitting will inevitably occur, and it is likely that setting the number of validations to three will increase this. However, this process is for illustrative purposes and needs to run reasonably quickly, and a low cross-validation count facilitates this.

Inside the `Validation` operator itself, there are operators to generate a model, calculate performance, and log data. These are shown in the following screenshot:




The `Naïve Bayes` operator is a simple model that does not require a large runtime to complete. Within the `Validation` operator, it runs on different training partitions of the data. The `Apply Model` and `Performance` operators check the performance of the operator using test partitions. The `Log` operator outputs information each time it is called, and the following screenshot shows the details of what it logs.



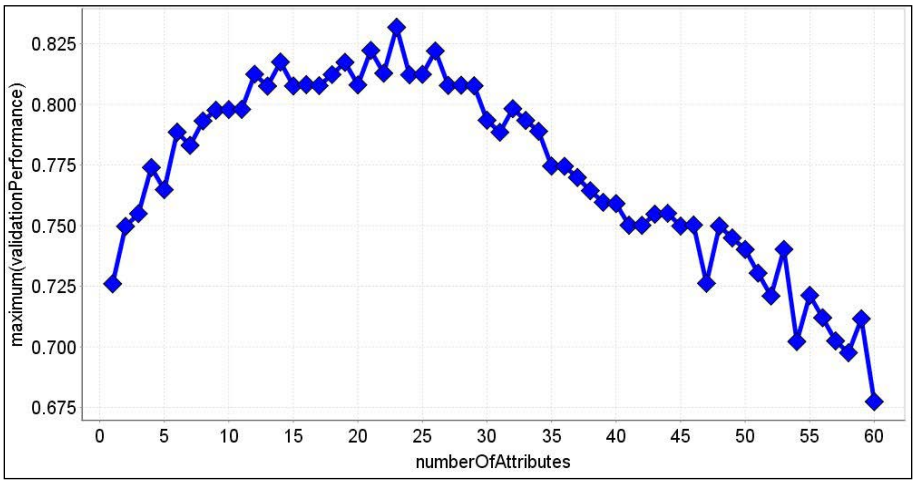
Running the process gives the log output as shown in the following screenshot:

ExampleSet (5490 examples, 0 special attributes, 7 regular attributes) Filter (5,490 / 5,490 examples): all

| ... | numberOfAttributes | applyCountPerfor... | applyCountValidati... | applyCountForwar... | innerPerformance | validationPerforma... | featureNames |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0.652 | ? | attribute_1 |
| 2 | 1 | 2 | 1 | 1 | 0.514 | ? | attribute_1 |
| 3 | 1 | 3 | 1 | 1 | 0.580 | ? | attribute_1 |
| 4 | 1 | 4 | 2 | 1 | 0.551 | 0.582 | attribute_2 |
| 5 | 1 | 5 | 2 | 1 | 0.557 | 0.582 | attribute_2 |
| 6 | 1 | 6 | 2 | 1 | 0.565 | 0.582 | attribute_2 |
| 7 | 1 | 7 | 3 | 1 | 0.565 | 0.558 | attribute_3 |
| 8 | 1 | 8 | 3 | 1 | 0.457 | 0.558 | attribute_3 |
| 9 | 1 | 9 | 3 | 1 | 0.580 | 0.558 | attribute_3 |
| 1 | 1 | 10 | 4 | 1 | 0.536 | 0.534 | attribute_4 |
| 1 | 1 | 11 | 4 | 1 | 0.557 | 0.534 | attribute_4 |
| 1 | 1 | 12 | 4 | 1 | 0.667 | 0.534 | attribute_4 |
| 1 | 1 | 13 | 5 | 1 | 0.449 | 0.587 | attribute_5 |
| 1 | 1 | 14 | 5 | 1 | 0.643 | 0.587 | attribute_5 |
| 1 | 1 | 15 | 5 | 1 | 0.667 | 0.587 | attribute_5 |
| 1 | 1 | 16 | 6 | 1 | 0.565 | 0.586 | attribute_6 |

It is worth understanding this output because it gives a good overview of how the operators work and fit together in a process. For example, the attributes **applyCountPerformance**, **applyCountValidation**, and **applyCountForwardSelection** increment by one each time the respective operator is executed. The expected behavior is that **applyCountPerformance** will increment with each new row in the result, **applyCountValidation** will increment every three rows, which corresponds to the number of cross validation folds, and **applyCountForwardSelection** will remain at **1** throughout the process. Note that **validationPerformance** is missing for the first three rows. This is because the validation operator has not calculated a performance yet. The first occurrence of the logging operator is called **validationPerformance**; it is the average of **innerPerformance** within the validation operator. So, for example, the values for **innerPerformance** are **0.652**, **0.514**, and **0.580** for the first three rows; these values average out to **0.582**, which is the value for **validationPerformance** in the fourth row. The **featureNames** attribute shows the attributes that were used to create the various performance measurements.

The results are plotted as a graph as shown:




This shows that as the number of attributes increases, the validation performance increases and reaches a maximum when the number of attributes is 23. From there, it steadily decreases as the number of attributes reaches 60.

The best performance is given by the attributes immediately before the maximum **validationPerformance** attribute value. In this case, the attributes are:

```
attribute_12, attribute_40, attribute_16, attribute_11, attribute_6,
attribute_28, attribute_19, attribute_17, attribute_44, attribute_37,
attribute_30, attribute_53, attribute_47, attribute_22, attribute_41,
attribute_54, attribute_34, attribute_23, attribute_27, attribute_39,
attribute_57, attribute_36, attribute_10.
```

The point is that the number of attributes has reduced and indeed the model accuracy has increased. In real-world situations with large datasets and a reduction in the attribute count, an increase in performance is very valuable.

# Summary

This chapter has covered the important topic of reducing data size by both the removal of examples and attributes. This is important to speed up processing time, and in some cases can even improve classification accuracy. Generally though, classification accuracy reduces as data reduces.

The next chapter continues the performance theme from a different angle and gives some methods for measuring and estimating the performance of processes containing sequences of operators.

# 9
# Resource Constraints

Processing large amounts of data requires a lot of physical processing power and memory, to say nothing of the amount of time needed for the processing. Sometimes, it is not possible to process the data using the available resources, and in this situation, some techniques can be adopted to induce the process to complete.

This becomes particularly important when building models where the processing time can depend exponentially on the number of attributes, the number of examples, and the model itself. It is therefore important to know how long something will take by doing a measurement on a smaller sample of the data. From there an estimate can be made of what performance will be for all the data, and then steps can be taken to improve performance if needed. This chapter is therefore structured as follows:

- Measuring and estimating performance
- Splitting data into batches
- Parallel processing

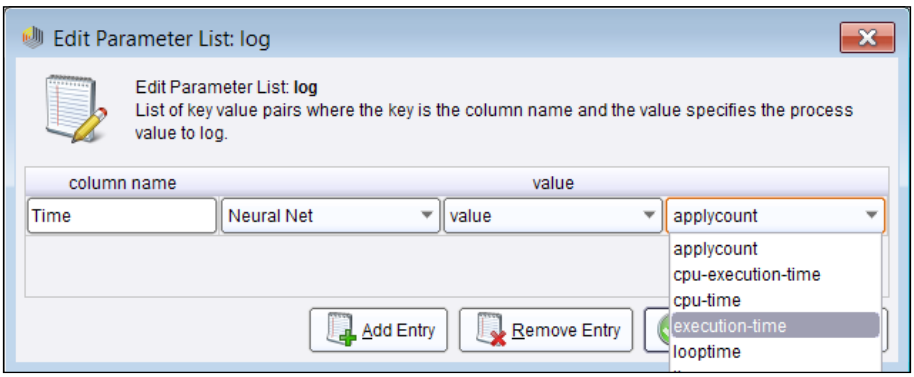
## Measuring and estimating performance

Often, when building a model or eliminating correlated attributes, I find that more than 10 minutes have elapsed, and out of frustration I stop the process. In reality, I may have no idea whether another 1 minute is needed or whether it will take a month. In fact, this is an important general point because large data and complex processing inevitably takes time. So, it is not an unreasonable question to ask how much processing time and what resources are needed in a production context. For example, a classification process might occasionally require that the classification model be recreated. It will be important to know how long this will take, so appropriate plans can be made.

Given that we have a data mining product in front of us, we can use it to predict how long something will take if we take some measurements.

# Measuring performance

It is very straightforward to measure how long an operator takes to execute. One simple approach is to use the `Log` operator to record time using the built-in values recorded by all operators. An example is shown in the following screenshot:




The dropdown has the following possible values that are relevant for time measurements:

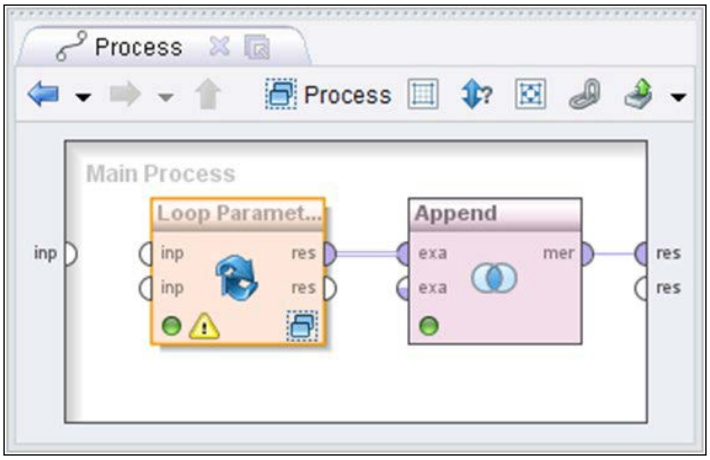
- **cpu-execution-time**
- **cpu-time**
- **execution-time**
- **looptime**
- **time**

Of these, `looptime` and `time` are measured in milliseconds and give a measure of how much time elapsed since the last time the operator was called. The **execution-time** measure is the time required to execute the operator itself, also measured in milliseconds, and is the one that is most useful.

By logging this data and converting it into an example set, calculations can be performed to see how an operator is performing. This is straightforward to implement and is potentially very accurate. For measurements where sequences of operators are involved and an approximate view of performance is acceptable, an alternative is possible. This alternative is to create a macro containing a timestamp immediately before the part of the process of interest and another immediately after. By subsequently using the `Generate Macro` operator, calculations can be done to determine a delta time between the start and stop timestamps.

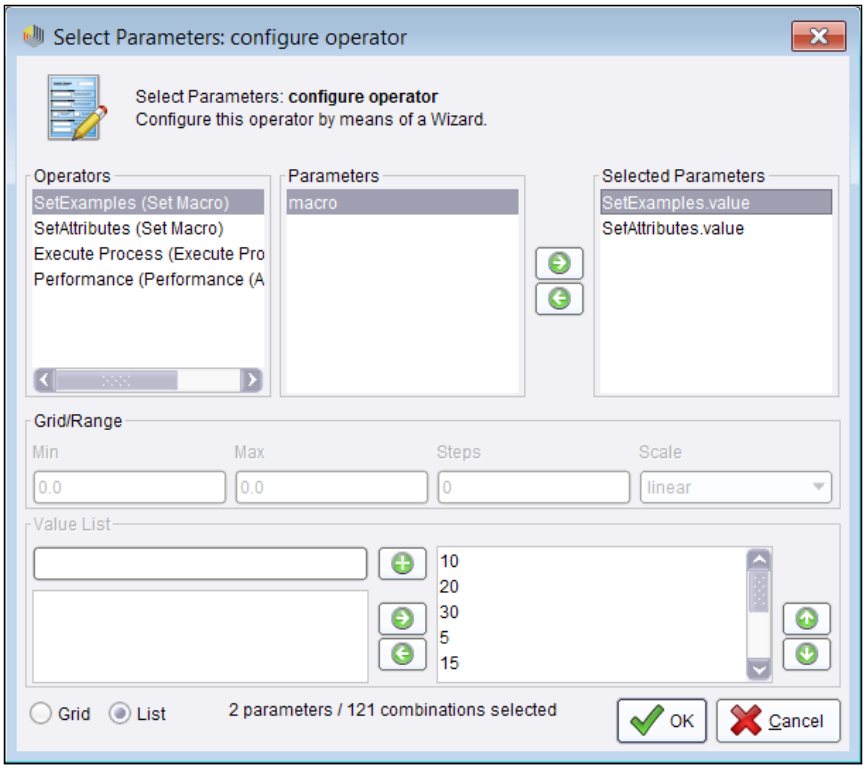
In addition to logging timings, it is important to record some starting environmental information—such as the number of attributes and number of examples being processed—that will potentially drive the elapsed time. By repeating the process for different starting conditions, a full set of training data can be obtained; this can be used to make a prediction about what performance would be with different numbers of attributes and examples.

An example process that can form the basis of such an investigation is shown in the following screenshot (the process is called `measurePerfBookVersion.xml` and is available with the files that accompany this book; note that the downloadable version has a few bonus features):



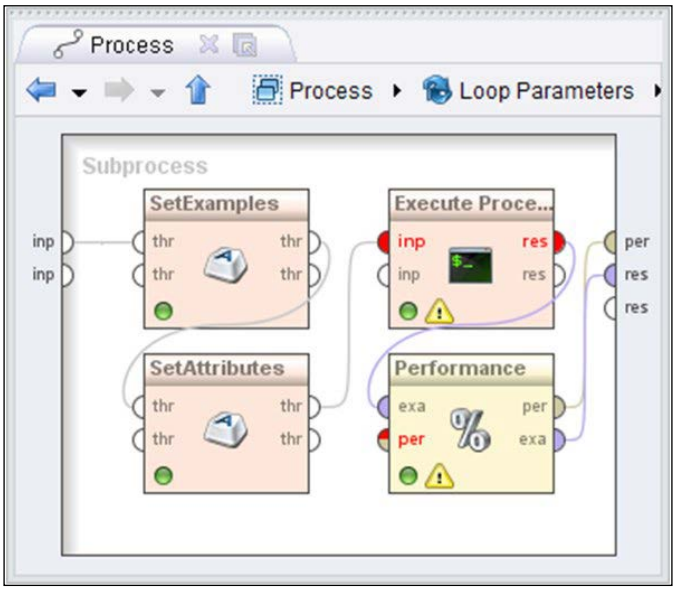

The `Loop Parameters` operator contains inner operators and is configured with lists of parameters for these inner operators. The loop operator iterates as many times as there are possible combinations of the inner operator parameters. The inner operators output example sets corresponding to the initial environment and to the time required for execution. The result of the `Loop Parameters` operator is a collection of example sets that can be combined into one using the `Append` operator.

To help understand the `Loop Parameters` operator, its parameters are shown in the following screenshot:



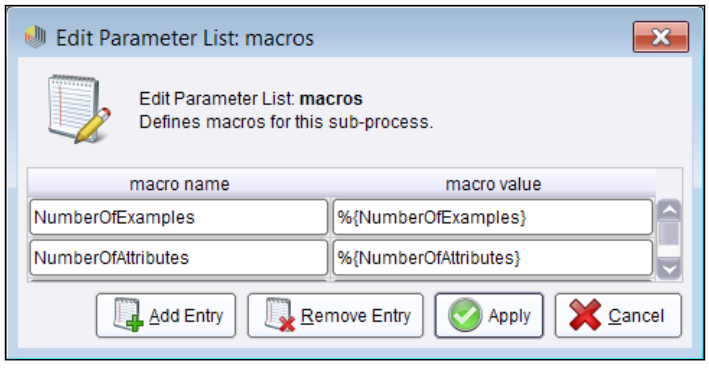

The top-left pane shows all the inner operators that are within the loop operator (these will be shown in the next screenshot). The top-right pane shows that one of the operators has been selected, and the bottom right pane shows the list of **Parameters** that will be applied in combination with the other parameters to produce a single run of the inner operators. In the example, the **SetExamples.value** operator has 11 possible values starting at 10, 20, and 30, as does the **SetAttributes.value** operator. The combination of 11 and 11 leads to 121 combinations—by simple multiplication—and this will be the number of times the loop will execute the inner operators.

Let's move on to the inner operators; these are shown in the following screenshot:




The first two of these—named **SetExamples** and **SetAttributes**—are `Set Macro` operators, and these `Set Macro` operators are used by the operator named `Execute Process`. The macros are called `NumberOfExamples` and `NumberOfAttributes` and the values for these are set by the loop operator. The **Performance** operator is simply present to provide a performance vector for the loop operator so that it functions properly. The performance vector is ignored and can be any convenient operator. I typically use the `Attribute Count` performance operator.

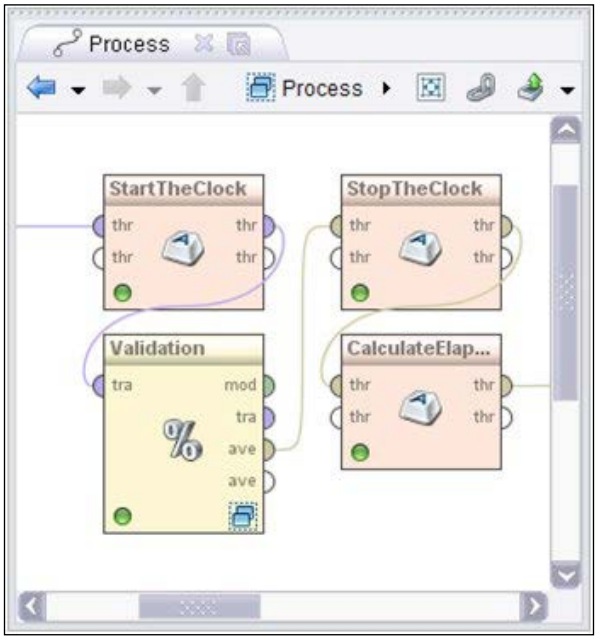
The macros used by the `Execute Process` operator and the parameters for this are shown in the following figure:



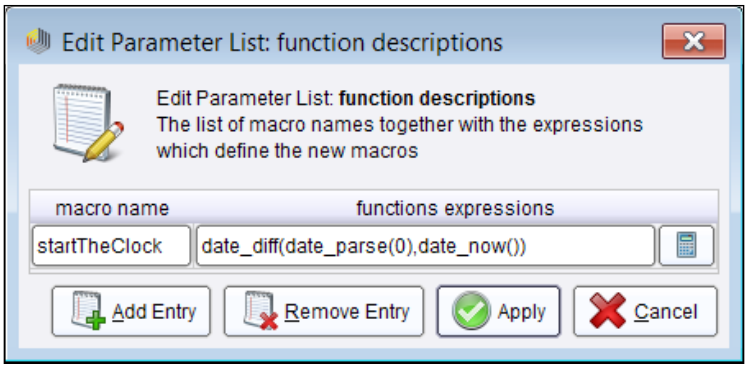
The `Execute Process` operator runs another process that has previously been created using RapidMiner and has been saved in the repository (the process is `dataGeneratorAndModel.xml`). This ability to execute processes is a very powerful and modular technique that facilitates the building up of libraries of processes to be used in different situations. Such a process can take named macros as parameters. The names of the macros that will be used inside the `Execute Process` operator are shown on the right-hand side of the preceding figure and their values are shown on the left. The values are taken from the current value of the macros defined within the loop operator. By virtue of being inside the loop operator, the process is executed multiple times with different parameters (in this case 121 times).

The process that is executed takes care of performing the modeling or other time consuming processes, and this example returns an example set with one row that contains an attribute for the number of examples, another for the number of attributes, and one more for the measured elapsed time.

The operators at the heart of the whole process are shown in the following screenshot:

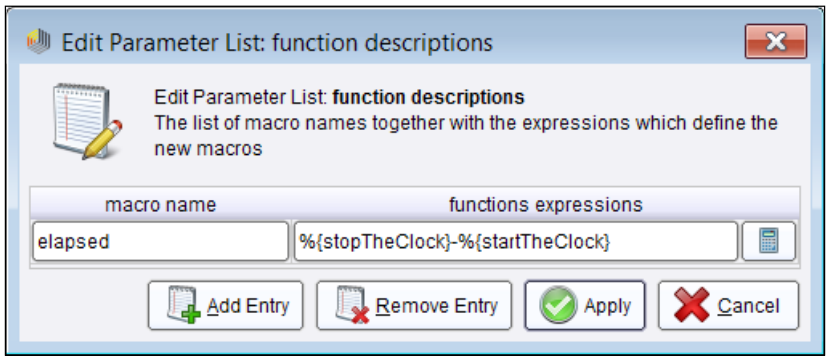



In this example, the operator performing the time consuming task is called `Validation`. The other three are `Generate Macro` operators and are extremely simple. The `StartTheClock` operator creates a macro called **startTheClock** as shown in the following image:




The function calculates the difference in milliseconds between the present time and the time at the beginning of the UNIX epoch.

The operator called `StopTheClock` creates a macro—creatively called `stopTheClock`—using the same function expression, and finally, the calculation of the elapsed time is done in the operator `CalculateElapsedTime` by generating a macro called `elapsed`, which simply subtracts the two macros. This is shown in the following image:



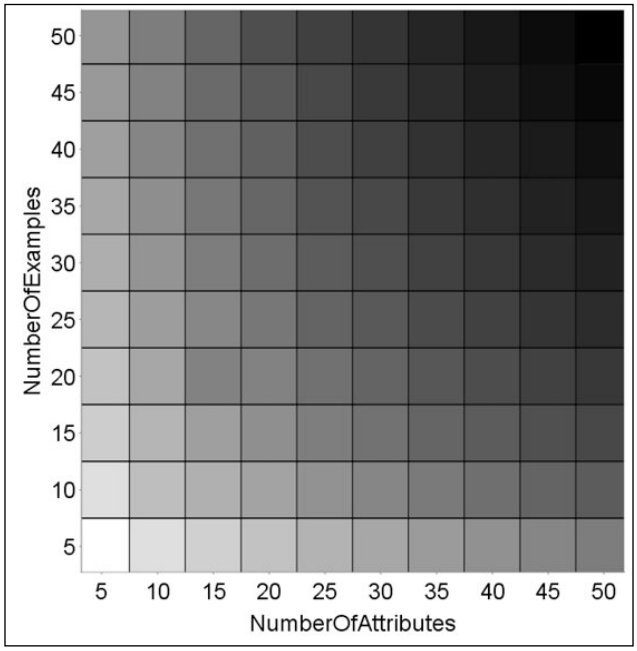

All macros can be recorded in the log by first using the `Provide Macro as Log Value` and then simply logging the macro's value from this operator. The log file can then be converted to an example set using the `Log to Data` operator.

The end result of this is an example set that looks something similar to the following table:

ExampleSet (121 examples, 0 special attributes, 3 regular attributes)

| Row No. | ElapsedTime | NumberOfAttributes | NumberOfExamples |
|---|---|---|---|
| 37 | 105 | 5 | 5 |
| 34 | 205 | 5 | 10 |
| 38 | 285 | 5 | 15 |
| 35 | 390 | 5 | 20 |
| 39 | 556 | 5 | 25 |
| 36 | 566 | 5 | 30 |
| 40 | 669 | 5 | 35 |
| 41 | 765 | 5 | 40 |
| 42 | 856 | 5 | 45 |
| 43 | 1096 | 5 | 50 |
| 44 | 1992 | 5 | 100 |

Visual inspection is often revealing, and this data can be plotted as a block plot as shown in the following figure:



This data shows the time performance of a neural network as it models data of different sizes. The color of a block is a measure of the elapsed time for the execution of a neural network as a function of `NumberOfAttributes` and `NumberOfExamples`. A darker shade means more elapsed time, and the range is 219 ms at the bottom-left and 7,920 ms at the top-right.

From here, it is quite simple to estimate the performance as the number of attributes or examples increases. Of course, it would also be possible to fit some sort of function to the data to make a prediction. It is beyond the scope of this book to go into these details, but from the data we just saw, a model I made predicted 94 seconds to process 100 attributes and 100 examples, 3,910 seconds to process 200 attributes and 200 examples, and 1.5 million years to process 1,000 attributes and 1,000 examples. The model turned out to be inaccurate because when I re-ran it with training data that explicitly had 100 examples and 100 attributes, the actual time was near 55 seconds and the prediction for 1,000 by 1,000 was near 10,000 years. Nonetheless, this is still a long time and there is enough accuracy to illustrate the main point.

# Adding memory

Adding more memory often speeds things up and allows some processes to complete. It is always worth checking to ensure that you are using the maximum amount of memory that is available.

The first thing to say is that 32-bit operating systems can generally address a maximum of 4 GB, so it is always worth getting a 64-bit version where this limit is much higher. A suitable processor on which the operating system can be run is also required. The general rule is to go for the 64-bit architecture; however, you can consult an expert to get clarity. Secondly, RapidMiner Studio can be run in a number of ways. I generally launch the GUI from a batch file because this gives more control, particularly in a Windows environment, where a separate console is launched. The log information is written to this console. This can be very useful if things become unresponsive. The file to launch the GUI is called `RapidMiner-Studio.bat` on Windows machines and is located in the `scripts` folder where RapidMiner is installed. In Linux environments, it is called `RapidMiner-Studio.sh`.

By defining the JAVA environment variable `MAX_JAVA_MEMORY`, it is possible to change the amount of memory RapidMiner uses. For example, to get 4 GB of memory, this variable would be set to 4,096 via the Control Panel in Windows or an appropriate configuration change in UNIX. If you find that processes are running slowly and the system monitor in the GUI shows memory getting low in RapidMiner, it is always worth setting the upper limit to the highest value that you can. Buying more memory is also a good option.

However, there will always be a time when you won't have enough memory and the question will arise as to what to do then.

# Parallel processing

If faced with a process that is simply taking too long, clearly more memory can help—as already discussed. If that fails, a more powerful processor is obviously something to consider. If there is not enough money to do that or if it simply does not work, a parallel approach can be considered. Some operators can be run in parallel, and RapidMiner Studio allows this to be done where the processor has two cores. To take advantage of this, it is necessary to download the parallel processing extension available from the Rapid-I Marketplace. Once this is done, a new configuration option checkbox appears on some operators; it allows them to be executed in parallel. Affected operators include the main process operator, looping operators, the subprocess operator, the branch and select operators, and the process evaluation operators. Typically, an operator that contains a loop or an implied subprocess can be made to run in a parallel fashion. Operators such as those in the `Modeling` and `Data Transformation` groups do not have this option.

For example, the `X-Validation` operator allows the partitions containing training and testing to be run in parallel. This is possible because the cross validation operation is inherently parallel as the individual partitions are self-contained and do not depend on each other.

Examples of processes that could not be carried out in parallel would include ones where calculations that require all the data are being performed. For example, normalizing an attribute within an example set requires all the data to be processed to determine various statistics to then apply to the individual examples.

In the context of exploring data, some activities could be carried out in parallel. For example, if multiple files are to be read in and processed so that the processing of one file depends only on the contents of that file, it would be possible to take advantage of parallel execution. The simplest possible process would be two `Read CSV` operators reading two files. If these are placed in the main process and the `parallelize main process` option is set to `true`, RapidMiner will execute the file reading across the available CPUs.

A word of caution about parallel processing. Even if the process can be done in a parallel fashion, there is still a risk that one instance will interfere with another. Perhaps macros are shared between instances, or it could be that the data is shared. Either way, this can cause processes to fail, so be careful.

# Restructuring processes

It is also always worth seeing whether the process can be restructured to be more efficient. A review of a process may reveal that expensive operations are being performed repeatedly and unnecessarily. In this situation there are some operators that can help.

The `Store` and `Retrieve` operators allow objects to be stored in the repository. Objects such as example sets, models, and weights can be stored and retrieved in this way. Typically, a process will perform an expensive operation once and store it in the repository. Subsequent operators can retrieve the object whenever it is needed. These operators can also be used to implement a checkpoint regime within a big process. This is relatively complex to set up, but it may be worth having the process determine where it reached in a long processing step so that it starts up where it left off.

The `Recall` and `Remember` operators are similar except they do not persist data to the repository. These can be used in a way similar to the `Store` and `Retrieve` operators except that the type of the object must be specified, and when using the `Recall` operator, the option to remove it from the store can be specified. These operators will consume memory but are likely to be relatively quicker than those that interact with the repository.

# Summary

It is part of human nature to always push the limits of what is possible, and so it is inevitable that you will encounter performance problems. This chapter has given some insight into ways to measure performance and some basic approaches to improve it.

The next chapter gives some tips to help debug processes when things aren't going your way.

# 10
# Debugging

When things aren't going your way, it is very important to have good tools to debug processes and their interaction with external entities, as well as have powerful supporting tools to mimic parts of the process execution.

This chapter will cover details of the tools, resources, and techniques that I have found to be very useful.

## Breakpoints in RapidMiner Studio

The RapidMiner Studio GUI has a very easy-to-use debugging facility. It is possible to add a breakpoint to any operator that will display results on completion of the operation. A breakpoint is enabled by selecting the operator and pressing the *F7* key. A right-click brings up a menu as an alternative way of doing this and it can be accessed from the **Edit** menu of the main GUI. When the process execution reaches the breakpoint, all the outputs from the operator can be examined to determine what is happening.

It is also possible to set a breakpoint before the execution of the operator. Again, select the operator and now press *Shift* + *F7* to set this type of breakpoint. This time all the inputs to the operator can be examined.

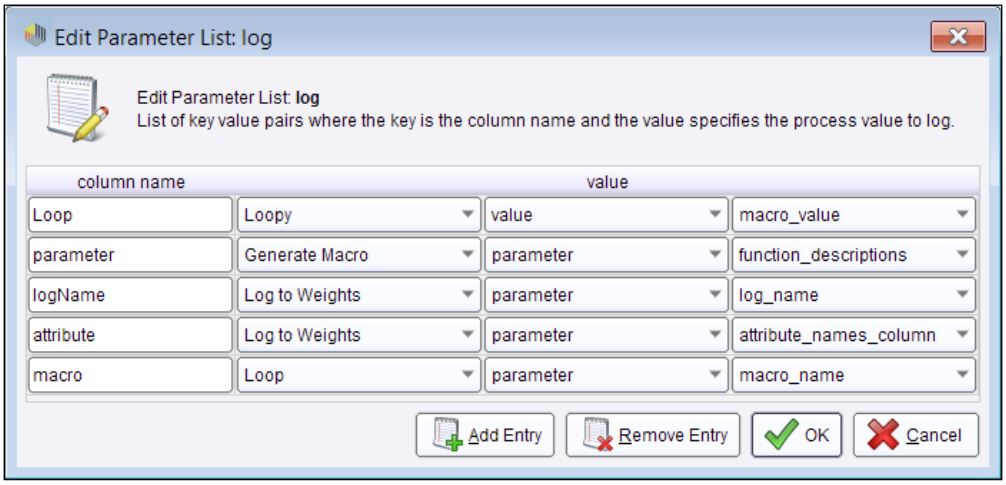
To continue from a breakpoint, press the Run button, or press *F1,* or use the **Process** menu from the main GUI.

A neat feature is the ability to change the value of a macro when a breakpoint has been reached so that the subsequent operators can use this value. This is done by displaying the **Macros** view and simply making the change to the macro value at the breakpoint and resuming the process. It is also possible to make changes to the process itself by selecting the operator that is to be executed and changing the parameters and even connections; however, this can get difficult to follow.

# Logging data in RapidMiner Studio

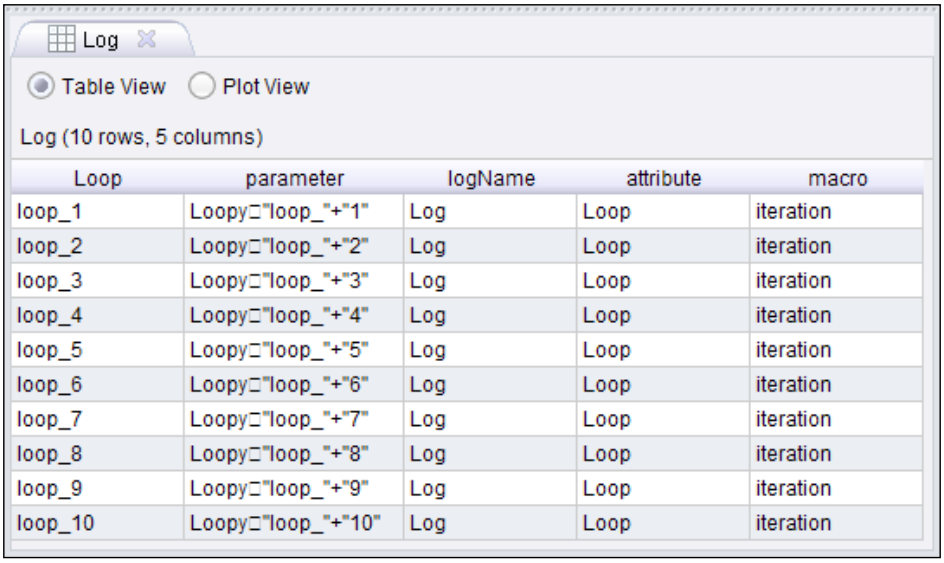
RapidMiner Studio provides the `Log` operator, which we have already seen being used in the previous chapters. Of all the operators, this is the one that I use a great deal, both for debugging and for creation of data.

Dealing with logging first, the `Log` operator can be inserted anywhere in a process and is configured to output the parameters or values associated with another operator somewhere in the process. For example, the screenshot that follows shows some example parameters for the `Log` operator:




The left-most column becomes the **column name** in the log, the second column is the name of the operator within the process, the third column is the type of information (either **value** or **parameter**), and the final column is the name of the information to log, which is filled in automatically with valid options by the RapidMiner Studio GUI. The **value** option is used to log the result of the execution of an operator, whereas the **parameter** option is used to log the parameters to the operator that controls its working.

The values of macros can be output using the `Provide Macro as Log Value` operator and this operator is then referred to by its name in the `Log` operator. An example of the output produced from the preceding screenshot is shown in the next table. The operator called **Loopy** is providing the value of a macro and this is accessed in the `Log` operator by the selection of the macro value. The other entries are all derived from the parameters for various operators.




| Loop | parameter | logName | attribute | macro |
|---|---|---|---|---|
| loop_1 | Loopy⊐"loop_"+"1" | Log | Loop | iteration |
| loop_2 | Loopy⊐"loop_"+"2" | Log | Loop | iteration |
| loop_3 | Loopy⊐"loop_"+"3" | Log | Loop | iteration |
| loop_4 | Loopy⊐"loop_"+"4" | Log | Loop | iteration |
| loop_5 | Loopy⊐"loop_"+"5" | Log | Loop | iteration |
| loop_6 | Loopy⊐"loop_"+"6" | Log | Loop | iteration |
| loop_7 | Loopy⊐"loop_"+"7" | Log | Loop | iteration |
| loop_8 | Loopy⊐"loop_"+"8" | Log | Loop | iteration |
| loop_9 | Loopy⊐"loop_"+"9" | Log | Loop | iteration |
| loop_10 | Loopy⊐"loop_"+"10" | Log | Loop | iteration |

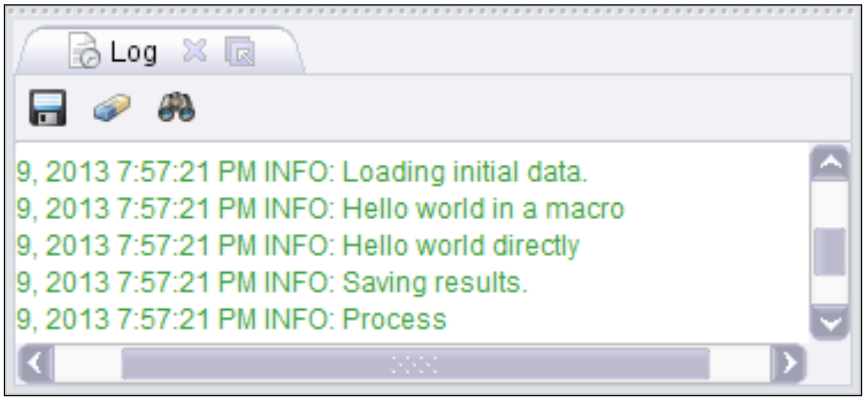
The value of a specific example of an attribute within an example set can be output using the `Extract Log Value` operator and as with the `Provide Macros as Log Value` operator, this is then accessed in the `Log` operator.

The resulting log can be converted to an example set if a more permanent record needs to be kept for further analysis or if the example set needs to be used later as part of the normal processing being done by the process.. This is done using the `Log to Data` operator.

# RapidMiner Studio console printing

RapidMiner Studio provides the `Print to Console` operator. This is a very simple operator that outputs a value to the **Log** view (enabled from the GUI **View** menu). This value can be a macro or a string. As a process executes, its progress can be monitored by observing the **Log** view.

The following screenshot shows a small portion of a logfile corresponding to the output from two `Print to Console` operators. The first operator logs the value of a macro, whereas the second operator outputs the text directly:




The text **Hello world in a macro** is the content of a macro, whereas **Hello world directly** is the text entered into the `Log to Console` operator. The process to do this is called `consoleLog.xml` and is available with the files that accompany this book.

# Groovy scripts

RapidMiner comes with Groovy already installed. This is a scripting language that is extremely close to Java in syntax and a process can run Groovy scripts by using the `Execute Script` operator. Groovy is very powerful and there are times when nothing else would do. Groovy can be used to output detailed information about anything in the RapidMiner Studio environment and this can be very useful when debugging.

# Outputting macros example

The following Groovy script outputs the values of all the macros defined in the process to the console. This is extremely valuable for debugging. A recent release of RapidMiner Studio provide a neat GUI interface that provides the same information; however, it may still be valuable to have a more permanent record for detailed diagnosis. The following code shows the Groovy script that would be used with the `Execute Script` operator:

```
import com.rapidminer.tools.Ontology;
String[] strings= new String[2];
for (String macroName : operator.getProcess().macroHandler.getDefinedMacroNames()) {
    String macroValue = operator.getProcess().macroHandler.getMacro(macroName);
    strings[0] = macroName;
    strings[1] = macroValue;
    operator.logNote ("Macro name | value:  " + macroName + " | " + macroValue);
}
return;
```

The line containing `getDefinedMacroNames` finds all the macros; the name-value pairs are accessed in turn and are output to the operator console.

If this script runs and encounters a macro called `M1` with the value `M1Value`, the **Log** view would contain the following text:

```
Execute Script: Macro name | value:  M1 | M1Value
```

The process that does this is `groovyExample.xml`, which is provided with this book.

# Console logging with Groovy

Sometimes debugging Groovy is a problem and code instrumentation is the only way out. The following fragment of Groovy prints the output to the console:

```
operator.logNote ("Macro name | value:  " + macroName + " | " +
macroValue);
```

This is invaluable when getting Groovy scripts working in the RapidMiner Studio environment. Note that this was shown in the example Groovy script in the previous screenshot.

# Regex tools

**Regular expressions** (**Regex**) require hard work, but they make a huge difference when working with RapidMiner Studio operators. They can be used in many ways and the following list gives us a flavor:

- Extracting features from structured or unstructured data to create new attributes
- Selecting attributes within an example set to form a set for a common procedure such as deletion, renaming, looping, or type changing
- Defining how to change the names of attributes when renaming
- Selecting examples within an example set for filtering
- Replacing the values of attributes with replacement values
- Restructuring example sets with the `Pivot` and `De-Pivot` operators

Clearly, Regex is used in many places and to become proficient with its usage, powerful tools and debugging facilities are needed.

An excellent free tool is **Expresso** available at `http://www.ultrapico.com`. Of course, many other tools exist and there are many resources on the Internet that give working examples for common regular expressions.

The interactive tool provided within the RapidMiner Studio GUI can also be very helpful.

# Using XPath effectively

XPath is extremely useful when parsing and extracting information from structured data, which is encountered all the time when exploring real data. As with regular expressions, it is important to become proficient and inevitably this means getting access to powerful tools and debugging facilities.

Fortunately, there are many tools, editors, and online resources that can be used.

A good free editor that I use is Notepad++ available at `http://notepad-plus-plus.org/`. This has a number of plugins including an XPath evaluator.

In addition, Google spreadsheets provide a way to build real-time XPath queries, and W3Schools (`http://www.w3schools.com`) gives a good set of tutorials for XPath and many other Web technologies.

# Summary

Debugging is a fact of life and the sharper the tools, the better. This chapter has given a brief overview of some of the techniques that I have found useful.

The next chapter gives a summary of where we have got to, and also points out some of the facts that are directly related to data mining that have not been covered, but which could offer the reader a springboard into further reading or research.

# 11
# Taking Stock

Let us take stock of where we are. One of the aims of this book was to give an overview of the process of data exploration to demonstrate that there is a lot to it, but this need not be daunting especially if the explorer is armed with sharp yet flexible tools, and has the ability and confidence to use them.

The methodology first mentioned in *Chapter 1*, *Setting the Scene* shows how a typical data mining activity is an iterative process, but with a general direction starting from the requirements and ending with the benefits. By its nature, a book has to present things in a linear order but I hope it is clear that the chapter ordering will not generally apply when real-world data is handled. Indeed, all the stages do not have to happen at all; for example the requirements could be met simply by importing and visualizing data.

Another aim is to provide real examples in sufficient detail to download with the processes. This allows them to be re-used without having to invent them first. While it is not necessarily time consuming to create RapidMiner Studio processes (the clue is in the name), sometimes a helping hand or a `Hello World` example to start from can save a lot of time. The product is huge, so knowing everything is a challenge and knowing how all the operators fit together is another. What is perhaps missing is context about what to do when a certain type of activity needs to be performed. This context, in the form of the exploration of data is one route into the book, and the processes and techniques that are shown should allow easier re-use.

The final aim is to show what could be possible. Sometimes seeing something being done in a different or unexpected way leads to new ideas and can certainly save time. I am certain there will be new examples invented as well as new approaches since I am by no means the only RapidMiner Studio practitioner; there are plenty of creative people out there. This is in fact one of the very desirable things about RapidMiner. Once you have arrived at a certain state of knowledge and confidence, all processes become extremely easy and in fact almost second nature. This frees the mind from having to worry about whether something is possible and allows the problem itself to be solved. This is the best bit about data exploration and data mining; no two problems are the same and there are tremendous opportunities to be creative.

# Exploring new techniques

Of course, there is more to RapidMiner in general than this book has covered and there is certainly more to data exploration. The interested reader is encouraged to keep finding more because, in my experience, new techniques lead to new insights and results in a self propelling virtuous circle. If only there was more time in a day.

The following sections give you a short list of areas that are well worth looking into.

## Time series

There are many examples of time series in the real world. Examples include stock prices, tree ring data, temperature records, sunspots, and audio files. RapidMiner Studio has an extension for series data and in fact, the `Window` operator is a part of it. This book has only scratched the surface of time series.

## Web mining

Text mining was briefly touched upon but there is a great deal that could be done to explore data derived from web pages or feed APIs. There will never be a shortage of data from the web.

## Using R

RapidMiner Studio integrates with R, the de facto standard package for statistical analysis in the academic world and increasingly outside it. R has a fantastic range of packages covering a huge array of subject areas with new ones right at the leading edge being added all the time. An R script can easily be integrated into RapidMiner Studio, so if there is something missing from RapidMiner Studio (and there is sometimes) it is almost certainly available in R.

R also has very good graphics and there is no reason not to use it as part of an exploration process. There is a downside to this. R has a huge learning curve but the value is so great you might as well start now.

## Java or Groovy

RapidMiner Studio is built using Java, and given that the community edition is open source, it would be completely possible to make changes to the software to solve a particular problem as well as give back to the community so that everyone benefits. This book has deliberately not looked at Java partly because I am not an expert but mostly because it would have put people off.

Having said that, this book does have some Groovy examples and there will be times when only Groovy will do. So, I do encourage this to become one of your areas of knowledge.

# Third-party components

Rapid-I, the company behind RapidMiner, operates a market place where third-party extensions are available. This is integrated with the RapidMiner GUI and it is worth visiting this location to see if there is anything there that could prove useful.

# RapidMiner Server

Rapid-I also produces RapidAnalytics. This is a server-based solution that provides a repository location and an environment for remote execution of processes that integrates with the RapidMiner Studio GUI. This can be very useful if you have a powerful server available. RapidMiner Server allows a long running process to be initiated on the server, which should complete more quickly; but while you are waiting, you can work on something else.

There is a lot more to RapidMiner Server, such as scheduled process execution, custom web-based reports, and user management; but it is beyond the scope of this book to go into detail.

# Where to go next

It is all very well reading things in a book. What really matters is to practice on real data. A good step is to enter some data mining contests that appear on the Internet. Your place of work will almost certainly have data; if you can add value to it to solve a business problem you will get a lot of interest. The Internet itself has more sources of data with more being added every day. Finding a new insight into public sources of data, even through a simple visualization, may get you noticed.

In short, there is a lot of data out there just waiting to be explored. I earnestly hope this book has whetted your appetite and given you the ideas, tools, and confidence to get stuck in.

# Index

## Symbols

**3D scatter plots 34, 35**

## A

**aggregation 98-100**
**aggregation attributes dialog 99**
**attribute name field 95**
**attribute relationships**
about 32
deviation plotter 35, 36
parallel plotter 35
Quartile color plot 38
Scatter 3D color 34, 35
scatter plot 32-34
**attributes**
about 14
generating 50, 51
Map operator, using 59
removing 108
renaming 59
Replace (Dictionary), using 60, 61
Replace operator, using 60
selecting, models used 114-119
weighting 111-114
**attribute values**
replacing 59
searching 59

## B

**Big Data 107**
**block plots**
using 45, 46
**Build SQL Query dialog 26**
**Business understanding 10**

## C

**Charts View 29**

## D

**data**
cleaning 12
logging, in RapidMiner 134, 135
missing values, detecting 12
transforming 93
visualizing 13
windowing 104-106
**databases**
accessing 25
**databases, accessing**
Read Database operator 25, 27
**data functions 52, 53**
**datatypes 11**
**data visualization**
starting with 29
Statistics View 30
**data volumes**
increasing 68
measuring 10, 11
**de-pivoting**
using 102, 103
**De-Pivot operator 102**
**Detect Outlier (Densities) operator 73**
**Detect Outlier (Distances) operator 69-72**
**Detect Outlier (LOF) operator 74, 75**
**Detect Outliers (COF) operator 75, 76**
**deviation plots 35, 36**

## E

**Edit Enumeration dialog 26**
**example 14**
**example relations**
about 43, 44
block plots, using 45, 46
histograms, using 44
**example set 14**
**Execute Process operator 125, 126**
**Execute Script operator 137**
**Expresso 138**
**Extract Log Value operator 135**
**Extract Macro operator 30**

## F

**file reading**
about 17-19
complete lines, reading 21
field delimiter 20
large file, splitting 23, 24
multiple attributes, reading 21, 23
Read CSV, using 17
**Forward Selection operator 115, 116**

## G

**Generate Attributes operator**
about 50, 51
data, extracting 54
date functions 51-53
regular expression functions 53, 54
regular expressions 54-57
XPath 57-59
**Generate Extract operator 54**
**Generate Macro operator 24**
**Groovy scripts**
about 136
macros example, outputting 137
**Guess Types operator 115**

## H

**handling options, missing data**
attributes, deleting 90
examples, deleting 90
ignore option 89
manual editing 89
modeling 91
root cause 89
single values, imputating 90
**histograms**
using 44

## I

**ID 15**
**Import Configuration Wizard button 19**

## J

**Java Database Connectivity (JDBC) 25**

## L

**label 15**
**Local Outliers Factor (LOF) operator 73**
**Log operator 117, 135**
**Log to Data operator 115, 127**
**Loop Attributes operator 30**
**Loop Batches operator 24**
**Loop Parameters operator 124**

## M

**macro name (macro) 96**
**macros**
about 14
using 27, 28
**manual inspection**
about 63-67
data volume, increasing 68
**Map operator 59**
**material**
accompanying 15
**memory**
adding 129
**Missing at random (MAR) 78**
**missing completely at random (MCAR) 78**
**missing data**
about 77
categorizing 79, 83-87
effects 88
handling, options 88
types 78

**missing data categorization**
about 79-82
MAR data, finding 85, 86
MCAR data, finding 83, 84
NMAR data, finding 86, 87
**missing data, types**
Missing at random (MAR) 78
missing completely at random(MCAR) 78
not missing at random 79
not missing at random(NMAR) 79

## N

**Naïve Bayes operator 117**
**new attributes**
creating 94-97
**new techniques**
exploring 142
Groovy 142
Java 142
RapidAnalytics 143
R, using 142
text mining 142
third party components 143
time series 142
**nominal useless below parameter 111**
**Notepad++ 138**
**not missing at random (NMAR) 79**

## O

**operator 14**
**outliers**
about 63
handling, rules 68

## P

**parallel plots 35, 36**
**parallel processing 130**
**performance**
estimating 121
measuring 121-129
**Performance operator 125**
**pivoting**
using 100, 101
**Principal Component Analysis (PCA) 112**
**Print to Console operator 135, 136**
**process**
about 14
restructuring 131
**process framework**
about 8, 9
Business understanding phase 9
evaluation 10
modeling, Evaluation, and Deployment phases 10
modeling step 9, 10

## Q

**Quartile color plot 38**

## R

**R**
using 142
**RapidMiner Server 143**
**RapidMiner Studio**
about 7, 8
attribute 14
breakpoints 133
console printing 135, 136
databases, reading 25
data, loading into 17
data, logging in 134, 135
example 14
example set 14
Groovy scripts 136
ID 15
label 15
macro 14
operator 14
process 14
repository 14
role 15
type 14
**Read Database operator 25**
**Recall operator 131**
**regular expression functions 53, 54**
**regular expressions 54, 55, 57**
**Regular expressions (Regex) 138**
**Reload data button 19**
**Remember operator 131**
**Remove Useless Attributes operator 109, 110**

**Replace (Dictionary) operator 60, 62**
**Replace operator 60**
**repository 14**
**Retrieve operator 115**
**role 15**

## S

**Sample (Bootstrapping) operator 108**
**Sample operator 107**
**sampling**
  used, for examples removing 108
**scatter plots 29, 32-34**
**Select by Weight operator 113**
**SetAttributes.value operator 124**
**SetExamples.value operator 124**
**Simple Object Access Protocol (SOAP) 58**
**StartTheClock operator 126**
**statistics parameter 96**
**Statistics View**
  about 30
  for Iris dataset 31, 32
**Store operator 131**

## T

**text mining 142**
**time series 142**
**time series data**
  about 39
  series, plotting 39-41
  survey plotter, using 42, 43
**type 14**

## U

**useless attributes**
  removing 109, 110
**useless below parameters 111**

## V

**Validation operator 117**
**visualization 29**

## W

**windowing 104**
**Window operator 142**
**Write CSV operator 24**

## X

**XPath**
  abou 57-59
  using 138
**X-Validation operator 130**


## Thank you for buying

# Exploring Data with RapidMiner

## About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

## About Packt Open Source

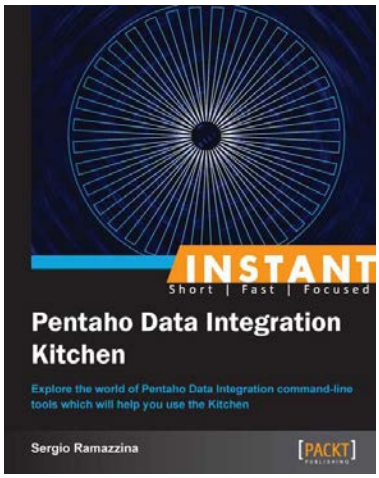
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING




## Instant Pentaho Data Integration Kitchen

ISBN: 978-1-849696-90-6 Paperback: 68 pages

Explore the world of Pentaho Data Integration command-line tool which will help you use the Kitchen

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. Understand how to discover the repository structure using the command line scripts
3. Learn to configure the log properly and how to gather the information that helps you investigate any kind of problem

---

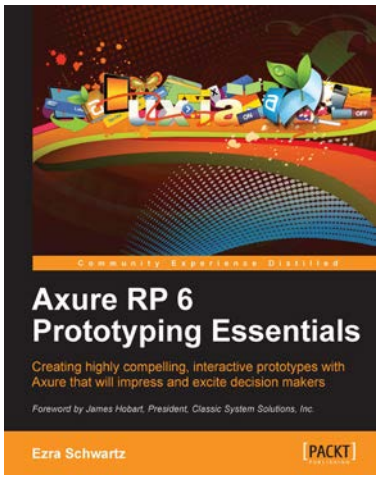

## IBM Cognos Insight

ISBN: 978-1-849688-46-8 Paperback: 142 pages

Take a deep dive into IBM Congos Insight and learn how this personal analytics tool can be integrated with other IBM Business Analytics products

1. Step-by-step, how to guide, for installing and configuring IBM Cognos Insight for your needs
2. Learn how to build Financial, Marketing and Sales workspaces in Cognos Insight
3. Learn how to integrate and collaborate with IBM Cognos Business Intelligence

[PACKT] open source
community experience distilled
PUBLISHING



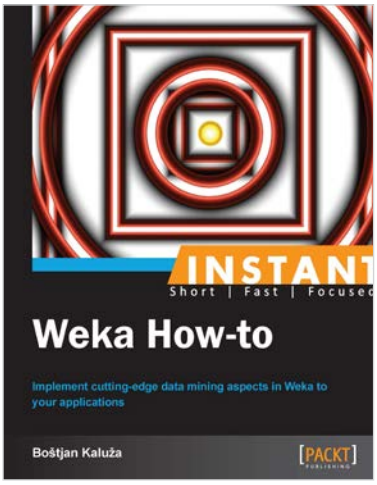

## Axure RP 6 Prototyping Essentials

ISBN: 978-1-849691-64-2 Paperback: 446 pages

creating highly compelling, interactive prototypes with Axure that will impress and excite decision makers

1. Quickly simulate complex interactions for a wide range of applications without any programming knowledge
2. Acquire timesaving methods for constructing and annotating wireframes, interactive prototypes, and UX specifications
3. A hands-on guide that walks you through the iterative process of UX prototyping with Axure

---




## Instant Weka How-to

ISBN: 978-1-782163-86-2 Paperback: 80 pages

Implement cutting-edge data mining aspects in Weka to your applications

1. Learn something new in an Instant! A short, fast, focused guide delivering immediate results
2. A practical guide with examples and applications of programming Weka in Java
3. Start with the basics and dive deeper into the more advanced aspects of Weka
4. Learn how to include Weka's machinery in your Java application